Introduction to Distributed Objects

The idea of distributed objects is an extension of the concept of remote procedure calls. In a remote procedure call system (Sun RPC, DCE RPC, Java RMI), code is executed remotely via a remote procedure call. The unit of distribution is the procedure / function /method (used as synonyms). So the client has to import a client stub (either manually as in RPC or automatically as in RMI) to allow it to connect to the server offering the remote procedure.

Object as Distribution Unit

In a system for distributed objects, the unit of distribution is the object. That is, a client imports a "something" (in Java's JINI system, it's called a proxy) which allows the client access to the remote object as if it were part of the original client program (as with RPC and RMI, sort of transparently). A client actually imports a Java class and this becomes part of the set of classes available on the client machine. So, for example, the client can define subclasses of this imported class, overload methods, etc.

In addition, distributed object systems provide additional services, like a discovery service that allows clients to locate the objects they need, security services, reliability services, etc.

Distributed object technologies:

1. DCOM (Distributed Common Object Model), developed by Microsoft, but also available on other platforms. Built on top of DCE's RPC, interacts with COM.
2. CORBA (Common Object Request Broker Architecture), defined by the Object Management Group, an industry consortium. CORBA is available for most major operating systems
3. JINI ("Genie," JINI is not initials — a joke; JINI actually doesn't stand for anything.) JINI is developed on top of Java. JINI was released by Sun in January, 1999.

JINI

TDS, Chp 12.3

JINI is a distributed object technology developed by Sun, partly to make better distributed programming tools available to Java programmers, and party to overcome some of the inherent problems with distributed programming.

RPC systems handle issues such as data transport, data formatting, finding correct port number, and to some degree finding the machine a server is running on.

However, JINI handles additional problems:

- Finding a service if you don't know the name of it!! Suppose you want to find a laser printer (but you don't care which one) and so you don't know a specific name to look for. None of the RPC systems we looked at handle this issue. JINI does, by allowing the client to search for a service based on attributes.
- Finding a replacement service if the service you've been using becomes unavailable, either because of network failure or server failure. (JINI again)
- Automatic discovery — client and server discover each other automatically, and discover what they need to know about each other.
- Coordination. Allows processes to coordinate their activities.

JINI Scenarios

Motivating examples for automatic discovery features of how JINI might be used.

Automatic discovery:

- telephone automatically finds answering machine
- refrigerator finds handheld PC to add milk to the shopping list (or sends message to home delivery service so milk is on front door step)
- digital cameral finds printer to print on
- PC automatically finds a printer on a network

Basic JINI Concepts

Required Servers

The following servers must be run to use JINI

- Web server. This must run on any machine which will host services, because JINI uses HTTP to transport code.
- RMI activation daemon (rmid) must also run on any machine that will host services
- JINI Lookup Service (reggie) must run on at least one machine

JINI services are organized into *communities.*

All the machines in a community will have access to the same set of services (shared resources), and a community must have one or more Lookup Services running on it. If there's more than one Lookup Service in a community, then they make the same set of services available, and the multiple Lookup services are for redundancy in case of a failure or for improved performance.

By default, a community is all the machines on your local network. If the administrator chooses, the community may be set up to be a smaller group (for example, at WPI the communities could be organized on department level, or in a company at the level of a workgroup.) Also, distinct communities can be federated, so that (some or all of) the services in one community are made available to clients in another community. In this way, the idea of a JINI community is scalable, and the presence of a central (per community) Lookup Service is not a barrier to scalability.

Key concepts of JINI:

1. Discovery
2. Join
3. Lookup
4. Leasing
5. Remote Events
6. Transactions
7. Coordination

Discovery

Discovery is the process by which client find which community they belong to, and where their lookup service is located. There are several discovery protocols:

- Multicast Request Protocol (Fig DJ1.1, AR2.2) — used when an application or a service wishes to locate a Lookup Service so it can register itself. The message is sent to a “Lookup Server” multicast address
- Multicast Announcement Protocol is used when a Lookup Service wishes to announce its availability as a Lookup Service. Potential JINI clients will be listening to the multicast address so they will know where to find a Lookup Service.
- Unicast Discovery Protocol is used when a service knows the name and address of the service it needs to communicate with. Services are addressed with a URL using the protocol name of jini, and in jini://jiniserver.wpi.edu Unicast Discovery Protocol might be used, for example, when one lookup service is contacting another one to federate the two communities. Another use of the Unicast Discovery Protocol would be when the service wanted to join a community other than its default one (which is what would be found through the Multicast Request Protocol). For example, if a group was developing a new JINI service, they might want to put it in a ”experimental” community, so that developers could test it out but normal users wouldn’t be able to find it.

Registering a Service (Join)

See Fig AR2.3

The name of the service that runs the Lookup Service is called the ServiceRegistrar , hence ”reggie”. The discovery process returns a interface to the ServiceRegistrar, which a service uses to register. The service uses the register() method of this interface to register itself with the ServiceRegistrar. The service passes two parameters to the ServiceRegistrar: a proxy and a set of attributes.

Service Object

The proxy is serializable Java object (if you don’t know Java, it’s just Java code that can be downloaded to another machine, like applets are). The proxy is what is downloaded to the client when the client wants to use the service. The client calls a method in the proxy, and the proxy takes care of contacting the server, or whatever it has to do to perform the service. The proxy is like the RMI stub (which is downloaded from the server to the client and runs on the client to handle the communications with the server) but it’s much more general. It could just be an RMI stub, which marshals the data and communicated it to the server. Bu, for example, the proxy could contact three different machines if that’s what was necessary to perform its service (for example, the proxy might have to consult databases that live on several different machines). The proxy might be something like a printer driver, so that when a new printer becomes available, clients obtain the printer driver for the new printer through the JINI Lookup process. The proxy can even do all the work within its own code without contacting any other machine, if that’s what’s appropriate. So the proxy is mobile code which is downloaded to the client to perform any necessary work.

Move Code to Clients

Proxies really represent a paradigm shift in distributed programming. In the past, the principle of distributed computing was to move the data to the code. That is, you had code (servers) running on certain machines, and to use them, clients sent data to the server machines. Even in RPC or RMI, which involved running stub programs on the client machines, task of the code on the client machines was just to move the data to the server machines (marshalling, data conversion, network transport), so RPC / RMI still represents a move-the-data-to-the-code philosophy. With proxies, you can move arbitrary code to the clients. (Mobile code is of course one of the key ideas of Java — applets are a prime example of mobile code.)

Service Attributes

The attributes are values the service uses to describe itself to the ServiceRegistrar and, through the ServiceRegistrar, to potential clients. There are some standard attributes that a service must provide: service name, location (server machine and port number), comments. But it's also possible to have other attributes. A printer, for example, might have attributes that indicate that it's a printer, a laser printer, and a color printer. Then when a client is looking for a color printer, it can search for attribute values rather than a service name. According to the Core JINI book, Sun is having conversations with industry groups to come up with a standard set of attributes and values for common services.

Finding a Service (Lookup)

The ServiceRegistrar interface has a method called lookup, which allows clients to look for a service. The lookup call allows the client to specify the type of the proxy (i.e., the name of the interface the proxy implements), the unique identifier of the service, or by attribute values. As a result of the lookup call, the client gets the proxy, which it uses to perform the service.

How does the client know what methods the proxy has. The client needs to know the names of the methods ahead of time.

One possibility is that the client knows the specific interface that the server implements. For example, if a client needs to print a bitmap, it might look for a service that implements the GraphicPrinter interface. (I just made this up, but something like this could be a standard JINI interface name). The they would know that the proxy that implemented this interface would have a printBitmap method, and so the client could call that method.

Another possibility is that the proxy would have its own user interface, and so the client machine wouldn't have to know what the proxy does, other than that it displays an interface to the user. So the client could search for services that have an attribute "arcade game", download the proxy, and run it. Then it would be up to the (human) user to interact with the interface; the user's commands to the interface would be handled by the proxy, which might modify the interface on its own, and might pass on data to the server. Another possibility is controlling a remote digital camera, which the user could control from an interface on the client machine.

Leasing

Leasing refers to the fact that instead of giving clients access to resources indefinitely, servers grant permission to use their service for a fixed period of time (a lease). Leases can be renewed, but only by explicit action of the client.

One way leasing is used is by the Lookup Service. From the standpoint of the Lookup service, the servers that register with it are clients. The lease from the Lookup Service to its clients (the JINI services) are for a fixed duration, and must be renewed periodically by the servers. So if a service crashes, or if it's removed from the system without a proper shut-down (for example, someone unplugs the printer to remove it, so it can't remove itself from the Lookup Service), eventually it just disappears from the Lookup Server, and clients will not be directed towards this non-existent service indefinitely. (In the RPC portmapper for example, a registered service stay there forever unless it's explicitly removed.) So leasing is one way in which a Java community is self-repairing, increasing reliability.

Another way leasing is used is between a JINI service and its clients. The same principle applies. Since the server only grants a lease for a fixed period of time. If the server dies, the client will not be able to renew its lease after the next lease expiration, and the client will have to go back to the Lookup Server to find another server to perform this service. More self-healing, more reliability. Contrast this with what happens if a network printer dies on LAN. On the CS system, the only way you know that the printer has died is that your printing doesn't appear. (You can actually check by running lpq —Pprintername, but you have to think to do this.) The some administrator has to clear out the print queue and re-start the printer. Much easier if we used JINI to connect the client and printer.

Remote Events

The Java event model can be used to allow services to notify clients when events of interest occur. There is a Java interface RemoteEventListener that must be implemented by objects that which to receive events, and single method in this interface, notify(). RemoteEventListener itself is a Remote interface, so notify can be invoked by remote objects via RMI. This means that a server can invoke a method running on a client. (This surely is a violation of the normal sense of what clients and servers can do — a good thing because of the power it gives you in designing distributed applications.) Of course the client has to be programmed to work correctly with the events that the server might invoke.

So what can you with remote events? Back to the example of the digital camera (the client) looking for a printer (service). Suppose when the camera is placed on the network, no appropriate printer is available on the network. (The camera queried the Lookup Service and didn't find a printer service.) The client could listen for an event, to be invoked by the Lookup Service, to inform it when a printer becomes available. So the Print button on the camera interface would be greyed-out when the camera starts up (because no printer is available) and then would automatically be enabled when a printer becomes available.

Alternatively, a printer could trigger events on their clients when some unusual event occurs, like out-of-paper, and the client could take appropriate actions (like relaying this message to the user, or making the printer unavailable until the condition is corrected.

Transactions

A problem with any kind of sequence of operations (whether distributed or not) is that there might be a problem if only some of the actions in the sequence are completed. For example, suppose you wanted to transfer $100 from your checking account to your savings account. This would be accomplished by a series of actions like this:

1. Subtract $100 from your checking account
2. Add $100 to your savings account

You would be fairly upset if step 1 got executed but the machine crashed before step 2 got executed — you would be out $100. The general computer science soluntion to this problem is to use transactions: transactions assure that all or none of the steps within a transaction are executed. From a transactional point of view, it doesn't matter whether all or none of the steps are performed — either one leave the system in a consistent state. (If you've taken the database course, you learned all about transactions. This isn't the place to go over how transactions are executed.)

In a distributed system, there are even more chances for things to go wrong — you can have serveral servers involved in processing the steps, servers can go down, messages can get lost, etc. JINI provides a transaction-based interface, TransactionParticipant, which allow a client and server to implement transaction-based series of actions. The TransactionParticipant interface doesn't actually provide the transaction code, but just provides a mechanism for the programmer to implement transactions.

Linda Systems

One classic shared data approach taken by Linda system and successors.

Immutable data approach typified by Linda-type systems, which use a *tuple space*. Tuples consist of a sequence of one or more typed data fields such as `<"fred", 1958>`, `<4, 3.2, "xyz">`.

Operations:

- read, take—block until a tuple is available that match tuple criteria such as `<String, 1958>`. *take* removes tuple from space.
- write—adds tuple to space

Example:

```
<s,count> = myTS.take(<"counter", integer>);
myTS.write(<"counter", count+1>
```

JavaSpaces

Shared data space.

Service that can be implemented in Jini. See Fig 12-15.

A Linda-like shared dataspace. Can be used for coordination of processes.

Tuples are typed references to Java objects (name, value pairs). Tuple contents are marshaled when stored into JavaSpace.

Can have multiple *instances* of a tuple. See Fig. 12-14. Can have multiple versions of a tuple.

Need a to supply a *template* when reading a tuple. Only instances matching the template are returned. Can have the client block until a matching tuple becomes available.

Can use events in combination with JavaSpaces.