Chapter 10. Jini: Sun's Technology of Impromptu Networks

- Examples of Jini
- Where Did Jini Come From?
- Our Working Jini Example
- Basic Jini Concepts: "Discovery, Join, and Lookup Oh My!"
- Getting Started with Jini
- Lets Get to the Code!

The introduction of the iMac was no small factor in Apple's rejuvenation. It was also a revolutionary computer. The iMac's success came from its appeal to regular folk, those ordinary people who cared less about the size of its hard drive and more about the color of the case. In short, the iMac was marketed (and purchased) as an appliance. People could buy it, bring it home, plug it in, and it just worked. It worked right out of the box, no hassles, and no complicated instructions. That concept is what Jini is all about.

Jini is Sun's solution for creating common, everyday, networking appliances that just "plug and work." I use the word "appliances" instead of "application" because an appliance is a simple piece of technology that everyone can use. An application, on the other hand, is a technology that often requires development of a skill set in order to use it productively. Few people would consider the ability to start a dishwasher a skill set.

Examples of Jini

So what is Jini? Jini is a paradigm for how the service providers and service consumers should interact on a network. It is a layer of architecture that is dependent on a variety of other pieces of architecture as described in Figure 10-1. Sun's current implementation of Jini, Jini 1.0, is written completely in Java, but not all pieces of the Jini architecture must be. The best way to introduce Jini is to describe the technology in use. Sun's promotional literature is full of interesting examples, and examples of the technology aren't hard to dream up. The most common example is that of a printer.

![Figure 10-1. The layers of architecture needed by Jini.](image)

*The Jini specification does not require RMI; Sun's implementation of Jini does. Technically, Jini is a specification and doesn't require Java at all. The only available implementations of Jini use Java, however.*

ABC Industries recently doubled the size of its office staff from 12 people, to 24. Laura Cogswell, ABC Industries office manager, noticed a situation brewing over the limited printing resource of their single laser printer. The lines were long, and the printer never seemed to stop spitting out paper. It was obvious that ABC Industries simply needed an additional printer.

Laura took a trip down to the local office supply depot where she casually purchased an additional name brand laser printer. Getting it back to the office, Laura took it out of the box, plugged in the power chord, plugged in the networking cable, and switched it on. Within moments, the printer came to life and began printing out reports. The two printers continued to work side by side satisfying the printing needs of ABC Industries. Several members of the office staff were impressed to find the new printer, which they hadn't even known existed there when they clicked the print icon, had automatically printed their reports. How was this possible? Both the printers, and the PCs of ABC Industries, were "Jini Enabled."
What's important about the last example? Well, the first and foremost thing is that Laura is not an IT specialist. She's an office manager. The same person who might arrange to purchase copier paper, a fax machine, or even break-room supplies. She treated the printer problem the same way she might have treated a problem with a refrigerator that was too small, or a coffee machine that didn't make enough coffee. She went out and got another one, plugged it in, and went about her business. She did not have to deploy a set of drivers across the company's computer system (incidentally consisting of UNIX boxen, Macintoshes, and Windows machines), nor did anyone have to set up the new printer on his or her PC.

The rest of the chapter is dedicated to explaining the basics of Jini and a little about how Jini can make this possible. This chapter is, by no means, meant to be an in-depth study of Jini, but it is intended to provide an introduction to the technology and a glimpse at how to get started with it.

Where Did Jini Come From?

Jini is Sun's continued dedication to the founding principles of a language called Oak. Oak was intended as a platform-independent, simple, object-oriented approach to working with "smart appliances," like set top boxes, clocks, microwaves, cell phones, you name it. The problem was that companies in the smart appliance market found themselves spending lots of time and money supporting a myriad of software environments. It seemed that each appliance they developed used different hardware that either had its own software environment or was different enough from the "standard" programming environments to require individual attention.

Oak was intended to help companies deploying different smart appliances concentrate on the appliance itself and not on the overhead of the software environment. Oak was supposed to be write once, run anywhere. Sound familiar? It should because out of Oak came Java. Java gained its popularity because it offered platform independence for application developers on personal computers. You can write an application on a Mac and have the exact same code run on a Windows machine, or a UNIX box. Over time, Java technology has become more robust and mature. Sun never abandoned its roots in the smart appliance market. Instead, they are redefining what people think an appliance is, and bringing the full weight of Java (both its strengths and weaknesses) to bear on the genre. Sun's reference implementation of Jini is a software architecture layer that makes extensive use of Java RMI, which makes possible the concept of a "plug and work" network appliance.

Our Working Jini Example

For the rest of this chapter, we will discuss Jini in terms of a single example, that of a Morse code printer. The Morse code printer itself is a small black box out of which come two cables, one for the Ethernet network, and one for power (some networking solutions use standard 60-Hz ac power lines as their medium rather than Ethernet cables. In that instance, we would only need one cable, the power cord!). Also affixed to the box are two large LEDs, one red and one green. Inside our little network printer is a complete JVM with all of the appropriate Java 2.0 and Jini 1.0 libraries. The printer works by translating messages sent to it into "dashes" and "dots" flashed out by the green LED according to the standard Morse code protocol. In this way, a message consisting of "SOS" would cause the green light to pulse three times quickly, three times slowly, and then three times quickly again.

Morse code was invented over a century ago and wasn't originally conceived of for use as a Jini-based print server (although it does work surprisingly well for this). During the course of a message translation, it is possible that a situation could arise that Morse code is unable to account for. In this case, the red LED will flash, indicating that an error has occurred, and the printer will skip the untranslatable characters and continue on.

In our example we will assume that a standards organization has blessed a particular Java interface to network printers and that this interface is well known. Any entity wishing to use the services of the network printer can do so by utilizing this well-known interface. In our example, an ambitious Java programmer has created a simple client utilizing this interface in the hopes of communicating with our network printer.

The idea, of course, is that someone can walk into a room, plug the "Jini'fied Morse code printer" into the network, go over to a computer that was already running and on the network, activate the client application, and be able to immediately print to the new printer. How could this happen? Let's look at Jini's basic infrastructure.

Basic Jini Concepts: "Discovery, Join, and Lookup Oh My!"

In any Jini community, sometimes called a Djinn, federation, or collectives, there exist three main elements: a service, in our example the Jini'fied Morse code printer; a client that consumes the desired service, like our Java PrintClient application; and a Jini Look Up Service (JLUS) that acts as a coordinator to help the Jini client find the Jini service it is looking for. To see how these three Jini elements interact with each other, let's look to our example.
In the beginning, we plug our patented "Jini'fied Morse code printer" into the network and switch it on. At first, it is unaware that anything else on the network exists. Luckily, Jini has a protocol for getting in touch with other Djinn; it's called Discovery. In our example, Discovery takes the form of a message broadcast to our entire local network asking for any available JLUSs to identify themselves. Each JLUS that hears this broadcast responds by giving our network printer a representative of the JLUS. This representative takes the form of a ServiceRegistrar object. The ServiceRegistrar object works as a proxy to the JLUS. Any work that we want to do with the JLUS we can do by invoking methods on the registrar object.

Figure 10-2 illustrates the concept of Jini Discovery. Here the service finds the local JLUS and obtains an object that functions as an interface.

To simplify things, only one JLUS responds to our printer's discovery effort. In reality there could be multiple JLUSs out there, or none at all! (In the last case, we could include in our service implementation code that would create its own JLUS. For now, we just assume that there will be at least one.)

Next we want to tell JLUS all about the great service our Jini'fied Morse code printer offers so that others can take advantage of our printer's availability. This process is referred to as the Join protocol.

In order to join a Djinn, our service has to do two things. First, it must create and provide a proxy object. A proxy, in general, is an agent through which someone or something interacts with another, a go-between. Here, the proxy object is exactly its namesake. It provides the mechanism through which an interested client will communicate with our printer.

The proxy can be any Java object! It could be a full-blown Java GUI application using TCP/IP sockets to "talk" to its server, or it could be an object implementing a simple Java RMI interface. The actual protocol used between the service and the client depends upon the particular implementation of Jini. The interface is defined by whatever the proxy object is. In our example, our printer will use Java RMI to communicate with any interested clients, so the proxy object will be something that implements PrintServiceInterface.

The second thing we should do as part of the Join process is define the set of attributes that our service possesses. These could be anything from defining the name of the printer, or the location of the printer, to expressing all the classes that our proxy implements. Why do this? Well, by providing more information about our service, prospective clients have more information from which to say why our service may or may not be the best for them.

Both pieces of information are packaged up and sent to the JLUS by providing them as arguments in one of the ServiceRegistrar's methods. The JLUS receives these items and stores them for later. This is shown in Figure 10-3.
In Jini Join, as shown in Figure 10-3, our service first constructs a service item describing itself as JLUS and then gives that service item to the JLUS, thus officially joining the local DJinn.

This is all well and good, but what happens if our Jini’fied Morse code printer gets "accidentally" kicked across the room by a frustrated red–green color-blind user? Well, in its battered, disconnected state it certainly isn't available to print messages anymore. How does the JLUS know that our printer isn't available?

One possible methodology would be for the JLUS to continuously poll the service to see if it's still alive. The greater the frequency of polling, the smaller the period of uncertainty about the status of a service and the quicker the response of the Djinn as a whole in dealing with the loss of a service. This approach, however, puts a great deal of burden on the JLUS and also creates quite a lot of network traffic.

As Figure 10-4 shows, the Jini Service continually tracks time elapsed on its lease and renews each lease cycle before the expiration of the lease.

Jini actually deals with this by assigning a "lease time" to a service. In an abstract sense, a Jini lease is an agreement between the JLUS and the Jini Service that guarantees that the Jini Service will be up and available throughout the duration of the lease. Jini puts the burden of renewing the lease squarely on the Jini Service, not on the central JLUS.
When the lease time period expires, the JLUS will simply remove the information it has about the subject service from availability to the Djinn. It will not, of its own accord, inform the subject Jini Service that the lease has expired. It is up to the Jini Service to enquire about the lease time granted to it by the JLUS. It is also the responsibility of the Jini Service to track the amount of time transpired and to renew its lease when appropriate. This is sort of a feed forward system. The Jini Server pushes the lease renewal effort. This is illustrated in Figure 10-4.

**Client**

Now that our printer is plugged in and participating in the Djinn, let's look at Jini from a service consumer's perspective. Remember that right now, in our example, a service item describing our Jini'fied Morse code printer exists on the JLUS. Also remember that the service item contains a proxy object for our Jini-enabled printer.

For the purposes of demonstrating Jini, we have a client whose only job is to find the Jini'fied Morse code printer and send messages to it. The client doesn't have to be a stand-alone Java program, however. It could just as easily be part of an operating system. Such integration would provide our plug-and-work Jini printing capabilities to every application running on that operating system. For the purposes of our example, we will just consider the stand-alone application case.

When the client is run, it has to find the JLUS, just like the service did. It goes through exactly the same Discovery process the service did, eventually obtaining a ServiceRegistrar object for each JLUS that responds. Again, in our example, only one JLUS exists and responds.

Next, the client has the task of using the JLUS to find the desired service. To do this, the client describes the desired service in any one of several ways. The general method is shown in Figure 10-5. The client provides a template against which the JLUS can stack up potential services and find the one that's a match. This template describes desired server characteristics including attributes that services may have defined, such as their name or location, and interfaces that services must match. Our client could try to look up the service by the "well-understood" PrintServiceInterface it should implement. In our example, however, the client actually looks for a match by name. It fills out the appropriate attribute in the template, and gives it to the JLUS by invoking a method on the ServiceRegistrar object. Again, the service registrar is the interface to the JLUS for the client as well as the service.

*Figure 10-5. The Jini Client constructs a template identifying the desired service, and the JLUS matches this template against all registered service items and returns the proxy object from the matching service item.*

The JLUS then uses the template to find potential matches, which it does in our example. It then takes the proxy object from the matching service item and returns that as the search result to the client. Had the search been unsuccessful, it would have returned a null instead.
The client receives the proxy and then uses it to communicate directly with the service, leaving the JLUS entirely out of the picture at this point. All future communication between the client and service can now happen directly. In our example, the proxy object given to the client simply uses Java RMI to remotely invoke the print method on our service. We could have chosen as our proxy to have a little program that would simply have streamed data to our print service through a TCP/IP socket.

The user makes the client send the message "SOS" and watches as the dots and dashes get flashed out on the Jini'fied Morse code printer.

It is interesting to note, however, that the JLUS itself follows a similar protocol. Both the client and the service received a ServiceRegistrar object through which they communicated with the JLUS. The ServiceRegistrar object is the JLUS's proxy!

Getting Started with Jini

Before you can delve into making your own Jini services, clients, or even look-up services, you must overcome a few preliminary hurdles before you set up your development environment. These hurdles are neither overwhelming nor are they trivial, however. They often represent an initial challenge to getting started with Jini.

- Obtain the latest Java SDK for your OS (should be 2.0 or higher).
- Install the Java SDK.
- Set up your Java environment as appropriate.
- Test your Java environment to make sure it works.
- Obtain the latest Jini SDK (should be 1.0 or higher).
- Install the Jini SDK.
- Adjust your classpath as appropriate.
- Test the Jini installation to make sure it works.
- Keep the Jini Service and Jini Client environments independent from each other during development. This avoids accidental and invalid dependencies on resources that otherwise wouldn't be shared.

The Java and Jini SDK are available by download directly from Sun [http://java.sun.com](http://java.sun.com). These both include some setup instructions and example applications, which you can run to make sure everything is working. It's just good common sense to make sure you have a working initial setup before you start writing and trying to debug your Jini code.

The last point is not so much a setup step as a development consideration. Keep in mind that, when you develop for Jini, you're developing for a distributed computing environment. When the Jini Service is first brought online, it will be completely unaware of the client. Likewise when the client is first brought up, it is unaware of the service. These two entities make initial contact through a JLUS and, before this, can have no shared resources (data files, code snippets, whatever) that aren't supposed to be explicitly built into both.

The resources mentioned at the end of this chapter can lend considerably more help in dealing with these issues than any discussion possible in the space allotted here. I encourage you to peruse them.

Let's Get to the Code!

The following code only looks at the Jini portions of our examples. The Java code comprising the LED device driver and Morse code translation system are not central to understanding how Jini works and aren't presented here. Those interested in further exploring these pieces of software can obtain the original source online at [http://watson2.cs.binghamton.edu/~steflik/jini](http://watson2.cs.binghamton.edu/~steflik/jini) or on the accompanying CD-ROM.

Implementing the Jini Server

One of the first stipulations we made about our Jini'fied Morse code printer was that it implemented an agreed-upon well-known interface. In practice, this is currently the most difficult part of Jini's promise. There have been ads showing Jini-enabling digital watches to communicate with toasters and other currently brainless home appliances. The problem here (besides understanding why you would want to do such a thing in the first place) is that this means all the people who make digital watches and all the people who make toasters have all agreed on what their equipment's interface will be. At the time of this writing, there are efforts underway to define such well-known interfaces for printers and what not, but they have not come to a conclusion, so for our purposes, we will assert that such a well-known interface already exists and looks like this:
import java.rmi.*;
import java.io.*;

public interface PrintServiceInterface extends Remote {
    public boolean print(String printString) throws RemoteException;
}

This code is standard Java RMI. It defines an interface called PrintServiceInterface that defines the relationship between the client and the service. The client can invoke the method print on any object implementing this interface and pass to that object a string. The object is allowed to pass back to it a boolean or throw a remote exception, that is, the whole interface. The idea, of course, behind this interface is that the object implementing this interface will be the proxy from our printer and that it will take the string passed to it and flash it out in Morse code. If it is unable to do so, for some reason, our printer should inform the client that there was an error by passing a false back as the return value from the method. Hopefully, however, it will be successful and pass a true value as the return value from the method call.

Let's look at the Service code:

/*----------------------Imports-------------------------------------*/
import java.io.*;
import java.rmi.*;
import java.rmi.server.*;
import com.sun.jini.lookup.*;
import net.jini.core.entry.*;
import net.jini.core.lookup.*;
import net.jini.core.discovery.*;
import net.jini.lookup.entry.*;
import com.sun.jini.lease.*;
/*------------------------------------------------------------------*/

public class PrintService extends UnicastRemoteObject
    implements PrintServiceInterface, ServiceIDListener, Serializable
{
    LightDriver ld;
    MorseTranslator mp;
    // Print Service Constructor
    public PrintService() throws RemoteException
    {
        super();
        try
        {
            ld = new LightDriver();       // create light driver
            mp = new MorseTranslator(ld); // connect translator to driver
        }
        catch (Exception pse) { pse.printStackTrace();
            System.exit(0);
        }
    }
    // This method is what satisfies the PrintServiceInterface,
    // requirement it is through RMI that the client remotely calls
    // this method to deliver a string to the server.
    public boolean print(String printString) throws RemoteException
    {
        try
        {
            System.out.print("PRINT SERVICE, Printing: " +
                printString + " Length= " + printString.length());
            mp.doTranslation(printString); // morse print message
            return true;                    // return success
        }
        catch (Exception pe) {pe.printStackTrace();
            return false;             // return failure
        }
    }
}
A Walk Through the PrintService Code

So what's happening here? Well, what do we know about a Jini Service so far? We know that a Jini Service must go through the Discovery process to find a JLUS. We know that once it's found the JLUS it must go through the Join process to make its services available to the Djinn. We also know that as part of the Join process the service has to provide a proxy object and somehow describe itself to the JLUS so that it can be found by interested clients. Does all this happen here? Yes, let's dissect the code a little.

After the initial includes, we can see that our constructor instantiates a LightDriver and a MorseTranslator object. These two objects simply provide the mechanics of making a message flash out in Morse code; as mentioned before, we won't discuss them in detail here. What's important is that our service constructor initialize the mechanics necessary to implement the service. No big deal yet.

The next thing we stumble upon as we traverse the code is the Print method. Aha! This is the method that our well-known interface said we had to have; indeed, if we jump back a couple of lines, we see that our class does in fact implement that interface. Examining the message reveals that basically all our service does is pass the string argument to a method on our MorseTranslator object and then pass back a true. Ideally, we should have some way to determine whether our message was printed or not and pass back a true or false accordingly; nonetheless, we
satisfied the requirements of the interface. OK, but that's only useful once you are actively participating in the Djinn. We haven't gotten to that code yet, so now what?

Continuing through the code we bump into an empty little method entitled service-IDNotify. It doesn't do anything, so we must need it to satisfy an interface, and sure enough we see that our class also implements the ServiceIDListener interface. What's that for? This is a piece of the Discovery process as we'll see in a moment.

Next we hit the main of our PrintService class. The next significant thing we do here is instantiate an RMI security manager. This is done so that we will be able to download remote code and use it locally. Why would we do that? We need to do that in order to obtain the ServiceRegistrar object from the JLUS.

Next we have to do a few things to explicitly set up for Join the discovery process. We create our proxy object, an instantiation of our PrintService class, and we create an array of Entry objects, although we only hold a single element. These entry objects are used to describe our service. In the very next line we define a name object, which we set equal to what we're calling our service. In our example, this is what is going to be used to identify our service by the client. We'll talk more about that later. For now we have proxy object, and we have our service description.

Finally, we instantiate a JoinManager. What is this guy? Well for us, it's the lazy coder's (and simplest) way to handle all the service's basic Jini responsibilities. Let's look at this.

It takes four arguments in its constructor: a proxy object, the Service Item, a Service-IDListener, and a LeaseManager. There's a lot of powerful stuff all packed into one class! We pass it our brand-new PrintService object as the proxy, which makes sense. We then hand it our attributes, which we knew we had to do, but for the third object we pass our PrintService again. What is a ServiceIDListener anyway?

Well, in Jini, all services are assigned an ID when they first register with a JLUS. The ID that is given to your service is guaranteed to be completely unique. There shouldn't be another service in the whole world that has that same ID. The idea is that your service is supposed to remember this ID and pass it as all the other JLUSs that it may register with as time goes by. In that way, there is a single common way to recognize a specific service regardless of what JLUS a client finds it on. The method for passing this ID back to a service is through a ServiceIDListener. Each ServiceIDListener has to implement the ServiceIDNotify method, which ours does. In reality, this is simply an RMI callback from the JLUS saying, "This is your serviceID." For our example, we don't do anything with this ID. In general, you will.

That leaves only the final argument. Here we create something called a Lease-RenewalManager. As you can guess by the name, that class is all we need to keep our lease fresh and our service part of Djinn. You can see that in one single object we've delegated Discovery (it downloads and utilizes the ServiceRegistrar object), Join, and lease renewal all away! This works great in our simple case, but there are many other circumstances when you will want a more fine-grain control over how your service participates in a Djinn. For that to happen, you won't be able to just use a JoinManager. You will have to do a little more work. Because this chapter is just intended to get you introduced to Jini, we don't cover that here, but all the references mentioned at the end of this chapter can assist with that.

What's after that? Nothing, our main ends, and our service exits. Right? Almost; it turns out that the Join manager creates some threads to handle all the activities that we've just delegated to it. As long as it's alive and active (which is until the power is pulled), our service will continue to run. If you wind up implementing Discovery, Join, and lease managing yourself, you will also have to include a way to keep your service from just exiting. A simple while(true) {} works pretty well.

Implementing the Jini Client

Our Jini Client is a simple stand-alone Java application that finds our Jini'fied Morse code printer service and prints out "Hello World." Somewhere in this code, we will have to create a template to identify the service we're looking for, perform Discovery to find a JLUS, do a lookup on that JLUS to actually find our service, and finally use that service through its proxy.

```java
/*------------------------ PrintClientImports -------------------*/
import java.io.*;
import java.net.*;
import java.rmi.*;
import com.sun.jini.lookup.*;
import net.jini.core.entry.*;
import net.jini.core.lookup.*;
```
import net.jini.core.discovery.*;
import net.jini.lookup.entry.*;
import net.jini.discovery.*;

/*-------------------------------------------------------------*/
public class PrintClient implements DiscoveryListener {

    public ServiceRegistrar[] registrars; // holds list of JLUSs'
        // registrars
    public boolean JLUSfound = false;     // will trigger lookup
        // process

    // This method is required to implement DiscoveryListener.
    // This is how our client will be notified when a
    //   ServiceRegistrar object is received from a JLUS
    public void discovered(DiscoveryEvent ev) {
        // Obtain the array of registrar objects from all available
        // LUSs'
        registrars = ev.getRegistrars();
        JLUSfound = true;
    }

    // Used to deal with the situation when a JLUS is discarded by
    // our discovery object.  We don't use it here, but more
    // robust code would.
    public void discarded(DiscoveryEvent ev) {
        //just satisfying DiscoveryListener interface
    }

    public static void main(String[] args) {
        LookupDiscovery discovery;   // discovers available LUS
        ServiceRegistrar ourreg;     // holds the test registrar
        LookupLocator lookup;        // used to lookup service
        Entry[] desiredattribs;      // attributes desired in service
            // a template the service must fill
        ServiceTemplate desiredtemplate;

        Object serviceobject;       //result of the search
        PrintServiceInterface printer; //handle to hold found service
        PrintClient    printclient;     //This holds the instantiation
            //of our client

        try {
            // Set the security manager for handling downloaded code:
            // needed to obtain the ServiceRegistrars and the servers
            // proxy object
            System.setSecurityManager( newRMISecurityManager() );

            // Perform Discovery Process for all JLUS containing the
            // public group
            discovery = newLookupDiscovery(new String[] {""});

            // add a DiscoveryListener to recieve the ServiceRegistrar
            // objects our LookupDiscovery finds.
            printclient = new PrintClient();
            discovery.addDiscoveryListener(printclient);

            // create a description for the service we are looking for
            // must have the name PrintService...
            desiredattribs = new Entry[1];
            desiredattribs[0] = new Name("Jini'fied Morse Code
                Printer");
            Class[] clArray = new Class {PrintServiceInterface};
            desiredtemplate = new ServiceTemplate(null, clArray,
                        desiredattribs);

            while (true) {
                // begin while
            }
        }
    }
}
if (printclient.JLUSfound)
    {                                     // begin if found
        printclient.JLUSfound = false;     // reset flag
        System.out.println("PrintClient: " +
           printclient.registrars.length +
           "Jini Lookup Service(s) Found.");

    // Go through the complete array of registrar objects
    // lookup on each one to find the PrintService
    // if found print out "Hello World!"
    for (int i=0; i<printclient.registrars.length; i++)
    {  //begin for
        lookup = printclient.registrars[i].getLocator();
        System.out.println(
           "PrintClient: LUS found at " +
           lookup.getHost() + " on port " +
           lookup.getPort());

        //perform the actual Jini lookup
        serviceobject = printclient.registrars[i].lookup
           (desiredtemplate);

        if (serviceobject instanceof PrintServiceInterface)
            {  
                System.out.println("Found a match!\n" +
                        "Calling print method: Hello World!\n");
                printer = (PrintServiceInterface) serviceobject;
                if (!printer.print("Hello World!\n"))
                    {  
                        System.out.println("PrintClient: Print Failed!\n");
                    }  
                else System.out.println("PrintClient: Print
Successful!\n");
            }  
        else System.out.println("No match Found. :-<");
                    }    //end for
    }  //end if
}  //end while
}  //end client

A Walk Through the PrintClient Code

Unlike our PrintServer class, we won't use a single class to handle all our responsibilities. Therefore, it will be the responsibility of our PrintClient class to handle the Discovery and Lookup process directly as well as have the main method. The first hint at how we are going to do this occurs as we see our PrintClient implementing the DiscoveryListener interface. Much as in AWT programming, by implementing this interface, we can use an instance of our PrintClient class to register itself as an event handler. The events we will handle are the two methods of the DiscoveryListener interface, Discovered and Discarded. Discovered is called when a JLUS is found and its ServiceRegistrar object is obtained. Discarded is called when a JLUS we had been aware of, becomes unavailable to us. Since we don't care too terribly much about the second situation in this example, we don't do anything in the Discarded method. On the other hand, the Discovered method is going to be the way our PrintClient will get its hands on the ServiceRegistrar object, so we will want to do something here, and we do.

We obtain the array of ServiceRegistrar objects from the method's only parameter and store it for later use. The next, and last, thing we do here is to set a flag. As you might guess, this will signal other mechanics to do the bulk of the work later.

In our PrintClients main method, we start off by creating placeholders to handle the other things we're going to need to get through the Discovery and Lookup process. Just like the service, we have to extend the security model in order
to run remote code in our local VM. Unlike the service, however, we will not just be obtaining the ServiceRegistrar object, but also the print service's proxy object.

Next, we actually go through and perform Jini Discovery. This is done by simply creating a LookupDiscovery object. Just creating it will send it off to find all the JLUSs it can. We don't have to tell it any more information. During the instantiation of this object, however, a new Jini concept called groups comes into play.

Basically, groups are a way to logically organize services together in a Djinn. Without going into great detail, a company may be a single large Djinn but may also decide to define different groups within that Djinn, like the "Engineering Services" group or the "Cafeteria Services" group. Groups are just a way for Jini Services to further differentiate and organize themselves within a Djinn. For our Discovery, we pass an empty string as the argument. This says that we want to look in the public group, which is the default group all services join.

Since we sent the LookupDiscovery object to go out and discover the available JLUSs, we must have a way for it to give us all the ServiceRegistrars it discovers. This is where our PrintClients DiscoveryListener interface comes in handy. We create an instance of our PrintClient and pass it to our LookupDiscovery object as the event handler for Discovered and Discarded events. Now whenever our LookupDiscovery object finds new JLUSs, it will notify us by calling our Discovered method.

Next, we define the service we want to look for. Here again, we create an array of Entry objects with only 1 element. We make this element a Name and initialize it to our "Jini'fied Morse Code Printer."

Finally we get to the driving while loop of our PrintClient class. Here we will loop forever. Each time we loop, we will check the JLUSFound flag kept by our PrintClient to see if there are any targets to perform Lookup on. While there aren't any, we'll loop forever. When we get one, we fall through the if statement and begin to process it.

Now, in a real Jini environment, there just may be multiple lookup services. That's why, when we ask the DiscoveryEvent object back in the Discovered method for a ServiceRegistrar, it actually hands us an array of them. We can query this array, just like any other in Java, and find out exactly how many registrars we found. The answer we get here will depend on how many JLUSs you started up when you went to run our little example, probably one. Our client is a little robust at least and will handle the case of N JLUSs.

We index through the array, just as you would any other. On each element (JLUS ServiceRegistrar object), we perform Jini Lookup. To do this, we pass the ServiceTemplate we made to identify our service by its name as the argument in the registrar's aptly named lookup method. The result will either be NULL, meaning no match, or the proxy object of the first service that matched the template.

We check to make sure that the proxy object implements our well-known PrintServiceInterface by using a bit of RTTI, and if it does, we send "Hello World!" out to the printer. If the proxy object doesn't implement that interface (who knows why you would have a service out there calling itself Jini'fied Morse code printer that didn't implement our interface, but you may), we say that we couldn't find a match.

After we have exhausted all the ServiceRegistrar possibilities, we fall out of the four loop and eventually out of the if where we check the JLUSfound flag. Since we reset that flag, we will loop forever until the user kills the process, or our LookupDiscovery Object finds a new JLUS.

**Running the Jini Server**

Because standard RMI stubs and skeletons are being utilized, this service requires a Web server running to distribute the stub file to anyone who needs it. Don't forget to set the RMI codebase parameter to point to the URL where the stubs are served from. If this is confusing, it may be helpful to go back and refresh your knowledge of Java RMI. In effect, the PrintService_Stub.class file is the PrintServices proxy object.

As an additional note, we will need a security policy file that allows the service enough privilege to be able to download and run code given to it by the JLUS.

**Required Files in Service Directory for running:**

- LightDriver.class—Light Driver for Linux System
- MorseTranslator.class—Actual Morse code translator
- MorseTable.dat—Table used by MorseTranslator
PrintService.class—The Jini PrintService
PrintServiceInterface.class—The well-known interface
PrintService_Skel.class—The server-side part of the RMI stubs
policy.all—The Java RMI security policy

Required Files to be served by the service's webserver:

PrintService_Stub.class—The client-side part of the RMI stubs

To run the server, you must have at least one JLUS running already on your local network, such as Reggie the JLUS that comes with the Jini SDK.

The following commands will start up the Web server to serve the Stub file and run the service.

In a UNIX environment:

echo Starting Server Codebase Webserver
java -jar /path_to_Jini/jini1_0/lib/tools.jar -dir
/path_to_service_codebase_directory/service-codebase -verbose
-port 8001 &

echo Starting Service
java -Djava.security.policy=policy.all
-Djava.rmi.server.codebase=http://192.168.1.4:8001/ PrintService

You would replace the text in italics with the appropriate path for your system.

In a Windows environment:

echo Starting Server Codebase Webserver
java-jar Drive:\path_to_Jini\jini1_0\lib\tools. jar -dir
\path_to_server_codebase_directory\service-codebase -verbose
-port 8001 &

echo Starting Service
java -Djava.security.policy=policy.all
-Djava.rmi.server.codebase=http://192.168.1.4:8001/ PrintService

You would replace the text in italics with the appropriate path for your system.

Assuming the appropriate classpath setup, the server should then run. As it is written, the server will execute and remain silent. It would be an easy task to include some code to inform the user that it is up and running. Because the serviceIDNotify method gets called when the JLUS first assigns an ID to the service, a statement could be placed here indicating to the user that a successful join had occurred.

Running the Jini Client

Running the client is a little simpler. Again, we assume that there is a JLUS already running on our local network. Here also, a policy file that allows the client enough privilege to download and execute code from both the service and the JLUS is needed.

Required Files in Service Directory for running:

PrintClient.class—Our PrintClient class
PrintServiceInterface.class—The well-known interface
policy.all—The Java RMI security policy
The following commands will run the client.

In a UNIX or Windows environment:

```
echo Starting the Client
java -Djava.security.policy=policy.all PrintClient
```

Again, this assumes that your classpaths are set up properly for your Java and Jini installations.

**Good References to Get You Started**

As mentioned earlier, this chapter is really only an introduction to Jini. I highly recommend purchasing *Core Jini* by W. Keith Edwards, as both an excellent introduction and a comprehensive text on this technology. In addition, several kind individuals have made tutorials and examples of their Jini efforts available online. Much of this is equally helpful in getting your own Jini projects up and running. Because the Web is ever changing, I don't include the URLs, but I do encourage people to use their favorite search engine to find and visit these resources:

- Noel Enete's Jini and Java Nuggets
- Jan Newmarch's Guide to JINI Technologies
- Bill Venner's Jini resources at Artima
- All the material available through Sun on Jini, from the FAQ to the specifications.

Concerning the actual Jini'fied Morse code printer, an actual Morse code printer was built at Binghamton University in 1999. The prototype system utilized a simple plastic box containing two LEDs wired to a parallel cable. This parallel cable was connected to a Red Hat Linux box running on an old 486 class PC. The PC acted as the "embedded smarts" of the Jini device, implementing the full Java 2.0 SDK and Jini 1.0 SDK. Though the prototype system was quite large, and thus not "embedded" by any standard, its use as a model was still valid. The operating system, Java and Jini Runtimes, Ethernet, and parallel ports all could have easily been implemented on one of several credit-card-sized PC systems commonly available from embedded systems manufacturers.

It is worth mentioning that Linux was the operating system of choice for this application for a couple of reasons. Linux is a freely available, full-fledged UNIX clone. Linux has been used in a variety of embedded applications. Also, because of the security restraints and cross platform features of Java, it wasn't possible to access the printer hardware directly as an address in memory, as you might in C or C++. However, in Linux, as in other UNIX systems, hardware is accessible as simple files in the `/dev` file structure. In this way, Java code could easily be written to twiddle the output bits on the parallel port (`/dev/lp1`) in exactly the same manner as writing to a file! This made the entire task of writing a "device driver" in Java trivial.

**Chapter 11. JMX/JMAPI: Java Management API**

- What Is Network Management?
- Modifying Clients for JMAPI
- Modifying Servers for JMAPI

Since the original publication of this book the Java Management API has been renamed Management Extensions and placed under the Java Community Process. In June 1999 at SuperComm '99 Sun Microsystems announced the availability of the Public Draft of the Java Management Extensions (JMX) specification. Sun and a number of leaders in the field of enterprise management—Powerware, IBM, Computer Associates, BullSoft, TIBCO, and Xylan—jointly developed the specification.

JMX is built on the JMAPI foundation and draws on Sun's experience with the Java Dynamic Management Toolkit (JDMT). The JDMT has been out and in the hands of developers for the last 2 years and has been proven to provide the tools for building distributed, Web-based modular and dynamic solutions for managing devices, applications, and service-driven networks.

The JMX Draft Specification can be downloaded from [http://java.sun.com/aboutJava/communityprocess/first/jsr003/index.html](http://java.sun.com/aboutJava/communityprocess/first/jsr003/index.html). What follows is the original book's discussion of the JMAPI, which, for the purposes of this book, should still be appropriate and informative.
The Java Management API as originally released was a heavy-duty implementation of the Solstice suite of network management tools that come bundled with Sun Microsystems' Solaris operating system. Even though JMAPI does not require Solaris, it can help you to bring the power of Solstice to your operating environment. Your Java objects can be ensured of some semblance of stability if you provide a means by which your applications can be monitored by a "neutral third party." If your applications go down, JMAPI can help you bring them back up. At the heart of networked communication is the need for the reliability that JMAPI can help provide.

In this chapter, we cover network management and show you how to introduce a management scheme for your clients and servers. Also, we touch briefly on how and when to manage your objects. The concept of network management is discussed in short, with emphasis on the needs of a management API and how those needs are met by JMAPI.

**What Is Network Management?**

At first glance, a book about Java networking does not appear to need a chapter on network administration. After all, system administrators are hired by most organizations to ensure that a network stays up and running. Most of the time, however, system administrators are presented with a horrendous number of different tools with which to do their job.

To make matters worse, these tools generally have no relationship to one another and have vastly different user interfaces. For the system administrator, this amounts to a significant amount of frustration. For the organization for which they work, this amounts to a significant amount of money spent on training.

What is needed is a simple set of tools that can be used to build a homogeneous environment for the administrator. If the tools have a common interface, then the system administrator needs to learn only the basics of one tool to understand the others. The Java Management API, or JMAPI, provides a robust environment in which you can create administrative tools, provide administrative functionality, and modify your regular Java objects so that they can be administered by the JMAPI.

As the Internet grows, and as programming the Internet becomes more and more accessible, the need for complex network management will be apparent. If you create your Java applications with management in mind, you can prepare for the eventual arrival of Internet system administrators.

**Network Management at a Glance**

A long time ago, the notion of a network did not exist. In fact, computers were connectionless entities that resided in a room and did not in any way talk to one another. Soon, the Local Area Network emerged and computers in the same physical location could be connected to one another. It enabled information to flow from computer to computer, and even for data to be centrally located on another computer. Then, these little networks began to merge with larger networks and, eventually, the Internet developed and connected them all together (see *Figure 11-1*).

*Figure 11-1. The growth of connectivity among small networks eventually gave rise to the Internet.*
Some day, our children will hear the tale of the birth of the network as they bounce on our knees, but today we are presented with a very adult problem: how to make sure each one of those computers stays up and running and how to fix them when they do break. This is the high-pressure world of network administration.

To complicate the matter further, network administrators often are asked to handle software concerns as well. To facilitate this, several protocols that hook into applications and determine and/or fix their health were developed. Once again, network administrators are called on to fix ailing applications and bring them back to a usable state.

As Java applications become more and more popular, and JavaStations and the Java operating system gain greater acceptance, a burden will be placed on network administrators to ensure the reliability of applications and the hardware on which those applications run. To assist with this matter, Sun's Solstice network administration group put forth the Java Management API. The JMAPI will be discussed in detail in the next section, but for now we concentrate on traditional network administration problems and how they relate to Java.

**Simple Network Management Protocol**

One of the protocols created by the Internet Engineering Task Force to assist with local and wide area network administration is the Simple Network Management Protocol. SNMP has several advantages over its competitors; chief among them is its ease of use. By setting up something called an SNMP trap, network administrators are able to identify crucial components, protect them, and give themselves a means by which to be notified when the component fails.

SNMP exchanges information between the manager application and the managed component through something called a Protocol Data Unit (PDU). A PDU contains information about a component and is sent over a network connection to the manager application. The application can read the PDU and determine the health of the component. PDUs usually contain information about an application's name, type, and current state. The SNMP trap we referred to is actually a form of a PDU.

SNMP is in wide use today. Chances are high that your network connection to the Internet uses SNMP in one form or another to maintain its integrity. SNMP manager applications are monitored by network administrators who can determine if and when a component fails and from there arrive at a solution to the failure fairly quickly.

It is important to understand that we refer to SNMP in "application" space. The truth is that SNMP can be incorporated within applications themselves. In so doing, a network administrator can pinpoint the exact causes of failures because he or she has a direct hook into the code that failed. In a moment, we will see how Java programmers can create similar applications by using JMAPI rather than SNMP.

**The Unique Management Problems of Java**

One of the biggest problems encountered with incorporating SNMP into Java is that SNMP is not Java. Java is a wonderful language, with great ease-of-use features. We want to be able to deploy large-scale Java applications both over the Internet and within our corporate intranets. In order to do so, and still have control over network administration, we must have a way to hook into Java code and obtain information about it.

In the next few sections, we will examine the JMAPI closely and learn where it can be used appropriately when deployed Java applications are created. As the language gains more acceptance, as Java hardware becomes more and more prevalent, and as applications that are written in Java exclusively are shipped, some form of Java management mechanism must be developed and used if our networks are to maintain a semblance of integrity.

**Network Administration Overview**

Network administration is often the underemphasized aspect of the Internet revolution. Without a coherent network administration strategy, all networked applications will fall apart, and the network backbone will break. It is because of the importance of this that we will undertake a discussion of its relevance to Java network programming.

**Modifying Clients for JMAPI**

The client code that is included as part of the Java Management API consists of a series of RMI clients that interact with managed object servers. These RMI objects enable you to communicate seamlessly with the object your client is designed to manage. Your client should be able to affect the performance and activity of the server, provided the managed object follows the JMAPI architecture and implements its core objects.
The JMAPI client architecture also consists of what can only be called a user interface bonanza. From pie charts to line graphs, lists to graphical lists, icons to animation, the JMAPI's Admin View Model (AVM) is nothing more than a layer on top of the Abstract Window Toolkit. In so doing, the AVM is, like the AWT, completely platform independent and AVM does not rely on any windowing system to function.

**AVM Base Classes**

The AVM base classes are, as we discussed, an extension of the AWT. They implement several components, including image buttons, scrolling windows and panels, toolbar, image canvases, dialog boxes, and things you can do while your application is busy. There are also several generic tables, HTML browsers, and chart objects for you to use as you see fit. We will not show you how to use each of these individually because they are used the same way the normal AWT classes are used.

**AVM Help Classes**

The AVM help classes provide a general-purpose help utility for application programmers. By using the AVM Help functions, your application's help documentation could be used just as easily by other, non-JMAPI, applications and vice versa. Why duplicate documentation efforts when the JMAPI can assist you in creating a uniform documentation structure? AVM help documentation is nothing more than HTML with a few JMAPI authoring tags sprinkled within it. The JMAPI tags are contained in comments within the HTML documentation, so the HTML documentation can be used elsewhere without giving away the fact that the same text is also used by the JMAPI.

The help classes consist of four modules. The first of these modules is the UI-based Table of Contents and Navigator. The TOC/Navigator allows you to survey your documentation and build a hierarchical list of the topics contained therein. It uses the authoring tags within the documentation set to determine the arrangement of the contents list.

A documentation generator also is included to assist you in creating indices, glossaries, and even table of contents files. The documentation generator (jmapidoc) acts on the HTML file, parses the authoring tags within it, and spits out a series of HTML files that can be used by the Help Navigator.

The third module is a series of help files built by the JMAPI documentation generator and referring to the JMAPI itself. This way you can pass on information about how JMAPI operates as part of the documentation for the ManagedObjects you create. A set of help templates that you can fill in yourself is included along with the standard JMAPI help files. They will help you get started with building documentation for your objects.

Last, a search engine is included with the AVM help utilities so that end users can find the information you have created for them quickly.

**Managed Object Interfaces**

Let's say we have a series of objects that model each individual employee in our large, monolithic corporation. Traditionally, the solution to poor employee morale is more management. Therefore, to improve our employee's morale, we will add a manager.

Our EmployeeManager is based on RMI, so we must include the RMI classes in our file. Furthermore, we must create a StatusObservable object to oversee the object. The StatusObservable object will link our ManagedObject to an event notification mechanism. If we so desire, we can set up a notification link within our client. If any other client fiddles with our employee, we would know about it instantly.

```java
public class EmployeeManager {
    public static void main(String args[]) {
        // our employee
        EmployeeInterface employee;

        // our observable class
        StatusObservable statusObserver = new StatusObservable();
    }
}
```
Once we have set up our EmployeeManager, we must go to the ManagedObjectFactory to get an EmployeeInterface object. The first argument to the newObj call is the name of the object, including any Java package containers that are associated with it. The second argument is the name of the object in the name space (in our case "EMPLOYEE").

```java
public class EmployeeManager {
    public static void main(String args[]) {
        // our employee
        EmployeeInterface employee;

        // our observable class
        StatusObservable statusObserver = new StatusObservable();

        // create the employee object
        try {
            employee = (EmployeeInterface) MOFactoryObj.newObj(
                "sunw.jmapi.EmployeeInterface",
                "EMPLOYEE");
        } catch (Exception exc) {
            System.out.println("Error in create: " + exc.toString());
        }
    }
}
```

Now we must set our employee's attributes. We will create a setTask method and a setName method here and implement them in a few moments. We will then go about adding the object to the management system (in other words, we'll place the employee in our org chart).

```java
public class EmployeeManager {
    public static void main(String args[]) {
        // our employee
        EmployeeInterface employee;
        // our observable class
        StatusObservable statusObserver = new StatusObservable();

        // create the employee object
        try {
            employee = (EmployeeInterface) MOFactoryObj.newObj(
                "sunw.jmapi.EmployeeInterface",
                "EMPLOYEE");
        } catch (Exception exc) {
            System.out.println("Error in create: " + exc.toString());
        }
    }
}
```
// assign a task
try {
    employee.setName("Heath");
    employee.setTask("Throw a touchdown");
} catch (RemoteException exc) {
    System.out.println("Error in setup: " +
        exc.toString());
}

// add this client to the list of objects listening
// in on this employee
try {
    employee.addObject(statusObserver);
} catch (Exception exc) {
    System.out.println("Error in add Object: " +
        exc.toString());
}
}

There are two operations analogous to the addObject method we just employed. We just as easily could have modified
the object and then notified any other observers of this change by invoking modifyObject on the employee. If we
wanted to remove the object from the management systaem, we could invoke deleteObject on the employee.
Remember that the clients can set and modify attributes, the servers can use those settings to do their business, and
the clients can notify other clients that a change has been made.

**Setting Up Notifications**

In order to create the notification mechanism in our client, we must implement the MOObserver and create an
ObserverProxy within the employee object. We must also implement the update function as required by the
MOObserver. The update function will be the callback function. As we have discussed in Chapter 6 on IDL and Chapter
5 on RMI, simply setting up a callback is not enough. We also must create a place for the server to invoke that
callback.

```java
public class EmployeeManager implements MOObserver {
    public static void main(String args[]) {
        // our employee
        EmployeeInterface employee;

        // our observable class
        StatusObservable statusObserver =
            new StatusObservable();

        // create the observerproxy
        ObserverProxyImpl observerProxy =
            new ObserverProxyImpl(this);

        // create the employee object
        try {
            employee = (EmployeeInterface)MOFactoryObj.newObj(
                "sunw.jmapi.EmployeeInterface",
                "EMPLOYEE");
        }
    }
```
catch(Exception exc)
{
    System.out.println("Error in create: " +
            exc.toString());
}

// add the observer proxy to the managed object and
// tell it to notify on modifications only
employee.addObserver(observerProxy,
        ManagedObject.OBSERVE_MODIFY);

// assign a task
try
{
    employee.setName ("Heath");
    employee.setTask ("Throw a touchdown");
} catch(RemoteException exc)
{
    System.out.println("Error in setup: " +
            exc.toString());
}

// add this client to the list of objects listening
// in on this employee
try
{
    employee.addObject(statusObserver);
} catch(Exception exc)
{
    System.out.println("Error in addObject: " +
            exc.toString());
}

public void update(
        ManagedObject mo,
        Observation observation)
{
    if (observation instanceof ModifyObservation)
    {
        . . . do our modification stuff here . . .
    }
}
}

When any other manager client invokes modifyObject on the ManagedObject, we get a notification in the Update
method. This enables us always to stay in synch with the managed object, even if there are many other clients
modifying it at the same time. Now we must set up our servers so that the clients we just created can manage them.

Modifying Servers for JMAPI

The Java Management API includes a set of objects to be used on the server side of a Java networked system. These
objects are known collectively as the Administration Runtime Module (ARM). The ARM is the focal point of all
management associated with Java applications. Once Java applications include the ARM, they are essentially
instantiated objects that can be readily administered.

The various components of the ARM are discussed in the next sections. Each component contributes a specialized
function to the overall goal of administering your clients fully. Your Java applications can be plugged into existing
system management protocols and software including SNMP, Solstice, and others using the ARM. Once your Java
applications can be administered, you can rest easy in the knowledge that your object system can handle network
situations beyond your control.
The Admin Runtime Module

The diagram in Figure 9-2 outlines the various components of the ARM that we will soon discuss. All of the components are interchangeable. For example, your application may need to use the Managed-Object routines, but you can omit the Notification module easily should you so desire. Furthermore, your managed applications remain fully scalable and the performance of your system should not be degraded. Figure 11-2 illustrates the modular design of JMAPI applications.

Figure 11-2. Modular design of JMAPI applications

When a JMAPI client is used to communicate with your server, it can inquire as to its status and overall health. This information may take the form of a notification, essentially a callback set up by the client using the JMAPI server, or the client may inquire of its own volition. This kind of communication and data exchange is the heart of networked computing and is required in order to administer the network with which you communicate.

ManagedObject Classes

The ManagedObject class implements a distributed management architecture. It enables multiple clients to obtain links to the same managed server and affect changes on them simultaneously. Its underlying communication mechanism is Java RMI (see Figure 11-3). Using remote method invocations, the ManagedObjects residing on the client side can talk to the RMI ManagedObject server running within the managed server.

Figure 11-3. RMI is the underlying communication mechanism of JMAPI.

In order for a ManagedObject to begin communication with the object it wishes to manage, the management server must be configured. This is done using the RMI paradigm of creating a public interface for the ManagedObject clients to talk to first. Here we are going to create an employee object and attempt to manage it:

```java
public interface EmployeeInterface extends ManagedObject {
    // set and get a task
```
public void setTask(String newTask);
public String getTask();

// set and get the employee's name
public void setName(String newName);
public String getName();
}

As you can see, this public interface inherits from ManagedObject and, therefore, gets all the RMI functionality it needs. In so doing, we need not mention RMI throughout the server explicitly. Indeed, the RMI network code is part of ManagedObject.

**Implementing the ManagedObject**

Once we have created the interface, we must create the implementation of the ManagedObject class. This adds the functionality to the interface so that we have much more than a skeleton. We will add a constructor and implement the functions prescribed in the EmployeeInterface object.

```java
public class EmployeeImpl extends ManagedObjectImpl implements EmployeeInterface
{
    public EmployeeImpl() throws RemoteException
    {
        super();
    }

    public void setTask(String newTask)
    {
    }

    public String getTask()
    {
    }

    public void setName(String newName)
    {
    }

    public String getName()
    {
    }
}
```

In addition, we must implement the performAction methods that map the creation of ManagedObjects using the ManagedObjectFactory of our implementation. Whenever this ManagedObject is created, deleted, or modified using the factory, one of the performAction methods will be called.

```java
public class EmployeeImpl extends ManagedObjectImpl implements EmployeeInterface
{
    public EmployeeImpl() throws RemoteException
    {
        super();
    }
```
public void setTask(String newTask)
{
}

public String getTask()
{
}

public void setName(String newName)
{
}

public String getName()
{
}

public void performAddActions(StatusObservable observable, CommitAndRollback commitObj)
{
}

public void performModifyActions(StatusObservable observable, CommitAndRollback commitObj)
{
}

public void performDeleteActions(StatusObservable observable, CommitAndRollback commitObj)
{
}

The perform methods accept an observable parameter. This facilitates notifications, which we will discuss shortly. Perform methods also take a CommitAnd-Rollback object to keep track of all the operations on the ManagedObject. If we encounter any kind of error during the Perform method, the CommitAndRollback object will allow the ManagedObject to backtrack and resume its previous unaltered state. This ensures that the perform methods are always atomic in nature, meaning that either the whole thing is complete or none of it is. In keeping with their implementation, the perform methods are intended for atomic operations (database access, file manipulation, or the like). We have not implemented anything in them here, but they are still required as part of the ManagedObject implementation.

Managed Attributes

The name and task attributes of our employee need not be created in the EmployeeImpl object itself. Instead, we can take advantage of the ManagedObject's attribute mechanism. We can enable the ManagedObject to handle persistence and network-related tasks associated with the attribute for us by storing our attributes within the ManagedObject's infrastructure.

public class EmployeeImpl extends ManagedObjectImpl
    implements EmployeeInterface
{
    public EmployeeImpl() throws RemoteException
    {
        super();
    }
}
public void setTask(String newTask) {
    setKnownAttributes("employee-task", newTask);
}

public String getTask() {
    return (String) getAttribute("employee-task");
}

public void setName(String newName) {
    setKnownAttributes("employee-name", newName);
}

public String getName() {
    return (String) getAttribute("employee-name");
}

public void performAddActions(StatusObservable observable, CommitAndRollback commitObj) {
}

public void performModifyActions(StatusObservable observable, CommitAndRollback commitObj) {
}

public void performDeleteActions(StatusObservable observable, CommitAndRollback commitObj) {
}

ManagedObjectFactory Classes

In the previous section, we saw how a ManagedObjectFactory enables a client to obtain a handle to the server with which it wishes to speak. The ManagedObjectFactory is no different from the BMW auto factory in Spartanburg, South Carolina. In Spartanburg, BMWs flow off the assembly line one by one. Here, our ManagedObjectFactory enables us to serve up ManagedObjects on demand, creating them on the fly, initializing them, and readying them for use. As long as our objects implement the ManagedObject interfaces, they can be created and obtained through a factory.

Notifications

We discussed how notifications are based on the Java Observer and Observable classes in the client section. Unlike in Chapter 5, "Java RMI: Remote Method Invocation," supporting notification callbacks in JMAPI involve virtually no effort on our server's behalf. The ManagedObject implementation takes care of tracking the individual subscribing clients and publishing information when necessary. By using attributes as we did earlier, the ManagedObject is able to intercept changes made to the server without the programmer having to supply it with any additional information.
Managed Data

JMAPI also enables you to set up your data structures and member variables so that they can be managed from clients as well. You can allow clients to check on the integrity of the data contained within your remote Java servers by registering your data with the ManagedObject class. The client then could execute a series of steps if it finds something to act on.

Server Agents

Leigh Steinberg is the sports world's greatest agent. He is able to obtain lucrative contracts and signing bonuses for his numerous clients. Similarly, software agents act on behalf of parent applications and obtain information or make invocations when triggered by certain events. The agents are remote objects, so they run in their own process, perhaps on their own machine, perhaps even on a remote network. In so doing, they do not affect the performance of the calling object.

Setting up an agent is similar to setting up an RMI object, but again you must handle much of the RMI overhead.

Chapter 12. What Are Directory Services?

• Some Background
• Introducing Java Naming Directory Interface
• Using the JNDI to Access LDAP-Based Data

Directory services are the services provided by special network databases that are used, much as paper phone books are (i.e., to map names to addresses, phone numbers, and services). The directory is really a distributed hierarchically arranged database made up of keys and associated attribute name/value pairs.

Whether or not you realize it, you use directory services whenever you use the Internet. The Domain Name System is a form of directory, although not as general purpose as the directories we will be discussing in this chapter. DNS provides a UDP-based naming lookup service, namely that of mapping IP addresses to host names and vice versa. Whenever we use our Web browser to retrieve a Web page, IP must use DNS to look up and retrieve the IP address of the host computer that has the page we wish to view. The same is true for FTP: whenever we want to retrieve a file from an FTP server, IP uses DNS to look up the IP address of the host running the FTP server.

Whenever we use our e-mail client, whether it be Netscape Messenger, Outlook, or any number of other e-mail clients (SMTP/POP3/MAPI), the e-mail address books provided are usually based on Directory Services and the Lightweight Directory Access Protocol (LDAP). If you are a Netscape Messenger user and check out directories under your browser preferences (for newer browser users Directory definition is right with the Address Book), you will be able to see the hostnames of a number of commercial directories that allow general access to the public. If you then select your personal address book, export (under the file drop-down menu) it to a file, and then use Notepad or Wordpad to view it, you will notice that each entry is keyed by a set of tags. You should see the dn: tag followed by a set of name/value attributes. Files in this form are in a format known as LDAP Data Interchange Format (LDIF). This file format is used for the batch process loading (and backing up) of Directory Servers. The LDIF file itself is usually populated with information from other data sources (possibly your company’s Human Resources Database, the Userid/password database for your Local Area Network, or data extracted from a relational database that keeps track of the hardware configuration of all the workstations in your site and is used via Directory Services by your local helpdesk).

Currently the two most popular uses of directory servers are for user authentication (by userid and password) for accessing limited access Web pages/sites and as e-mail address books for intranet-based address books (for a corporation) and by the large ISPs as address books for their users. In the near future, a big user of directory servers will be e-Commerce applications using the Enterprise Java Beans framework because directory servers are extremely fast. As a special-purpose database, they provide an ideal mechanism for giving persistence to Java objects.

Some Background

The concept and architecture of modern directory services comes to us from the ISO X.500 specification for directory services.

A directory is intended to be a hierarchically arranged database; arrangement of the database is left up to the designer of the Directory Information Tree (DIT). The DIT of a multinational corporation could be arranged a number
Every directory entry is uniquely identified by an ordered sequence of name/value attributes. The ordering of the attributes is such that it reflects the hierarchical relationship that exists between the attributes. Assume the following naming attributes:
If we use the concatenation of these attributes and their values to identify an entry in the directory, we have defined the entry's distinguished name. Distinguished names are unique, much like the primary key in a relational database table. My distinguished name using the preceding schema would be

\{C=US, O=SUNY, OU=Binghamton, CN=Dick Steflik\}

Figure 12-1 shows this pictorially.

Each node in the tree structure has a distinguished name made up of the list of attributes up to and including that node. The distinguished name for Binghamton University would be

\{( dn: ou=binghamton, o=suny, c=us\}

X.500 defines the directory service as an object (see Figure 12-2), accessed through a set of service ports. Each port is intended to provide access to a specific set of services. Three of the primary service ports defined early in the X.500 development are:

1. **Read Port** provides the ability to read information from a directory.
2. **Search Port** provides the ability to search and list directory information.
3. **Modify Port** provides the ability to add, modify, and delete directory entries.

To support these service ports, the DAP has a very comprehensive set of protocol-based operations that address all the facilities needed to create and maintain a large distributed directory.

For applications that are directory-oriented but of a scale smaller than what X.500 and DAP were designed for, applications like address books for Web browsers, authentication for Web pages, a lighterweight version of the DAP has been developed. This slimmed down version of DAP is named Lightweight Directory Access Protocol or LDAP for short. LDAP is described in RFC 2251 as an access mechanism to X.500 directories. It is a language-independent description of the protocol operations required to interact with an X.500 directory.

**Introducing Java Naming Directory Interface**

The architecture of Java Naming and Directory Interface (JNDI) is a Java-specific architecture for accessing a number of directory-based data repositories including LDAP. The Java Interface to LDAP is only one of a number of services provided through the JNDI architectural model shown in Figure 12-3.

**Figure 12-3. JNDI architecture.**
If we re-examine the architectural picture of JDBC in Chapter 4, we can see some very real similarities. In Figure 12-3, if we replace "JNDI" with "JDBC," "Service Provider Interface" with "Driver Manager," and "CORBA, NDS, NIS, LDAP, DNS, RMI" with words like "DB2, Oracle, Access, Sybase," we essentially have the same picture. The architecture is essentially the same after all; directories are really just special-purpose databases, and for each database (datasource) there is a driver or service provider.

The main difference between directories and relational databases is that the directory information model is hierarchical, whereas, the model for relational databases is a set of tables. Relational databases are much more general purpose than directories; because directories are special-purpose, their data model can be tailored to their special-purpose uses. This can make them extremely fast for the types of queries done against them, much faster than the equivalent query using a relational database.

Using the JNDI to Access LDAP-Based Data

The Netscape Directory server comes with a sample LDIF file for the Airius Corporation that can be imported to set a reasonably sized and typically set up database. We'll use this directory to demonstrate the major LDAP features.

Setting up the Airius Directory

To start this exercise, go to the Netscape download site for server software and download a trial copy of the Directory Server. This will get you a 30–60 day copy of the world's best Directory Server (yes, I am a little biased). This won't run on W95/W98 so make sure to download a copy for the appropriate platform you want to run the service on.

Using the Netscape Admin Interface, turn off the instance of the Directory Server that you wish to install Airius on. (In Netscape Suite Spot there is a separate Admin server through which you do all Suite Spot server administration.) Click on the button with the name of the instance you want to administer. When the page for the Netscape Directory Server administration is displayed, click the Database Management button and then click on the Import choice in the left-hand frame. On the Import panel, select the radio button for Airius.ldif and then click the OK button. Once imported, remember to turn the Directory Server back on before exiting the Admin Server.

To test your installation, enter the directory setup screen for your browser (and add the server by, assign it a name, enter the IP Address/Hostname or Localhost (if you are using it locally), enter port 389 as the LDAP port, and enter "o=airius.com" as the search root. Save this and try to query *; you should get an address book filled with the people of the Airius Corp.

The Airius Schema

The following is the first part of the Airius LDIF file. If we examine it a little bit, we can determine the schema of the Airius directory and will start to see the power of the LDIF import/export file format. We will also see that some of the information in the LDIF file can be added/updated through the Administration Interface to the Directory Server and that some of the data is best put in via the LDIF file, even though some of the data is directly put in by the people in the directory.

NOTE
When examining the LDIF file, keep in the back of your mind the fact that lines that are indented by a single space are continuation lines for the preceding line.

```
dn: o=airius.com
objectclass: top
objectclass: organization
o: airius.com
aci: (target ="ldap:///o=airius.com")
    (targetattr !="userPassword")
    (version 3.0; acl "Anonymous read-search access"; allow
    (read, search, compare)(userdn = "ldap:///anyone");)
aci: (target="ldap:///o=airius.com")
    (targetattr = "*")
    (version 3.0; acl "allow all Admin group"; allow(all)
groupdn = "ldap://cn=Directory Administrators, ou=Groups, o=airius.com")
```

The line `dn: o=Arius.com` identifies this as the distinguished name for the root of the directory tree and is also a member of the "top" and "organization" object classes; the only information stored in this node is "o: airius.com," which identifies the organization as airius.com.

```
dn: o=airius.com
objectclass: top
objectclass: organizationalUnit
ou: Groups
dn: cn=Directory Administrators, ou=Groups, o=airius.com
cn: Directory Administrators
objectclass: top
objectclass: groupofuniqueNames
ou: Groups
uniquemember: uid=kvaughan, ou=People, o=airius.com
uniquemember: uid=rdaugherty, ou=People, o=airius.com
uniquemember: uid=hmillner, ou=People, o=airius.com
```

The next group of lines is really a single line (notice the indentation) that identifies the aci (access control information) for the current node (root)

```
aci: (target ="ldap:///o=airius.com")
    (targetattr !="userPassword")
    (version 3.0; acl "Anonymous read-search access"; allow
    (read, search, compare)(userdn = "ldap:///anyone");)
```

Letting our imagination run a little bit wild, we can surmise that anyone in the directory has authority to read, search, and compare the userPassword. The second aci:

```
aci: (target="ldap:///o=airius.com")
    (targetattr = "*")
    (version 3.0; acl "allow all Admin group"; allow(all)
groupdn = "ldap://cn=Directory Administrators, ou=Groups, o=airius.com")
```
authorizes anyone in the group with the common name "Directory Administrators" full authority for the tree rooted at "o=airius.com" (target attribute). The next distinguished name:

dn: ou=Groups, o=airius.com
objectclass: top
objectclass: organizationalunit
ou: Groups

identifies at the next level in the tree an organizational unit called Groups. The addition of the next dn: ou=Directory Administrators structures our tree as shown in Figure 12-4. This dn: has additional information in it in the form of the "uniquemembers," which stores each dn: of the unique members as part of the data for that node.

Examining the node

dn: cn=Directory Administrators, ou=Groups, o=airius.com
cn: Directory Administrators
objectclass: top
objectclass: groupofuniquenames
ou: Groups
uniquemember: uid=kvaughan, ou=People, o=airius.com
uniquemember: uid=rdougherty, ou=People, o=airius.com
uniquemember: uid=hmillar, ou=People, o=airius.com

we notice that the uniquemembers tag identifies a dn: that is in another branch of the tree. The unique members are in the ou=People branch of the tree—which implies that the tree actually looks like Figure 12-5.

Figure 12-4. Airius.com.

Figure 12-5. Airius.com.
Notice that to reference information in another branch of the tree all that needs to be done is to identify its dn:

If we examine more of the LDIF, we quickly come to the realization that the majority of the file is taken up with the definitions of the individual people in the company. Let's look more closely at a single entry because it is the meat of the schema and identifies the attributes that we can query against:

dn: uid=kvaughan, ou=People, o=airius.com
cn: KirstenVaughan
sn: Vaughan
givenname: Kirsten
objectclass: top
objectclass: person
objectclass: organizationalPerson
objectclass: inetOrgPerson
ou: Human Resources
ou: People
l: Sunnyvale
uid: kvaughan
mail: kvaughan@airius.com
telephonenum: +1 408 555 5625
facsimiletelephonenum: +1 408 555 3372
roomnumber: 2871
userpassword: bribery

Most of the attribute entries are self-explanatory. Now that we understand the schema, let's get on to the business of using the JNDI to access information in our directory.

Connecting

Recall from JDBC that, before we can query, add to, update, or delete anything from a database we must make a connection to it. To do this we need to create a reference to an object that implements the DirContext interface. This is usually done by creating an InitialDirContext object and assigning it to a DirContext variable. To make the connection, we need to pass some environmental information to the InitDirContext object; this is done by loading a Hashtable with a minimum of two pieces of information:

1. The fully qualified name of the service provider to be used
2. The URL (including port number) of the directory server we want to access.

We do this using predefined keys set up in the Context interface.

```java
// create a hash table for passing environment info
Hashtable environment = new Hashtable();
```
Searching

This gets us a connection to the directory we wish to use, but to do a search we must supply some additional information. A useful search would be to search the directory for uid=kvaughan and display her attribute information.

When searching, we must set up a SearchControls object to tell the search engine the scope of our search. The SearchControls class has three scopes, identified as constants, that we can use depending on what it is we want to search for:

1. OBJECT_SCOPE—Limits the search to the names object.
2. ONE_LEVEL_SCOPE—Limits the search to all of the objects at the same level in the named context.
3. SUBTREE_SCOPE—Limits the search to the named subtree.

To do the actual search, we invoke the Search method on the DirContext object we created a little while ago. The Search method has a number of overloads; make sure that you read the API carefully, or you may not get the results you are after. For our example, we will need to identify the base of our search, a search filter (similar in purpose to the "where" clause of an SQL select statement), and any constraints we set (SUBTREE_SCOPE). The results from a directory search come back in a data structure called a NamingEnumeration. NamingEnumeration is a JNDI-specific extension of the Enumeration class that allows exceptions to be thrown during enumeration (this implies that it needs to be in a try/catch statement).

```
public static String BASE = "O=airius.com";
public static String FILTER = "uid=kvaughan";
NamingEnumeration result = context.search(BASE, FILTER, scope);
```

If we have done everything right to this point, the search results are waiting for us in the NamingEnumeration. Each entry in NamingEnumeration is a SearchResult object; remember that NamingEnumeration holds objects and that as we take objects out of it we must cast them to the appropriate type.

```
SearchResult srchresult = (SearchResult) result.next();
```

At this point, we can retrieve the distinguished name from the SearchResult object by using its getName method;

```
String dn = srchresult.getName();
```

This will give us the distinguished name relative to where we rooted our search. To get the entire distinguished name, we must concatenate the variable we used to base our search on
String temp = "dn= " + dn + BASE;

The attributes are still in the SearchResults object and can be retrieved into an Attributes object using the SearchResults getAttributes method

Attributes attrs = srchresults.getAttributes();

Recall from our brief introduction to LDIF that any attribute may have multiple values (e.g., a person may have multiple e-mail addresses). What we need to do now is iterate through the returned attributes and, for each attribute found, iterate through the list of returned values for that attribute.

NamingEnumeration ne = attrs.getAll();
While (ne.hasMoreElements)
{
    Attribute attr = (Attribute) ne.next();
    System.out.println(attr.getID());
    Enumeration values = attr.getAll();
    while (values.hasMoreElements())
    {
        System.out.println("    " + values.nextElement());
    }
}

The whole example follows:

import java.util.Hashtable;
import java.util.Enumeration;
import javax.naming.*;
import javax.naming.directory.*;

public class DirectorySearch
{
    public static String BASE = "o=airius.com";
    public static String FILTER = "uid=kvaughan";

    public static void main (String args[])
    {
        try
        {
            // create a hash table for passing environment info
            Hashtable environment = new Hashtable();
            // identify the service provider
            environment.put(Context.INITIAL_CONTEXT_FACTORY,
            "com.sun.jndi.ldap.LdapCtxFactory");
            // identify the directory to be accessed
            environment.put(Context.PROVIDER_URL,
            "ldap://mydirectory.com:389");
            // get a reference to the directory context
            DirContext context = new InitialDirContext(environment);
            // set the search scope
            SearchControls scope = new SearchControls();
            scope.setSearchScope(SearchControls.SUBTREE_SCOPE);
            NamingEnumeration result = context.search(BASE, FILTER,
            scope);
            SearchResult srchresult = (SearchResult) result.next();
            String dn = srchresult.getName();
            String temp = "dn= " + dn + BASE;
            Attributes attrs = srchresult.getAttributes();
            NamingEnumeration ne = attrs.getAll();
        }
while (ne.hasMoreElements())
    {
        Attribute attr = (Attribute) ne.next();
        String attrname = attr.getID() + " : ";
        Enumeration values = attr.getAll();
        while ( values.hasMoreElements() )
            System.out.println(attrname + values.nextElement());
    }
} catch (Exception e) {
    System.out.println("Exception: " + e.toString());
}
Modifying Information Already in the Directory

To modify attributes of an existing entry, we first set up our environment and get a reference to a directory context as we have in the previous examples. After we have the reference, we create a ModificationItem array to hold the modifications we wish performed on the item. Finally, we use the modifyAttributes method of the DirContext object to update the data.

```java
ModificationItem[] updates = new ModificationItem[3];
Attribute update0 = new BasicAttribute("roomnumber", "1234");
Attribute update1 = new BasicAttribute("l", "Chicago");
Updates[0] = new ModificationItem(DirContext.REPLACE_ATTRIBUTE,update0);
Updates[1] = new ModificationItem(DirContext.REPLACE_ATTRIBUTE,update1);
context.modifyAttributes(dn,updates);
```

The DirContext interface also provides the tagged constants `ADD_ATTRIBUTE` and `DELETE_ATTRIBUTE` for effecting the adding and deleting of attribute information.

Removing Entries from the Directory

To delete an item from the directory, we start out as we have previously, by obtaining a reference to a directory context; then all we do is use the destroySubcontext method of the DirContext interface.

```java
context.destroySubcontext(dn);
```

Authentication

One additional thing we must remember for any of the directory modification operations is that, according to our directory ACI (refer to the section on LDIF), only people belonging to the Directory Administrator group had all authority and would be allowed to write into the directory. To do this, the application must authenticate with the uid and password of one of the Directory Administrators. This is done by adding three additional tagged values to the context environment Hashtable.

```java
public static String ADMIN = "uid=kvaughan, ou=People,
  o=airius.com";
public static String ADMIN_PW = "bribery";
..
..
env.put(Context.SECURITY_AUTHENTICATION,"simple");
env.put(Context.SECURITY_PRINCIPAL,ADMIN);
env.put(Context.SECURITY_CREDENTIALS,ADMIN_PW);
```

Chapter 13. Java and Security

- Safety in Java
- The Java Security Model
- Java Class Security
- Encryption
- Authentication
We have all heard that Java is a "secure" programming language. What exactly does that mean? In this chapter, we discuss the unique features of Java that make it the ideal choice for distributed network programming. Furthermore, we will discuss the nuances of the applet host security model, as well as how security is handled from within your Java applications.

We will also touch very briefly on Internet security and some of the alternatives you may want to explore in your own networked applications to make them safe for cross-network transmission. We begin our examination with the topic of cryptography. The primary goal of cryptography is to provide data privacy, but, as we will see, cryptography can be used to provide other essential security principles including nonrepudiation, data integrity, and access-controlled authentication. We will then look at the issues surrounding authentication, a security process that attempts to identify a participant (user, server, and applet) transaction.

**Safety in Java**

When we refer to Java as a safe language, we are referring to the fact that you cannot "shoot yourself in the foot." There are no memory leaks, out of control threads, or chance of ending up in the dark spiral of C++ debugging. Make no mistake—Java is a powerful language, and you will always end up with the possibility of sitting in an infinite loop. You can still freeze your Java code with thread deadlocks, and you can certainly end up accessing parts of an array that aren't really there. In short, Java is safe, but it isn't idiot-proof. The fact remains that, in order to screw up your Java programs, you still have to make a major effort.

Most Java programmers are pleased that Java has no pointers to memory locations. This makes program debugging much easier, and it also makes security verification possible. It cannot be verified at compile time that a pointer will do no harm. It can be loaded at runtime with a naughty address to poke a hole in the password file or branch to some code that sprinkles at-signs all over a disk. Without pointers, Java ensures that all mischief is done within the downloaded applet running inside a Java Virtual Machine. Moreover, memory is not allocated until runtime, and this prevents hackers from studying source code to take advantage of memory layout because it is not known at compile-time. Attempts to write beyond the end of an array, for example, raise an ArrayIndexOutOfBoundsException. Had the C language had this feature (array bounds checking), the infamous Morris Internet worm would not have been able to trick the sendmail daemon (running with root privileges) into giving root access to the worm.

Garbage collection, exceptions, and thread controls are part of Java no matter how you try to use it. But, security and safety are two entirely different things. Safety refers to protecting ourselves from our own misadventures. Security refers to protecting ourselves from other people's devices. Because Java objects are loaded dynamically, Java ensures that the objects are "trusted." Java's class security mechanism makes sure that your applications are using the proper objects and not an object that someone has slipped into the download stream to wreak havoc on your machine.

**The Java Security Model**

The Java security model has been a constantly evolving part of Java. In the JDK 1.0 model, the "sandbox" concept was introduced. In the sandbox model, all local code (JDK-provided code) was run as part of the Java Virtual Machine, and all code retrieved from remote servers (applets) was run in a "sandbox" area of the JVM that provided only a limited set of services. The reason for doing this was based on the fact that any remotely retrieved code could be hostile. To protect the local machine the sandbox provided only minimal access to the machine resources (Figure 13-1).

*Figure 13-1. JDK 1.0 sandbox model.*
The JDK 1.1 added to the JDL 1.0 security model the concept of "trusted applets" that could run with the same privileges with respect to the local hosts system resources as local code. This was done through the advent of the Java Archive file format and the inclusion of a correctly signed digital signature in the JAR file. Unsigned applets in JDK 1.1 still run in the sandbox (Figure 13-2).

The JDK 1.2 evolves the security model by changing the goals to make it:

1. Easy to use fine-grained access control
2. Easy to configure security policy
3. Easy to extend the access control structure
4. Easy to extend security checks to Java applications as well as applets (Figure 13-3).

Figure 13-2. JDK 1.1 security model.

Figure 13-3. JDK 1.2 security model.
Easy to Use Fine-Grained Access Control

Fine-grained security has always been a part of Java; the main problem was that the JDK 1.0 and 1.1 models made it extremely hard to use. To get the degree of control required, subclassing and customizing of the SecurityManager and ClassLoader classes is required (not a task for the uninitiated or the faint of heart). This required quite a bit of programming and an in-depth knowledge of computer and Internet security.

Easy to Configure Security Policy

Because of the amount of code required to configure security policy with the earlier JDKs, it would be more user friendly if the software developers and users could easily configure the security policy via an external policy file built with either a text editor or a GUI tool.

Easy to Extend Access Control Structure

To extend the access control structure in JDK 1.1 required adding additional "check" methods to the SecurityManager class. The new model does not require the addition of new "check" methods to the SecurityManager; the new architecture is based on permissions in the policy file. Each permission defines access to a system resource.

Easy to Extend Security Checks to Applications

In an effort to simplify things and have all code treated equally, the JDK 1.1 concept of "trusted" code was dumped in favor of a model where all code (local or remote) is treated equally, including JDK 1.1 trusted applets. It is for this reason that some JDK 1.1 applications and trusted applets will fail with security exceptions when run under the JDK 1.2 virtual machine.

Java Class Security

Java's security model is made up of three major pieces:

- The Bytecode Verifier
- The Class Loader
- The Security Manager

The Bytecode Verifier

The designers of Java knew that applets could be downloaded over unsecured networks, so they included a bytecode verifier in the Java Virtual Machine's interpreter. It checks to make sure that memory addresses are not forged to access objects outside of the virtual machine, that applet objects are accessed according to their scope (public, private, and protected), and that strict runtime type enforcement is done both for object types and parameters passed with method invocations. The bytecode verifier does these checks after the bytecodes are downloaded but before they are executed. This means that only verified code is run on your machine; verified code runs faster because it does not need to perform these security checks during execution.

The Class Loader

Each imported class executes within its own name space. There is a single name space for built-in classes loaded from the local file system. Built-in classes can be trusted, and the class loader searches the local name space first. This prevents a downloaded class from being substituted for a built-in class. Also, the name space of the same server is searched before the class loader searches other name spaces. This prevents one server from spoofing a class from a different server. Note that this search order ensures that a built-in class will find another built-in class before it searches an imported name space. So, when classes are downloaded, the client's built-in classes are used because they are trusted (See Figure 13-4).

Figure 13-4. Downloaded Java objects use the local built-in classes rather than their own.
New to Java in the JDK 1.2 is the ability to define a security policy that can be defined for each application separately from the Java code in a policy file. The policy defined in this external file is enforced at runtime by the Java security manager class. Java classes that have the possibility of doing things that might violate the security policy have been rewritten to include checks of the defined policy so as to verify that the application writer really wants to allow certain operations.

Java 1.2 Security Policies

New to Java with the release of Java 1.2 is a methodology that provides a much finer-grained approach to the security of important system resources like the file system, sockets access, system properties, runtime facilities, and security facilities themselves. This is done by establishing security policies; when an application/applet/servlet is loaded, it is assigned a set of permissions that specify the level of access (read, write, connect,...) that the code has to specific resources. If code isn't specifically given permission to access something, it won't be able to. These sets of permissions are specified in an external text file called a policy file. Policy files can be created with a text editor or by using the policy tool that comes with the JDK.

For the sample code in this book, a policy file called "policy.all" is provided on the CD. This file will grant all permissions to everything (which is good for the purposes of this book but bad from the standpoint of production code deployment; code placed into a production environment should define only the permissions that it needs to run).

Policy Files

Policy files are made up of a set of "grant" statements that have the general form of:

Grant [SignedBy "signer names"] [, CodeBase "URL"]
{
    permission "permission_class_name" ["target name"]
    [, "action"] [, SignedBy, "signer names"];
    permission. . . .
}

where

- SignedBy—Indicates that this is signed code (as in a signed JAR file) and that signatures should be checked. This is used to verify that downloaded code is from a trusted source. This is an optional attribute; if it is absent, signature checking is skipped.
- CodeBase—A URL (usually either http:// or file://) of either a file or a directory to the grant applies.
- permission—The class that enforces the policy; the most commonly used are:
  - java.io.FilePermission—access to files
  - java.io.SocketPermission—access to sockets
  - java.lang.RuntimePermission—access to threads and system resources
- java.util.PropertyPermission—access to properties
  - target—A path to the resource. This is optional and, if absent, refers to the current directory.
  - action—Operations allowed (read, write, execute, delete).
  - SignedBy—Signers of the permission classes; if signers can't be verified, the permission is ignored.

There are, by default, two policy files that establish the permissions that an application runs under—a system-wide policy file and an optional user (application) specific policy file. The system-wide policy file is kept in /java.home/lib/security/java.policy (java.home is a system property that contains the name of the directory that the JDK is installed in).

The default policy java.policy follows. It grants all permissions to standard extensions, allows anyone to listen in on ports above 1024, and allows any code to read standard system properties that aren't considered sensitive.

```java
grant codeBase "file:${java.home}/lib/ext/*" {
    permission java.security.AllPermission;
};

// default permissions granted to all domains
grant {
    // Allows any thread to stop itself using the
    // java.lang.Thread.stop() method that takes no argument.
    // Note that this permission is granted by default only to remain
    // backwards compatible.
    // It is strongly recommended that you either remove this
    // permission from this policy file or further restrict it to code
    // sources that you specify, because Thread.stop() is potentially
    permission java.lang.RuntimePermission "stopThread";
    // allows anyone to listen on un-privileged ports
    permission java.net.SocketPermission "localhost:1024-", "listen";
    // "standard" properties that can be read by anyone
    permission java.util.PropertyPermission "java.version", "read";
    permission java.util.PropertyPermission "java.vendor", "read";
    permission java.util.PropertyPermission "java.vendor.url", "read";
    permission java.util.PropertyPermission "java.class.version", "read";
    permission java.util.PropertyPermission "os.name", "read";
    permission java.util.PropertyPermission "os.version", "read";
    permission java.util.PropertyPermission "os.arch", "read";
    permission java.util.PropertyPermission "file.separator", "read";
    permission java.util.PropertyPermission "path.separator", "read";
    permission java.util.PropertyPermission "line.separator", "read";
    permission java.util.PropertyPermission "java.specification.version", "read";
    permission java.util.PropertyPermission "java.specification.vendor", "read";
    permission java.util.PropertyPermission "java.specification.name", "read";
    permission java.util.PropertyPermission "java.vm.specification.version", "read";
    permission java.util.PropertyPermission "java.vm.specification.vendor", "read";
    permission java.util.PropertyPermission "java.vm.specification.name", "read";
    permission java.util.PropertyPermission "java.vm.version", "read";
    permission java.util.PropertyPermission "java.vm.vendor", "read";
    permission java.util.PropertyPermission "java.vm.name", "read";
};
```

User- or application-specific policy files are kept by default in user.home/.java.policy (user.home is the system property that specifies the user's home directory).

Your overall security policy is created at runtime by first setting up permissions in the java.policy file and then setting the permissions found in the user policy file. To set up the system policy to your own policy just set the java.security.policy property to the URL of the policy file to be used. The URL can be specified as:

1. A fully qualified path to the file (including the file name).
3. java -Djava.security.policy=c:\advjavacd\rmi\stats1\policy.all
4.
5. rmi.Stats1.StatsServerImpl
6.

7. Any regular URL.
8.
9. java -Djava.security.policy=http://policy.allStatsServerImpl
10.

11. The name of a file in the current directory.
12.
13. java -Djava.security.policy=policy.all rmi.Stats1.StatsServerImpl
14.

The *policy.all* file we have been referring to follows:

```java
// this policy file should only be used for testing and not deployed
grant
{
  permissionjava.security.AllPermission;
};
```

**Security Tools**

The JDK comes with several tools to help you manage the security of code that you write and wish to deploy:

1. policytool—A Java application that comes with the JDK and that provides you with a GUI tool for creating and maintaining policy files.
2. keytool—Used to create digital signatures and key pairs and to manage the keystore database.
3. jarsigner—Allows the attaching of a digital signature to a JAR file.

For detailed instructions on these tools, refer to the JDK documentation and the security path of the Java Tutorial at [http://java.sun.com/docs/books/tutorial/security1.2/index.html](http://java.sun.com/docs/books/tutorial/security1.2/index.html).

**Security Problems and Java Security Testing**

Finally, the Java language has been thoroughly field-tested by high school and university students, college dropouts, and professional hackers lurking in the dark alleys of the World Wide Web. Each and every one of their creative minds was confident it could find a flaw in such a seemingly wide-open door to any system in the world! The most publicized security breaches happened early in Java's distribution, and all have been corrected in the current releases. It has been very quiet ever since. The flaws that were uncovered were implementation errors, not design problems. One group was able to insert its own class loader instead of the one loaded from a secure local file system. Clearly all bets are off if an untrusted class loader that doesn't enforce the class search order we described earlier is used. Another implementation bug was exploited by using a bogus Domain Name Server in cahoots with an evil applet. Java 1.0.2 uses IP addresses instead of hostnames to enforce the network access security levels described earlier.

Details about these early security flaws and their corrections can be found at [http://java.sun.com/sfaq](http://java.sun.com/sfaq).

**Encryption**

In this section, we describe some of the techniques commonly used to provide privacy during data exchanges between two parties. Data traveling through the Internet can be captured (and possibly modified) by a third party. Certainly, you do not want your credit card number to be revealed to a third party and you probably also want the merchandise
A very simple algorithm used to scramble "sensitive" jokes on the Internet is called "rot13" because it rotates each character by 13 positions in the alphabet. That is, "a" is mapped to "n," "b" is mapped to "o," and so on. This algorithm also decrypts a message that was scrambled by it. This is adequate for its purpose: to protect people from reading a joke that they might feel is offensive. This is an example of symmetric key encryption, where both sides use the same key (13) to encrypt and decrypt a scrambled message (see Figure 13-5).

**Figure 13-5. Symmetric key encryption decodes messages with a key on both the sending and receiving ends.**

In its most commonly used mode, data encryption standard (DES) uses a 56-bit key to scramble message blocks of 64 bits; in this form DES encrypts large amounts of data relatively fast. DES is currently one of the encryption algorithms used by Secure Sockets Layer (SSL). Recent research has shown that 56-bit DES is becoming insufficient for providing robust encryption for security-sensitive applications. Many companies now use "triple DES," which encrypts each block of data three times with three different keys.

One problem with symmetric key algorithms such as DES is key distribution (i.e., how do I share the private key securely among the participants?).

Public key, or asymmetric cryptography, uses a pair of mathematically related keys for each user. Everyone can know a user's public key, but the private key must be kept secret. To send data to another user, the sender encrypts the data using the recipient's public key and sends the encrypted message to the recipient. The recipient decrypts the message using his or her private key. Because only the recipient knows the private key, data privacy is ensured. Asymmetric algorithms are inherently slower than their symmetric counterparts. The key distribution problem of symmetric algorithms is overcome through the use of the public/private key pairs because the public key can be widely distributed without fear of compromise. There is still one problem with key management in public key encryption schemes. Namely, how do I know that the key I am using for Joe is really Joe's public key? It could be possible for a network interloper to substitute his or her public key for Joe's public key. A variety of trust models have risen to combat this problem. For corporations, the most prevalent model is the hierarchical trust model, which relies on the use of digital certificates and certificate authorities to validate users' public keys.

Real-world cryptographic implementations utilize a combination of public and private key encryption to provide not only data privacy but also nonrepudiation (via digital signatures), access control, and authentication. These solutions use the strengths of both public key (key distribution) and private key cryptography (speed); an example follows.

John creates a document and wants to send it to Mary. John first encrypts the document using a symmetric algorithm (like DES) and a randomly selected key. The randomly selected key is then encrypted using an asymmetric algorithm (like RSA) and Mary's public key. A message digest function (one-way mathematical function (like MD5) is performed on the original document producing a fixed-length message digest. This message digest is encrypted using an asymmetric encryption algorithm using John's public key. These three elements are then sent to Mary over some unsecured communications link. This is shown in Figure 13-6.

**Figure 13-6. A combination of symmetric and asymmetric encryption.**
The process of decrypting and verifying the encrypted document is shown in Figure 13-7 and goes something like this: Mary uses her private key to retrieve the random symmetric key used to encrypt the document. Because Mary is the only one who knows her private key, she is the only one who can open the "digital envelope," thus ensuring data privacy. The retrieved symmetric key is used to decrypt the document. Using the same message digest function as John, Mary produces a message digest for comparison to the one sent by John. Mary now uses John's public key and the asymmetric encryption (RSA) to retrieve the message digest sent with the document. By using John's public key to retrieve the message digest, Mary has also verified that the message was sent by John (i.e., retrieved his digital signature) because only John's private key could have been used to encrypt the message digest. The message digest sent with the document is compared with the one computed by Mary. This comparison ensures the data integrity. If the digests match, the document was unaltered during transmission.

**Figure 13-7. Decryption of example.**

**Java Cryptography Extension (JCE)**

The JCE provides a set of APIs that allow you to encrypt, decrypt, and password-protect information.

**Authentication**

In many applications, it is important to authenticate the identity of a client making a request for a service. Examples include banking, financial, real estate, medical records, and ISP (Internet Service Provider) applications. An ISP, for example, wants to ensure that Internet access is being provided to a paying customer and not the customer's
housekeeper. The online stock trading application wants to make sure that it is the portfolio owner who is making trades.

The usual way to do this is to require an account number or customer name and a password. This is adequate for workstations and time-sharing systems and client/server sessions such as calling Charles Schwab to manage your stock portfolio. In a distributed system, many different servers provide services. Instead of a single authentication to a single server or application, that server must authenticate each service request sent over the network.

One obvious requirement of such an authentication system is that it be transparent to the user. The user does not want to type in a password for each service each time it is requested. Another requirement is that it be available at all times because, if a server cannot authenticate a request, it will not provide the service. When the authentication service is unavailable so are all the services that use it. A less obvious requirement is that authentication must be protected against capture and playback by another user on the network. Capture cannot be prevented on broadcast media such as an Ethernet cable, so the authentication procedure must be able to prevent a playback by an impostor.

**Kerberos**

One popular authentication system is Kerberos, which is named after a three-headed guard dog in Greek mythology. It depends on a third party that is trusted by both client and server (see Figure 13-8). Clients request a ticket from the third party. The ticket is encrypted using the server's secret password, so the server trusts the client when it can decrypt the ticket. The server's password is known only to itself and the third party. The third party knows everyone's password! This means that all systems are vulnerable if the trusted third party is compromised.

![Figure 13-8. Servers can trust clients only if they can decrypt the ticket from the Kerberos server.](image)

A well-known bank has two major data centers, one in San Francisco and the other in Los Angeles. Each center backs up its data at the other site. In this way, the bank can resume operation soon after serious damage to either data center. The Kerberos servers are replicated at both sites and kept behind "the glass wall." In fact, there is a sealed walkway with locked doors at both ends and a badge reader with a video camera in the middle. If your face doesn't resemble the one on the badge, you are not allowed into the room that houses the Kerberos servers. In fact, two or more very large people will promptly escort you out of the building.

Including a timestamp in the ticket thwarts playback. That is, the Kerberos server encrypts the client's IP address, a session key, and a timestamp using the server's key. The client encrypts its service request message with the session key and sends it, along with the ticket, to the server. The server uses its key to decrypt the ticket. If the IP address in the ticket matches the IP address in the IP packet header and the timestamp is within a few milliseconds of the current time, then the server accepts the client's request. It uses the session key to unscramble the request and perform the service. It's as simple as that. Playback is impossible because the encrypted timestamp will have "timed out" before an impostor can capture and try to replay the request. Also, the IP address of the impostor will not match the IP address encrypted in the ticket.

**Digital Signatures and Public Key Encryption**

The theory behind digital signatures and public key encryption is that in a given system every user has a pair of digital keys. In the case of the Web, the mere act of installing a browser on your system will generate the private and public keys to be used with that browser. If you have two browsers installed (e.g., Netscape and IE4), then you will have two sets of private and public keys, one set for each browser. The basic premise behind public key encryption is that using some algorithm you can use your private key to generate a permutation (encrypt) of a message that can only be decrypted using your public key. If you carefully distribute your public key to the people you normally deal with, anytime you send them a message they will be able to read it using your public key.
Secure Sockets Layer

By far the most widely used authentication and encryption on the Internet in general and on the Web specifically is Secure Sockets Layer. SSL can be used with any connection-based protocol. It's called a layer because we essentially insert an additional protocol layer between TCP and the Application layer of the TCP/IP stack (see Figure 13-9).

SSL adds the following features to the reliable stream provided by TCP/IP:

1. Authentication and nonrepudiation of the server via digital signatures.
2. Authentication and nonrepudiation of the client via digital signatures.
3. An encrypted stream to provide privacy.
4. Data integrity through message authentication codes.

Netscape Corporation designed SSL as a way of ensuring secure communications between its browser and server products. SSL has become the de facto standard for secure communications between Internet clients and servers.

For a look at the SSL v3.0 specification, see http://home.netscape.com/eng/ssl3/ssl-toc.html. To use SSL requires cooperation between the client browser and the server. At the server, a secure instance of the Web server must be running on the well-known port 443. (Some Web sites run both an unsecure and a secure instance of the Web server on the same machine. The unsecure instance is listening on port 80, while the secure instance is running on port 443. Some sites run the unsecure and secure instances on completely separate hosts.)

At the browser end, all references to the URLs of documents or applications must be preceded with the protocol https rather than http. As long as the protocol notation is https, the port is defaulted to 443 (the secure server port).

The attachment of an SSL client to an SSL server starts off with what is known as an SSL handshake. During the handshake, the client and server agree on the protocol version to be used, select the cryptographic algorithm they will use to protect data transmission, optionally authenticate one another, and use public-key encryption to generate shared secrets. After this has been done, the rest of the transmission takes place in an encrypted manner using the parameters selected during the handshake.

The Government and Security

The issue of security on our computers is greatly affected by the restrictions on security technology placed on a company by its home government. Because this is not by any stretch of the imagination a comprehensive text on security, we instead outline the two major controversies concerning government intervention in computer security. We attempt not to pass judgment on either the government or the security community; you can make that determination for yourself. Instead, in this section, we simply point out the two sides to the arguments of governmental control of security export and the government's right to possess keys to domestic security apparatuses.

Export Control

The United States government is extremely adamant about protecting against U.S. technology falling into nondomestic hands. Two of the more important regulations that are in place are the DoD International Traffic in Arms Regulation (ITAR) and the U.S. Department of Commerce Export Administration Regulations (EAR). Both sets of regulations concern the export of technology to foreign governments; ITAR primarily concerns U.S.-based defense contractors, and EAR applies to all commercial ventures that involve the sale and export of technology-related items to non-U.S. persons.
Because the Internet is a worldwide medium and social phenomenon, without boundaries and governments to hinder it, the government realizes that some form of security technology must be used to transmit information across national boundaries. Therefore, the U.S. government restricts the level of security found in certain products that are international in nature. For example, the Netscape browser has two versions. One is a U.S. domestic version with full browser security features. The other is an international version that implements the Secure Socket Layer with less security. The international version may be exported outside the United States, whereas the domestic version may be used only within the United States.

Never mind the inability to actually protect against the dissemination of the more powerful security technology to international audiences, the United States simply makes the distinction. If Netscape were to blindly distribute the domestic version without making a statement such as "Domestic Use Only," they would be breaking the law. Is the law enforceable to end users? Probably not, but the law is there, written as plain as day, and should be followed by "morally upstanding citizens." For you, as application programmers, secure networked applications should follow the same kind of export controls if they are applicable.

**The "Clipper" Controversy**

Historically, the U.S. government has always known that there are ways for its citizens to keep information hidden from the government. In fact, the Fourth Amendment to the Constitution of the United States of America specifically outlines this right that all American citizens possess:

> The right of the people to be secure in their persons, houses, papers, and effects, against unreasonable searches and seizures, shall not be violated, and no Warrants shall issue, but upon probable cause, supported by Oath or affirmation, and particularly describing the place to be searched, and the persons or things to be seized.

But, over the years, a distinction has been made as to what is "unreasonable." The government, in interests of "national security," may, with permission from the Judicial branch, execute a search of one's property and possessions. How does this apply to the digital age?

The entire "Clipper chip" controversy centers around the government's willingness to publish an encryption algorithm for telephones, computer files, and any other form of communication. The transmissions would be encrypted and mathematically impossible to break. However, the government would always be able to have a "back door" to the encryption with its own special key. As outlined in the Fourth Amendment, the government may use the key only with a written warrant; nevertheless, the idea that "Big Brother" may be watching is enough to bring chills down the spines of some people.

Lost in the argument is the fact that there are several other encryption methods that could be used instead of Clipper (e.g., PGP) and that are just as good and do not encourage governmental interference. Clipper represents the entire belief that, in the end, the U.S. government, as well as the other governments of the entire world, has no idea how to protect itself in the digital age without sacrificing intellectual freedom.

**Chapter 14. Making an Architectural Decision**

- Java Sockets
- Java RMI Decisions
- Java IDL
- JDBC
- Other Java Technologies
- Application Servers

Making a decision is difficult, particularly when the fate of your company's entire vision may be at stake. Although we make no attempt to salvage the many Titanic of free enterprise, we do offer our thoughts on what the world of Java networking can mean to you. In this chapter, we candidly browse the advantages and disadvantages of each communication alternative. Do you want the heavy-duty power of CORBA or the lightweight simplicity of RMI? Are databases vital to your business process, or do you require customizable protocols?

As we have seen, Java networking is a vast and expansive subject. This book is the tip of the iceberg, and as the industry begins to shake out, more and more information will be brought forward. This chapter will help you separate fact from fiction, reality from hype, and engineering from marketeering.
Java Sockets

Many of the alternatives we have discussed in this book involve sockets in one way or another. To recommend that you not use sockets essentially would be to say that you should not use any of the technologies we talk about. Sockets by themselves are useful for quite a few different things. Remember that, when you send an RMI or IDL message, you are essentially sending a big chunk of data and the headers to that data. When we discussed our own message format in Chapter 3, "Java Sockets and URLs," we were able to put together a small, lightweight messaging system. If speed and efficiency are of the utmost importance to you, then certainly you would be interested in using Java sockets alone.

Flexibility

Remember that we created our own message format and transmitted it with great speed. Our message format was not inadequate as it transmitted all the information we required. Notice too that we did not have to learn anything new. As long as we know what a socket is and how to use it, we can easily transmit a message to our server.

Servers are equally easy to create. With Java IDL and Java RMI, we needed to create an entire infrastructure for our server. With sockets, converting an application to a server application was not only easy but also extremely powerful. Once again, we lost no functionality by using sockets instead of some other communication alternative.

Furthermore, we could simply convert our connection-oriented socket to a broadcast socket. Then we could use the broadcast socket to send information to a port while allowing anyone else to listen in on that port. Because of this ability to switch between paradigms easily and quickly, sockets can be an excellent choice for both the beginner and the advanced networking programmer who wants to build his or her own infrastructure.

Simplicity

As we saw, using sockets is extremely simple. Once you get the concepts down, actually changing your applications to use sockets is quite an easy task. Using the ServerSocket, you can build a simple server. By integrating threads, you can make sure that your server handles data efficiently. In addition, there is no confusing IDL to learn and no RMI API to understand. By using only sockets, you sacrifice the functionality of RMI and IDL for speed and ease of use.

Because the networking world understands and knows sockets so well, having built and deployed applications that use sockets for years, you will also have a ready supply of applications to use from within Java. Because sockets do not actually send data "over the wire" and instead send strings of information, you can seamlessly plug your Java applications into new or existing applications written in other languages. Just as with Java IDL, sockets give you the promise of being able to easily integrate legacy applications.

Again, there are several tradeoffs between sockets and the other alternatives we discuss in this book. Java IDL also integrates legacy applications well, but the plug-and-play ability of Java IDL gives it a distinct advantage over using sockets alone. With sockets, you have to make sure that everyone is speaking the same protocol. With Java IDL, there is no message format or protocol to worry about. Simply invoke remote objects as if they were already on your machine.

Java RMI Decisions

After surveying the entire spectrum of Java solutions we offer in this book, it is time to make a decision. Perhaps Java RMI has piqued your interest. The promise of never having to see C++ again seems like a good thing. Using the fun and robust networking ability inherent in Java may be an even better reason to turn to a Java-only alternative. Whatever the reason, this is the place to get an honest account of what RMI can and cannot do for you.

RMI Advantages

One of the absolute best things about JavaRMI is that you never ever have to see C++ again. C++ is arcane, difficult, and frustrating. Meanwhile, Java is fun, easy, and exciting. Because Java offers the strongest alternative yet to a series of frustrations wrought upon the computer science population, Java RMI has garnered significant attention from the masses. It follows a simple notion of abstracting
distributed implementations by publishing interfaces and linking in implementations of those interfaces later on.

Because we invested a significant amount of time, money, and effort in the Java revolution by learning and promoting the language, we may be tempted to jump directly into an all-Java solution to the communication quandary. Because invocations on Java objects are simple to begin with, Java RMI makes sure that it is equally simple to make the same kinds of invocations across different virtual machines. It is precisely this simplicity that makes Java RMI appealing.

Riding on the coattails of Java 1.2, the long awaited RMI-IIOP connection is now in place. This technology allows RMI’s ease of use with CORBA's cross-language interoperability. By following a few rules, we can now mix-'n-match RMI and CORBA clients and servers.

Another new feature of RMI is Remote Object Activation. This feature allows an RMI server to be shut down once it has been registered with the registry and then be restarted remotely (functionally the same way a CORBA server can be remotely started).

**RMI Disadvantages**

With the introduction of RMI-IIOP one of Java's main drawbacks has been eliminated (i.e., the Java 1.1 restriction that RMI was a "Java Only" solution). Because we can now mix-'n-match RMI and CORBA clients, we can still put a CORBA wrapper around a legacy application and access it with a Java client application.

This leaves the old "Java is not fast" argument. Indeed, it is an interpreted language and, therefore, is subjected to a layer of processing that C++ and C are not. However, the introduction of JIT compilers and other performance enhancements (like Sun’s HotSpot technology and IBM's current JVM technology) help negate the issue. Still, it is important to realize that if performance is of the utmost importance, Java may not be the language for you.

**Three-Tier Applications in RMI**

As we discussed in previous chapters, the notion of three-tier and n-tiered client/server computing will not go away. It is the foundation for most of today's distributed systems. MIS managers love it because it enables them to funnel access to data sources through a central repository. Programmers love it because they can revise and update the various components of their applications without massive overhauls. After all, the business logic contained in servers defines how and when databases are accessed. Client-side GUIs are concerned only with getting and displaying information. If a programmer makes a change in the business logic, there is no need to push the change to the client as well.

JavaRMI does not readily facilitate the notion of three-tier client/server computing any more than JavaIDL does. Both are, in fact, middle-tier technologies. Java RMI can easily use JDBC to connect to a relational database and JNDI to connect to Directory Services just as CORBA can do with ODBC and LDAP. The real functionality, brains, and resource management take place on the server end. The data source is nothing but a repository of information.

Once again, the performance problem rears its ugly head. Because the middle tier is intended to be home to all the business logic in an object system, JavaRMI servers may have to process data extremely efficiently, perhaps more efficiently than possible.

**Java RMI Is Not Robust**

Perhaps the most important aspect of RMI is its lack of support for true distributed computing. When invoking across machines and networks, the fact is that a client generally has no control over how processes are executed on the remote end. Indeed, the remote end can very well be an entirely different hardware architecture than expected. Java RMI offers no ability to allow a client to invoke without knowing
the destination of the request. The lack of location independence should be quite a significant factor in making an architectural decision toward RMI.

Even though Java RMI is easy to understand, get started with, and design frameworks around, it does not address some of the fundamental network concerns of distributed-object programmers. Location independence is one of these concerns. Another concern is automatic startup. With the recent introduction of Remotely Activatable Objects, when a client invokes a server for the first time, as long as the server has been registered, an attempt to restart the server will be done.

One thing we shouldn't lose sight of regarding RMI registry is that it is only one possible implementation (Sun's implementation) of what is really RMI's naming service. It could be implemented a number of other ways that would allow for automatic load balancing, fail-over, and all those things CORBA is famous for.

This requires the server programmer not only to have the server available but also to provide for fault tolerance. What if the server goes down unexpectedly? Part of the software design specification should provide an automatic fail-over to a backup server or automatically restart the server itself. Needless to say, these are difficult tasks to program and may be more trouble than they are worth.

Java IDL

Every year for the past 3 years was touted as "the year CORBA will break out." Every January a flood of articles in trade rags and industry newsletters trumpets the arrival of the Common Object Request Broker Architecture. Although it is anyone's guess as to what the future will be, it is a relatively safe assumption that CORBA, or a derivative thereof, will power the forces of the Internet for quite some time. The reasons are numerous, but the fact remains that CORBA technology, although not devoid of major shortcomings, is the most robust, mature, and powerful alternative presented in this book. Any investigation into an Internet communication strategy should place CORBA at the top of the list of technologies to investigate.

Advantages of Java IDL

Java IDL is a well-thought-out, coherent set of base objects that can be used to create a tightly woven distributed-object system. Because of the maturity of CORBA, many of the questions about Java RMI and sockets have been addressed in the specification and in the products currently available. In a moment, we will discuss the advantages and disadvantages of the various implementations of the specification that are on the market today. Yet, regardless of the great number of ORBs, Java IDL is a solidly engineered set of core components that facilitate Java to ORB programming.

As we have discussed, the ORB isolates an object from the underlying mechanisms that ensure that a client does not need to know the physical location of a server, how to start the server, or even if it should shut the server down. When you walk into a supermarket, the doors are automatic. You don't have to open them automatically, and you don't have to close them behind you. Similarly, an ORB handles a lot of the internal machinations of networked communication for you.

Beyond its maturity and the fact that it handles much of the boredom of working with networked objects, Java IDL is also Java. It uses the same memory handling, parameter passing, serialization, and so on, that Java does and, therefore, helps to alleviate the learning curve of CORBA itself.

Disadvantages of Java IDL

Java IDL's biggest disadvantage is also one of its strong advantages: Java IDL is CORBA. CORBA is a complex series of rules and regulations (in the software sense) governing how distributed objects should behave. Java IDL is completely CORBA compatible and is, therefore, an extension of CORBA itself. It plugs into CORBA very easily and without much hassle, but at a pretty steep price. In order to use Java IDL effectively, you must understand CORBA and truly understand the principles of distributed objects. Even
though this book attempts to outline what CORBA is and why you would want to use it, it is not the ultimate resource.

Yet, because of Java, much of the memory management morass and the differences between varied ORBs is rendered moot because the nature of Java removes it. Java is platform independent and requires no memory management on the programmer’s part. Even though CORBA programming is hard, thank your lucky stars for Java. Just taking a look at a C++ CORBA program compared to a Java CORBA program will make a Java believer out of you.

### Java IDL Implementations

Sun Microsystems is a major proponent of CORBA but has announced that it is getting out of the business of providing full-featured ORBs (NEO/JOE) as a product, and it is deferring to such companies as IONA (Orbix), BEA, and Borland/Inprise (Visibroker). Inprise's Visibroker is a smart, easy-to-use CORBA option that offers strong three-tier client/server capabilities. If talking to a database is of the utmost priority for your software architecture, Visibroker for Java might be your best option.

The current industry leader is Orbix. Orbix is available on every platform and is a reliable, easy-to-use object broker. Many customers find getting started with Orbix to be a relatively easy task and discover soon thereafter that CORBA isn't as bad as it was cracked up to be.

One of the biggest problems with the various CORBA implementations is that the code is not portable from one ORB to another. Although they all comply with the CORBA specification, the specification is general enough for each implementor to do it its own way. APIs from one ORB to another are quite different.

### Java IDL Is Robust

Imagine creating a client application that can invoke a server, get information, and report results without even once having to worry about network code, server-side behavior, or slow system resources. CORBA, and the ORB specifically, handle all those tasks for you. So long as an ORB is on both the client and the server platforms, the request can get through to the server, the server can be started up if necessary, and the server can process information for the client.

The notion of an ORB on every platform is not as far-fetched as you might expect. Sun's Solaris operating system is incorporating Sun's own NEO family of CORBA products directly. When you get Solaris, you will also get the plumbing necessary to create CORBA fixtures. Similarly, OLE and COM have always been present on the Microsoft Windows operating environment, and with CORBA offering a strong OLE/COM to CORBA connectivity solution as part of its specification, the client side on Windows platforms will soon be a reality.

Furthermore, a Java IDL application also includes its own "mini-ORB" that provides limited functionality so that an ORB need not be present within the Web browser itself. Netscape, however, as part of its ONE technology includes a version of the Visibroker ORB with every 4.0 or newer browser. In this way, the Web browser can act as a communication mediator between clients and CORBA servers.

### Java IDL Is Difficult

One of the big gripes we have heard and emphasized in this book is that CORBA is difficult. Well, there's no getting around the fact that in the past you had to be a true C++ expert to understand CORBA itself. You could allocate a chunk of memory on the client side, pass it to the server, where it got deallocated, and still have a memory leak on your client side. That was just one of the many, many, many problems with C++ and CORBA.

Yet, that is much more of a C++ problem than a CORBA problem. True, you still need to know much more than the basics of object-oriented programming to use CORBA, but with Java things become much easier. Memory management, for one, is no longer even an issue.
The Interface Definition Language is blasted by critics as just one more thing you need to know in order to use the CORBA architecture. True, the IDL is a layer on top of your normal application, but it serves a very important purpose. It prevents your applications from being locked into one language. Who knows? Tomorrow, a new programming language may emerge with its own cool name, its own cult following, and its own list of strengths. The entire world may jump on that bandwagon much as it has with Java. But CORBA applications still will be important and will not be rendered obsolete because they can be phased into the new language in a short time without affecting the rest of the system.

Language independence, while not of real importance to the subject of this book, is the single most interesting thing about CORBA. It enables you to migrate applications to new platforms, new languages, and even new algorithms without having to adjust the entire object system. Remember that, with JavaRMI, you are locked into Java until you have a reason to change. That kind of thinking is why many people are trying today to figure out how to migrate from COBOL.

**Java IDL Is Powerful**

Java IDL is a flexible, distributed-object environment. With it, you can invoke C++ objects half a world away as if they were both local and written in Java. To you, the application programmer, the Java to CORBA to C++ is hidden. You simply instantiate Java objects and talk to C++ servers on the other end without even knowing. Of course, if you prefer to write Java servers, more power to you.

Remember that language independence is a very good thing for large-scale object systems. You can swap components in and out using the language most appropriate for the task. If you happen to have a CORBA to LISP language mapping (don't panic, there isn't one), you could write all your artificial intelligence components in LISP, while saving UI or computation components for an object-oriented language like Java or C++. Java IDL is the only alternative we present that can possibly integrate such disparate object components.

But, for many people, the simplicity and elegance of Java RMI may be all that is needed. Maybe you don't have any legacy systems to be integrated. Maybe language independence is of no use to you. Maybe all you want is a simple remote object invocation system. In that case, Java RMI is definitely your cup of tea.

**JDBC**

Java Database Connectivity is an enabling technology, not necessarily a communication framework in and of itself. By "enabling technology," we mean that it enables you to link other communication strategies with repositories of information and data to form a cohesive network of objects that can communicate vast quantities of information. JDBC is not the answer in itself, but in combination with Java IDL, Java RMI, or even Java Sockets, it can be a heck of a powerful answer to the Internet question for the next decade.

**Why JDBC Is Not Enough**

JDBC alone limits you in what you can accomplish with advanced networking. Every client that talks to a database connects directly with the database. There can be no additional intelligence added in the business logic to assist with routing messages. Basically, your applications are connected to the database, and if that causes some kind of sluggishness between the database and the client, then so be it. In the end, the decision to use JDBC alone or with another technology amounts to a decision between the two-tier and three-tier architectural models.

The two-tier architecture links clients directly with the data repository as shown in Figure 14-1. This means that any kind of processing for the access and any further processing for the data retrieved from the repository is left to the client. Splitting the business logic out of the client is the driving force behind the three-tier model. However, in some cases that trait is not a necessary qualification. If your applications are deployed often, or maybe even deployed over the Web, then updating a client is not a major factor because it will be done no matter what architecture you choose. If you are deploying shrink-
wrapped applications written in Java—as will be common in just a few years—then updating applications constantly will be a major pain, and you may want to revert to a three-tier model.

**Figure 14-1. Two-tier client to database architecture.**

![Diagram of two-tier architecture](image1)

The biggest drawback to the two-tier model is the sheer number of clients that may attach themselves to a data repository. Typically, data repositories are not set up to handle the intelligent management of resources required to process multiple simultaneous invocations. If your applications ping the database only rarely, then this is not a factor for you. However, if there are to be many instances of your client application, you will want to go to a three-tier model.

A three-tier model is predicated on the belief that business logic should not exist in either a client or a database. It dictates that the client should be a pretty application the sole purpose of which is to funnel information back to the user. The client is typically a rich GUI with simple execution steps that relies completely on the information given to it by the middle tier (see **Figure 14-2**).

**Figure 14-2. Three-tier application architecture with server middleware.**

![Diagram of three-tier architecture](image2)
The middle-tier is a server that talks to a data repository. The server is written using Java IDL, Sockets, or Java RMI and can talk to the database using JDBC. JDBC acts, as it always does, as the interface from a client (in this case the middle-tier server) to a database. It just so happens that the server is fully capable of handling multiple invocations and requests and houses all the business logic. The business logic could range from simply adding a number of results from a database query and passing it back to the client, to invoking other servers using the same data. Whatever it does with the data it retrieves, the server can manipulate the information as it sees fit and then pass it back to the client.

**JDBC and Java IDL or Java RMI**

As we discussed, the middle tier in the three-tier architecture could easily be Java IDL or Java RMI. Indeed, IDL and RMI are complementary technologies to JDBC. JDBC is not their competition because the vast majority of people using JDBC use it within a middle-tier paradigm. This is why Java IDL and Java RMI are vital to JDBC's success. Moreover, JDBC lends credibility to Java IDL and Java RMI. Without a simple technology to enable database access, Java IDL and Java RMI would be largely useless in the business community.

The largest investments made by most companies in their computing infrastructure is contained within their databases. Databases often are used to maintain important records ranging from medical history to employment records and to keep track of business processes from supply purchases to stock maintenance. Most of the time, changing the database to a Java-only application is not only difficult and expensive but also completely unreasonable and unfeasible. For this reason, JDBC can be used to communicate and update the database, while the middle-tier server can be quickly migrated to Java using the techniques in this book.

Client applications can be generated quite easily using the many visual Java builders on the market today. Often, client applications are not only simpler to create, test, and deploy but are also less vital and less error prone than the rest of the architectural model.

**JDBC Alone**

While using JDBC alone is certainly not out of the question, it is highly discouraged for mission-critical applications. However, for proof of concept applications, applications requiring limited data access, and even for heavy-duty applications with large chunks of data transfer, JDBC may be an excellent option.

What JDBC gives you is a simple, clean interface to a database that requires no additional knowledge of network programming, distributed design, or remote procedure calls. For database programmers, JDBC is a welcome arrival for Java because it means that they need not build special server programs whose sole purpose is to funnel information back to the client. In other words, for those programmers who desire not to use three-tier computing, JDBC is the perfect answer.

Because of its simplicity, you will find that, for major application development efforts, JDBC is all you need to affect some kind of persistence for your client applications. Clients can do their heavy computation, cool graphics, or whatever and store their state in a database using JDBC. The next time the client is executed, it can retrieve its previous state from the database and start again where it left off.

**JDBC Overview**

JDBC is a fantastic set of APIs to connect Java applications and applets directly to databases. With its simplicity, robustness, and ability to bring together the disparate worlds of databases and the Internet, JDBC will be a successful venture for Java. By modifying your existing database clients for Java, you can capture all the usefulness of the Java Revolution without sacrificing the power required to manipulate your data stores.
Other Java Technologies

In addition to the four major Java communication technologies, we have shown you three other mechanisms that you can use to plug your Java applications into the Internet. Beans, servlets, and JMAPI give you the means necessary to package, publish, and administer the applications you have written in RMI, IDL, JDBC, or Sockets. Even though the "big four" are fascinating and powerful in their own right, they need the additional functionality provided by the other Java APIs that have been or will be published in the future.

When to Use Beans

Let's say that you've created a bunch of gee-whiz Java applications to interface with your hand-held Personal Information Manager. These applications have several modules that translate the data on the device to a format that is readable by your on-disk schedule manager. These modules are for your address book, to-do list, and schedule. By dividing your Java applications into separate, self-contained Beans, you can publish the components. Moreover, if you were to split out the network component that interfaces the device with your computer, then others could write their own customizable applications that use your network module (see Figure 14-3).

**Figure 14-3. Beans enable you to build components such as the Schedule component that can be used by other applications.**

This is precisely what we intended to do with our featured application. Although we didn't exactly use Beans, we could have done so easily and allowed others to pick out the Beans they wanted and interface with our calendar manager. Currently, the network module talks to a server on a remote machine. The server stores the information on the disk on which it resides.

What if we were to modify our calendar manager to use Beans? It would simply be a matter of encapsulating our various Java objects in Java Beans containers. Then we could allow anyone who wanted to interface to the rest of our calendar manager to do so using the Network module. Remember that Java Beans supports the notion of introspection, which enables people to take our Network module, browse it from within a GUI builder, and then generate their own objects that interface directly to it. Even if they do not like our user interfaces, people still can use the Network module rather than invest their own time and effort into learning the RMI, IDL, or JDBC APIs.
When to Use Servlets and Java Server Pages

Servlets and Java Server Pages are information publishing tools. If we wanted the people in our department to know what our schedule is simply by browsing our personal Web page, we could allow them to do so by sticking the server portion of our calendar manager inside a servlet. The servlet then could be queried via an HTTP request, and the information contained within the server could be displayed on the Web page. Then, when we modify our server, people talking to our servlet would get the latest and greatest list of what we are doing that day (see Figure 14-4).

Figure 14-4. Servlets provide dynamic documents via Web servers.

The alternative to servlets is to create a Web page by hand and stick it on a Web server. But, if we were to change the times of our appointment, we would have to generate a new Web page. By incorporating the servlet technology within our server, we do not have to regenerate a Web page every time. Remember that the entire Internet game is about information—how to get, disseminate, and update information constantly. Servlets enable you to publish information contained within servers that get and update that information constantly.

Java Server Pages are an extension of servlet technology and allow the initial creation of server-side Web pages to be done using traditional GUI-based html editors. After we get the page to look the way we want, we attach it with a text editor or our favorite Java IDE and add Java functionality via the JSP API. Once we rename it from .html to .jsp we have our JSP, and it can be pretty much managed and served as a plain old html page.