CHAPTER 1

Overview of Jini

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

Jini

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

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

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

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

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

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

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

Components

Jini is just one of a large number of distributed systems architectures, including industry-pervasive systems, such as CORBA and DCOM. It is distinguished by being based on Java and deriving many features purely from this Java basis. One of the later chapters in this book discusses bridging between Jini and CORBA, as an example of linking these different distributed architectures.

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

You should be aware that Jini is only one competitor in a non-empty market. The success or failure of Jini will result partly from the politics of the market, but also (hopefully!) the technical capabilities of Jini, and this book will deal with some of the technical issues involved in using Jini.

In a running Jini system, there are three main players. There is a service, such as a printer, a toaster, a marriage agency, etc. There is a client which would like to make use of a service. Third, there is a lookup service (service locator), which acts as a broker/trader/locator between services and clients. There is one additional component, and that is a network connecting all three of these. This network will generally be running TCP/IP. (The Jini specification is fairly independent of network protocol, but the only current implementation is on TCP/IP.) See Figure 1-1.

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

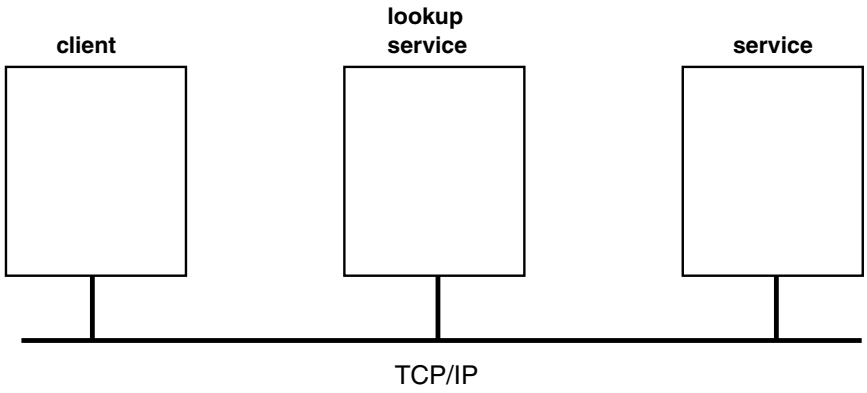

Figure 1-1. Components of a Jini system

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

Service Registration

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

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

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

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

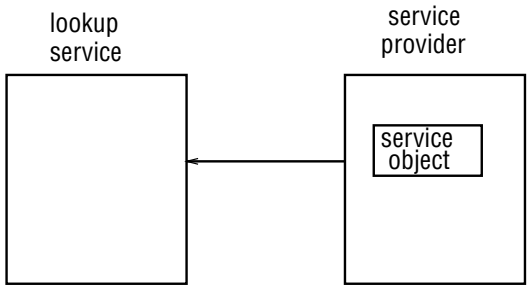

Figure 1-2. Querying for a service locator

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

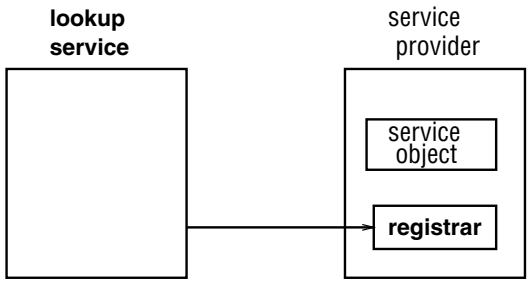

Figure 1-3. Registrar returned

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

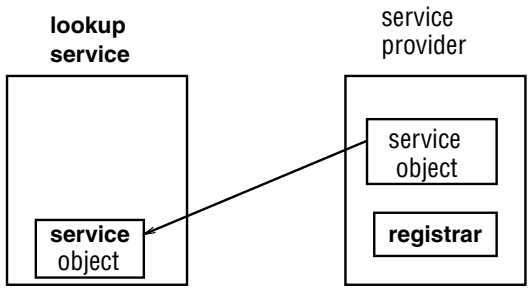

Figure 1-4. Service uploaded

Client Lookup

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

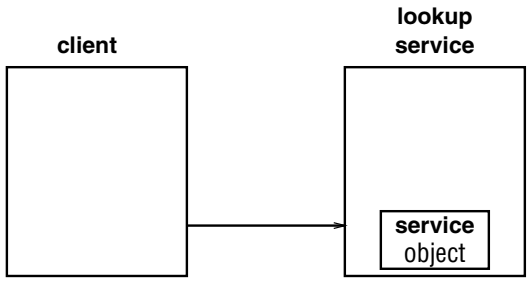

Figure 1-5. Querying for a service locator

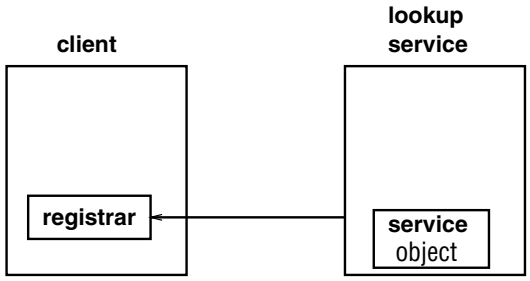

Figure 1-6. Registrar returned

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

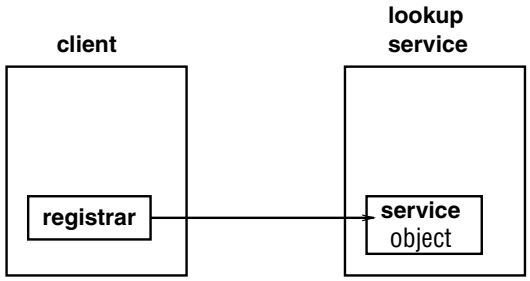

Figure 1-7. Asking for a service

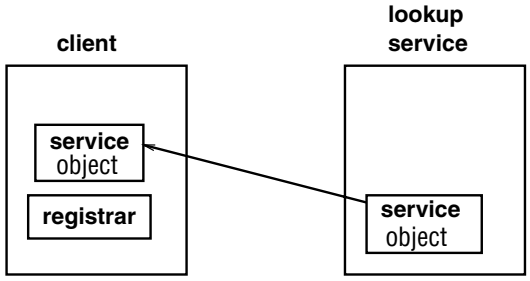

Figure 1-8. Service object returned

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

Proxies

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

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

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

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

How is the proxy primed? This isn't specified by Jini, and it can be done in many ways. For example, an RMI naming service can be used, such as rmiregistry, where the proxy is given the name of the service. This isn't very common, as RMI proxies can be passed more directly as returned objects from method calls, and these can refer to ordinary RMI server objects or to RMI activable objects. Another option is that the proxy can be implemented without any direct use of RMI and can then use an RMI-exported service or some other protocol altogether, such as FTP, HTTP, or a home-grown protocol. These various possibilities are all illustrated in later chapters.

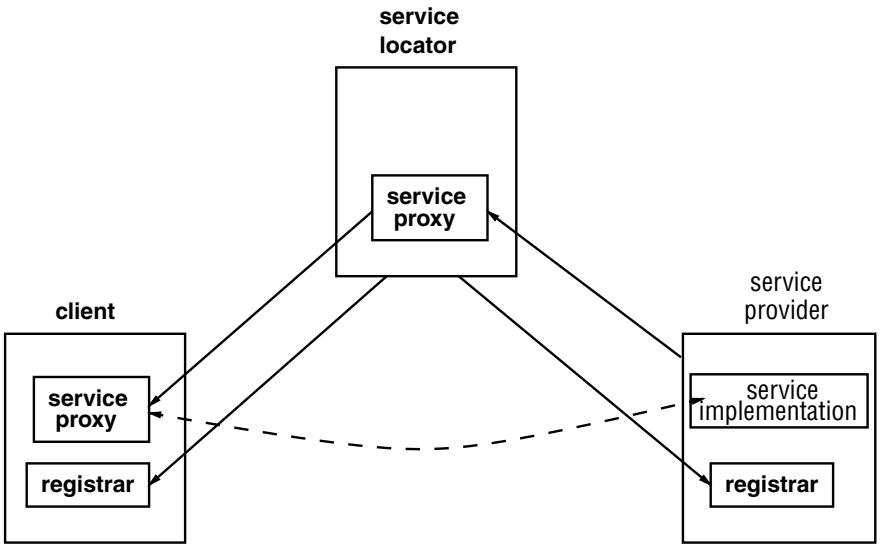

Figure 1-9. A proxy service

Client Structure

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

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

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

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

```
public class TestUnicastFileClassifier {

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

  public TestUnicastFileClassifier() {
    LookupLocator lookup = null;
    ServiceRegistrar registrar = null;
    FileClassifier classifier = null;

    // Prepare for discovery
    lookup = new LookupLocator("jini://www.all_about_files.com");

    // Discover a lookup service
    // This uses the synchronous unicast protocol
    registrar = lookup.getRegistrar();

    // Prepare a template for lookup search
    Class[] classes = new Class[] {FileClassifier.class};
    ServiceTemplate template = new ServiceTemplate(null, classes, null);

    // Lookup a service
    classifier = (FileClassifier) registrar.lookup(template);

    // Call the service
    MIMEType type;
    type = classifier.getMIMEType("file1.txt");
    System.out.println("Type is " + type.toString());
  }
} // TestUnicastFileClassifier
```

Server Structure

A server application will internally look like this:

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

Again, the following code has been simplified by omitting various checks on exceptions and other conditions. It exports an implementation of a file classifier service as a `FileClassifierImpl` object. The full version of the code is given in Chapter 8. I won't give detailed explanations right now—this is just to show how the preceding schema translates into actual code.

```
public class FileClassifierServer implements DiscoveryListener {

  protected LeaseRenewalManager leaseManager = new LeaseRenewalManager();

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

    // keep server running (almost) forever to
    // - allow time for locator discovery and
    // - keep re-registering the lease
    Thread.currentThread().sleep(Lease.FOREVER);

  }

  public FileClassifierServer() {
    LookupDiscovery discover = null;

    // Prepare for discovery - empty here

    // Discover a lookup service
    // This uses the asynchronous multicast protocol,
    // which calls back into the discovered() method
    discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
```

```
    discover.addDiscoveryListener(this);
  }

  public void discovered(DiscoveryEvent evt) {
    ServiceRegistrar registrar = evt.getRegistrars()[0];
    // At this point we have discovered a lookup service

    // Create information about a service
    ServiceItem item = new ServiceItem(null,
      new FileClassifierImpl(),
      null);

    // Export a service
    ServiceRegistration reg = registrar.register(item, Lease.FOREVER);

    // Renew leasing
    leaseManager.renewUntil(reg.getLease(), Lease.FOREVER, this);
  }
} // FileClassifierServer
```

Partitioning an Application

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

To support this claim, we can look at a smart file viewer application. This application will be given a filename, and based on the structure of the name will decide what type of file it is (`.rtf` is Rich Text Format, `.gif` is a GIF file, and so on). Using this classification, it will then call up an appropriate viewer for that type of file, such as an image viewer or document viewer. A UML class diagram for this application, using a standard object-oriented approach, might look like Figure 1-10.

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

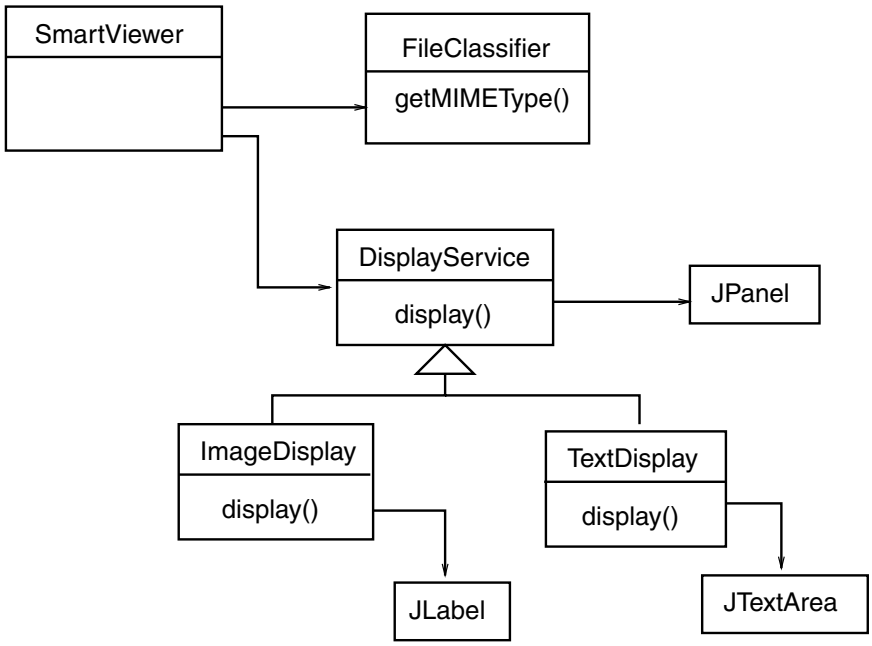

Figure 1-10. UML diagram for an application

However, this is not to say that every class should become a service! That would be overkill. What makes these qualify as services is that they all

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

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

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

Each service may be running on a different machine on the network (or on the same machine—it doesn't matter). Each exports a proxy to whatever service locators are running. The `SmartViewer` application finds and downloads whatever services it needs, as it needs them.

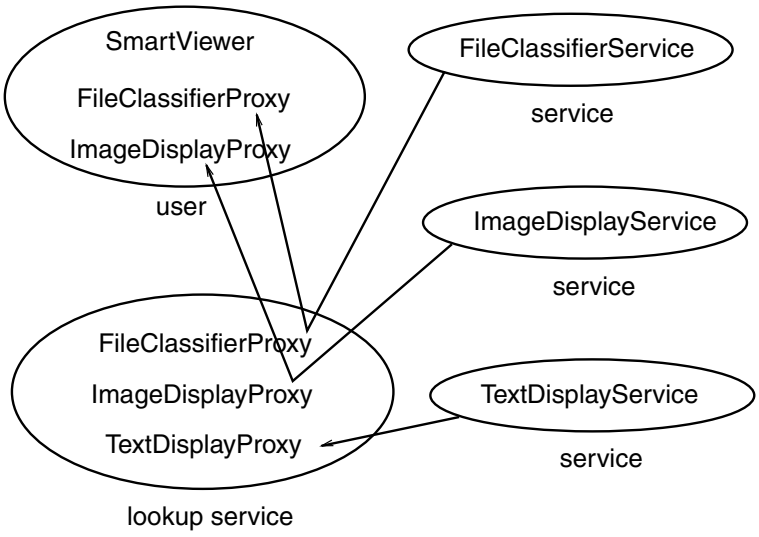

Figure 1-11. Application as a collection of services

Support Services

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

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

HTTP Server

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

snapshot of the object's state is taken using serialization, and this snapshot is moved around.

An object consists of both code and data, and it cannot be reconstituted from just its data—the code is also required. So, where is the code? This is where a distributed Jini application differs from a standalone application or a client-server application: the code is not likely to be on the client side. If it was required to be on the client side, then Jini would lose almost all of its flexibility because it wouldn't be possible to just add new devices and their code to a network. The class definitions are most likely on the server, or perhaps on the vendor's home Web site.

This means that class definitions for service proxy objects must also be downloaded, usually from where the service came from. This could be done using a variety of methods, but most commonly an HTTP or FTP protocol is used. The service specifies the protocol and also the location of the class files using the `java.rmi.server.codebase` property. The object's serialized data contains this codebase, which is used by the client to access the class files.

If the codebase specifies an HTTP URL, then there must be an HTTP server running at that URL and the class files must be on this server. This often means that there is one HTTP server per service, but this isn't required—a set of services could make their class files available from a single HTTP server, and this server could be running on a different machine than the services. This gives two sets of class files: the set needed to run the service (specified by `CLASSPATH`) and the set needed to reconstitute objects at the client (specified by the `codebase` property). For example, the `mahalo` service supplied by Sun as a transaction manager uses the class files in `mahalo.jar` to run the service and the class files in `mahalo-dl.jar` to reconstitute the transaction manager proxy at the client. These files and support services are shown in Figure 1-12.

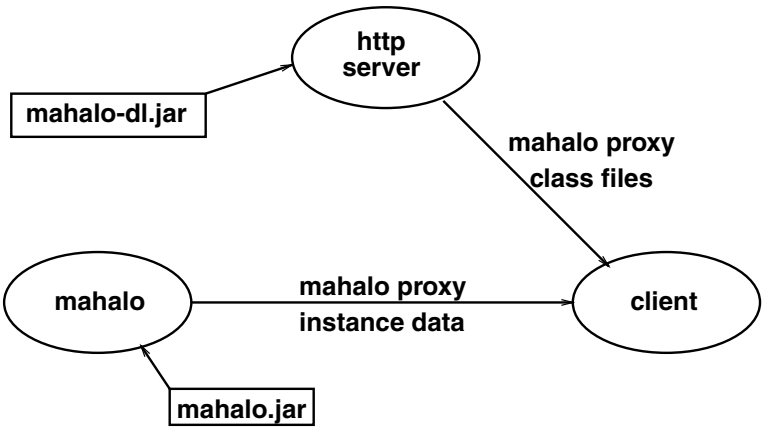

Figure 1-12. Support services for mahalo

To run `mahalo`, the `CLASSPATH` must include `mahalo.jar`, and to reconstitute its proxy on a client, the `codebase` property must be set to `mahalo-dl.jar`.

RMI Daemon

As mentioned earlier, a proxy service gets exported to the client, and in most cases it will need to communicate with its host service. There are many ways to do this, which are discussed in full in later chapters. One mechanism is the Java Remote Method Invocation (RMI) system. This comes in two flavors in JDK 1.2: the original `UnicastRemoteObject` and the newer `Activatable` class. Whereas `UnicastRemoteObject` requires a process to remain alive and running, `Activatable` objects can be stored in a passive state and the Activation system will create a new JVM if needed when a method call is made on the object. While passive, an activatable object will need to be stored on some server, and this server must be one that can accept method calls and activate the objects. This server is called an *RMI daemon* and Sun supplies such a server, called `rmid`.

This is really obscure and deep stuff if you are new to RMI or even to the changes it is going through. So why is it needed? Sun supplies a service locator called `reggie`, and this is really just another Jini service that plays a special role. It exports proxy objects—the `registrar` objects. What makes this complex is that `reggie` uses `Activatable` in its implementation. In order to run `reggie`, you first have to start an `rmid` server on the same machine, and then `reggie` will register with it.

Running `rmid` has beneficial side-effects. It maintains log files of its own state, which includes the activable objects it is storing. So `reggie` can crash or terminate, and `rmid` will restore it as needed. Indeed, even `rmid` can crash or be terminated, and it will use its log files to restore state so that it can still accept calls for `reggie` objects.

Summary

A Jini system is made up of three parts:

- Service
- Client
- Service locator

Code is moved between these parts of applications. A `registrar` acts as a proxy to the lookup locator and runs on both the client and service.

A service and a client both possess a certain structure, which is detailed in the following chapters. Services may require support from other non-Jini servers, such as an HTTP server.