Jini
by Joe Tavares

Introduction

With widespread use of
   Embedded processors
   System-on-a-chip technology
Devices are becoming smarter and smarter
In parallel networks becoming
   Faster,
   More widespread
   Beginning to infiltrate the home through
      Wireless connections,
      Cable
      Telephone
Common everyday appliances are becoming smarter
   Trashcan
      Keeps track of what you throw away
      Will print out a grocery list for you
We are entering a world where all devices will be smarter and networked
   Including consumer electronics and appliances

Problem
   Currently
      Configuring a
         Network
         Device to be on the network
   Too complicated for the average person
   Jini promises to change all that
      Making connecting to a network as easy as "flicking a light switch".

Jini Specification Overview

Architecture Overview
   Jini is a distributed system architecture
      Provides a reliable and scalable model for distributed computing
   Takes advantage of three existing technologies
      Java programming language
Remote Method Invocation
Object Serialization
All three technologies are fundamental to the Jini system
By using these underlying technologies
Jini introduces an infrastructure and programming model
To build a service based network system

Architecture
Jini infrastructure consists of
• Discover/Join protocol
• Lookup service

Lookup Service
Lookup service is heart of the Jini system
Contains a directory of services available within the federation
Clients can
Browse the lookup service
Query it in order to find a particular service
Generally clients do not know where the lookup service is beforehand
Must use a set of discovery protocols to find it
In addition services that wish to make themselves available to the federation
Must use the same discovery process to find the lookup service a
Then they must join

Discover and Join Protocol
There are 3 discovery protocols
• Multicast request
• Multicast announce
• Unicast discovery
These protocols work together
To allow services and clients to find lookup services
When a service finds a lookup service
It receives a service registration object
which it can use to add itself to the lookup service

Programming Model
The Jini programming model consists of
Interfaces which help to build reliable distributed systems
Three major components of the programming model
Leasing
Leasing functions very similarly to the leasing used in everyday life
Clients are granted a lease to resources
They have access to the resource for a fixed amount of time
At the end of the time period the resource is freed
Unless the lease has been renewed
Therefore, we avoid the common problems of starvation and deadlock
Which can result from partial failure of the system.

Transactions
Jini introduces a transaction mechanism
In true object oriented paradigm
Only the interfaces to the server are defined
Implementation details are left to the programmer
The transaction specification defines a two-phase commit protocol
Allows for the implementation of
ACID (Atomic, Consistency, Isolation, Durability) properties

Distributed Events
A distributed event model is supplied to
Allows event-model programming over the distributed system
Clients or third party objects can
Register interest in certain events
Receive notification of when they occur
Example an administrator can be notified
When a printer runs out of paper

Services
Services are the fundamental pieces of the Jini puzzle.
A service can be anything
It can be hardware or software
It can do anything on the system
All work is done through services
Notice

The lookup service is a service
The transaction interface is implemented through a service
Thus even the fundamental components are services
The service model takes Jini away from the monolithic programming model
Which is unwieldy due to large complexity and is fundamentally buggy
Jini services also separate the functionality
Into distinct parts with well-known interfaces

Foundations of Jini

At first glance Jini's reliance on Java may appear to be a limitation
Jini gains many advantages by actually leveraging Java's strengths

- Java is object-oriented, platform independent
- Supplies a security model (based on the applet sandbox).

The strengths of object-oriented programming has been touted for years
Java also achieves platform independence
By abstracting the results of compilation.
Instead of compiling code and getting a binary executable
Java compiles the code into a byte stream
The byte stream is then interpreted or "just in-time compiled"
By a Java Virtual Machine at run-time
Therefore, any platform that has a virtual machine can run the program
Jini uses the Java security model that is used for applets
This is a very restrictive security model
Only allows the program to do things
For which it has been granted explicit permissions

RMI
Remote Method Invocation (RMI) is fundamental tool
Used to distribute program execution
Supplies the stubs and skeletons for distributed interfaces
It takes care of the marshalling and unmarshalling of parameters
RMI greatly simplifies the construction of distributed applications
In its most basic form
RMI can be looked upon as Java's remote procedure call
Object serialization in combination with Java's platform independence
One of the primary advantages Java's RMI
Has over other object broker systems such as CORBA
Object serialization allows the serialization of complete object hierarchies
In other distributed architectures
Code can not be distributed over the network and
Only basic types can be sent
Therefore in order to pass objects
They must be deconstructed on the server
Then reconstructed on the client
Further those objects can only contain basic types and not other objects

Legacy Code and Java
It is important to address the legacy code issue
By moving to an exclusive Java environment
It may appear legacy code is lost
However, Jini can incorporate legacy code
By writing proxy-objects
Uses the Java Native Interface (JNI)
To communicate with the legacy code
Jini will work with the proxy-objects
Which will, in turn, call the appropriate legacy function
For more information on JNI consult the sun home page.

Discovery and Join
In a Jini federation there is no pre-defined central node
Instead, there is a lookup service
Which is a list of services and attributes
Clients can use to find the services offered by the djinn
The lookup service can be located anywhere in the djinn
Therefore, clients who wish to find a service
will first need to find a lookup service
Services wishing to add themselves to the djinn have it even harder
They not only have to find a lookup service
They then have to register with the service
To support registration
Jini introduces two protocols that run on top of existing network protocols
- Discovery
- Join

The discovery and join process is fundamental method
How a service discovers a Jini federation and joins it

Terminology
The discovery and join specification defines several terms
- A **host**
  A single hardware device
  Connected to one or more networks
  A host may have one or more JVMs (java virtual machines).
- A **discovering entity**
  One or more cooperating objects on the same host
  which are about to start or in the process of finding a lookup service
- A **joining entity**
  One or more cooperating objects
  That have a reference to a lookup service and
  Are in the process of obtaining references from or exporting references
to a lookup service.
- An **entity**
  May be either a
  Discovering entity
  Joining entity
  Entity that is already part of a djinn

Groups
It may be advantageous to partition services into logical groups
To see the advantages in having groups
Consider the UW
- If the entire university was one monolithic group
  - Then imagine trying to find a printer
  - There would be hundreds of printers available
    - You would have to scroll through to select the one that
1. you have permissions to use;
2. Is nearby.

On the other hand
If each research lab had a group
   It would be much easier to scroll through the printing services
   To find the printer one wishes to use

Groups in Jini can be based upon arbitrary criteria
   Examples would be geographic location, cost center or research lab.
Each lookup service has a set of 0 or more groups associated with it
When entities attempt to discover lookup services
   They announce what groups they are interested in
Only the lookup services associated with the appropriate groups will respond
Such a system allows the most flexibility
   Since a group's lookup service can move
   Discovering will still be able to locate it

Groups are just arbitrary text strings
   Sun recommends DNS style names to avoid naming conflicts
   For example ee.washington.edu
   If no group name is specified it defaults to the public group

Discovery
The goal of the discovery process is to find lookup services in the djinn
Depending on the environment and situation
   There are 3 different ways that an entity can find a lookup service
   1. Entity could explicitly search the djinn for lookup services.
   2. Entity could receive an announcement from a lookup service announcing its existence.
   3. Entity could register with one particular lookup service
In order to support these three different types of discovery
   Sun introduced three discovery protocols
   1. multicast request protocol
   2. multicast announcement protocol
   3. unicast discovery protocol
In all of the protocols discovery takes place between two entities
- The discovering entity
- The lookup service.

Multicast Request Protocol
This is the protocol used by discovering entities when they first connect to the djinn.
One can look at this protocol as a type of bootstrap protocol used by discovering entities.
Four components take place in the multicast request protocol.
Two run on the discovering entity and two on the lookup service.

On discovering entity
- Multicast request client performs multicast to discover lookup services.
- Multicast response server listens for responses from lookup services.

On lookup service
- Multicast request service listens for incoming multicast requests.
- Multicast response client answers request with information that allows discovering entity to access lookup service.

The steps taken by the discovering entity
Using the multicast request protocol are as follows:
1. Sets up a multicast request client.
2. Sets up a multicast response server.
   That is a TCP based socket that listens for incoming connections. Incoming connections will use unicast discovery.
3. Sets up an array of references to lookup services which will be filled as responses are received.
4. Sends a fixed number of multicast requests at periodic intervals.
   Each time it sends a response it appends the list of lookup services from which it has already heard.
   Sun recommends that a total of 7 requests be sent at 5 second intervals.

Steps taken by the lookup service
1. Listens for incoming multicast requests using multicast request service.
2. When it receives a request, the service checks two things.
a. First, it checks the list of lookup services that have already responded

b. Second, it checks to make sure that it is a member of one of the group that the multicast request desires.
   - If it has not already responded and it is a member of one or more groups it responds to the request.
   - If it needs to respond, it connects to the discovering entity using unicast discovery and passes its service registrar object.

---

**Figure 1 – A graphical representation of the multicast request protocol**

---

**Multicast Announcement Protocol**

The multicast announcement protocol is used

- Generally by a lookup service
- To announce its presence to the djinn.

In particular a lookup service will periodically send out an announcement

Primary purpose for this

- To avoid partial failure conditions
- Prevalent in distributed systems

Example

- If during the discovery phase the lookup service had
  - Crashed or been inaccessible
Due to network difficulties
A service would be unable to find the lookup service
However with the announcement protocol
Once the lookup service becomes accessible once again
Will announce its service
Client can then register with the lookup service

Steps taken by lookup service:
1. Sets up datagram socket to send to well-known port where the multicast announcement messages are to be sent
2. Listens for connections using unicast discovery
3. Sends out announcements which include service ID, connection information and group information at periodic intervals.
   Sun recommends every 2 minutes.

Steps taken by service receiving announcement:
1. Listens for incoming announcement on pre-defined port
2. For each announcement it receives
   a. It checks to see if it has already heard from the lookup service
      By checking the announcer’s service ID
      Against the array of service ID’s that it has already heard from
   b. If it has not heard from the lookup service
      It checks to see if the lookup service belongs to a group that it is interested in.
If either of the above two checks fail
Service ignores the announcement
If they both succeed
1. It uses unicast discovery to obtain a reference to the lookup service
   Using the information transmitted in the announcement
2. It adds the lookup service to the array of services it has already heard from

Unicast Discovery Protocol
The multicast protocols work well for LANs but not for WANs
Most routers will not route broadcast packets
Therefore unicast is used to make connections to the outside world
Unicast uses reliable protocols (TCP) to establish connections to known lookup services
Note
Unlike the multicast protocols
Unicast will not find lookup services
The connection information must be known
The unicast discovery protocol is a simple request-response protocol.

Join
Most of the procedure to join a lookup service is the discovery
As discussed above
To join the service
1. Sets up a set of attributes
2. Discovers lookup services
3. Registers with the lookup service
The service then periodically needs to renew its lease with the lookup service

Leasing
In non-distributed systems
A common model for resource allocation is to allocate them
Until they are explicitly freed by the application
A good example of this is memory allocation
Where one must explicitly allocate and free it
This works for non-distributed applications
Since the environment is fairly reliable
However this model does not work well for distributed systems
• Primarily because of the high potential for partial failures
• Because of the long-life of most distributed systems.

Partial failures are a characteristic of distributed systems
Any of the several distinct parts of a distributed system could fail
Leaving the other components hanging
Examples of reasons for partial failure are
• Network errors
• System crashes
If resources had to be explicitly freed
A client could be given access to a resource
Then a partial failure could occur
Causing the resource to needlessly continue being allocated to the cut-off client.

Another problem with the explicit freeing of resources in distributed systems
These systems tend to be long-lived
The long life can cause an accumulation of
Outdated and unwanted references to resources
Such accumulations can be due to a wide variety of reasons
- System crashes
- Periodic maintenance of resources
- Partial failures
- Programmer's forgetfulness

Jini uses the concept of leasing to address these problems
Leasing in Jini is similar to the lease used today in everyday life
A client is able to lease a service for a negotiated amount of time
Note that leasing is meant to supplement, not replace
Other time-based tools such as
  Time-to-live and ping.
As the Leasing specification states,
  It just adds another tool to the programmer's arsenal.

The lease is granted for a time interval
Not until an absolute time
  for 30 minutes and not until 3:00
This is done to avoid the problem of clock synchronization across multiple systems.
However it does mean that the two parties must be aware of the latency in communications
Different calls may take different time
  Based on outside factors such as network traffic
Example of when the latency maybe a problem is during renewal
  If the granter attempts to renew too late
  The notice of renewal may return
  After the lease has expired and the references freed

It is important to clarify that the resource does not necessarily have to be a limited one
It may be a resource which

- An unlimited number of clients can use or
- It may be a resource that only one client at a time can use

The lease can also be used by the granter to supply some level of synchronization.

Leasing is part of the Jini programming model
That aims to build reliable distributed systems
Which define predictable responses to

- Failures
- Forgetfulness
- Disinterest

In a way that does not allow unbounded resource consumption

**Lease Characteristics**

1) Lease is a time period in which the granter guarantees the availability of the resource to the holder.

2) Leases can be explicitly canceled by the holder.

3) The lease holder can attempt to renew the lease for the same, for a shorter or longer period of time before lease termination.

   The granter can either grant, reject or grant for a period of time less than or equal to the amount of time requested.

4) If no action is taken the lease expires and both the granter and the holder are expected to free their references and clean up appropriately.

**Lookup**

The purpose of a lookup service is to act as a central repository for services

*Services* - can register their service with the lookup service

   To make it available to potential clients

*Clients* - can connect to the lookup service

   To find the services they need to do their work.

The lookup service maintains a flat collection of service items

There is no hierarchy maintained within it

   Although one could impose hierarchical views upon the items

Within the lookup service

The lookup service can be viewed as a directory of services

Each service item consists of
A service ID
An RMI stub (if the item is implemented as a remote object)
A list of sets of attributes.
Below is an example of what a lookup service may look like

<table>
<thead>
<tr>
<th>Service ID</th>
<th>RMI Stub</th>
<th>Sets of Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>3729817439874398</td>
<td>PrinterService</td>
<td>name, set print properties</td>
</tr>
<tr>
<td>3328490280942934</td>
<td>LookupService</td>
<td>name, manufacturer</td>
</tr>
<tr>
<td>3489798732498273</td>
<td>TransactionService</td>
<td>name, version</td>
</tr>
</tbody>
</table>

Table 1 - A sample snapshot of a lookup service

An attribute set is an instance of an object
Each attribute is a public interface to that class
As stated earlier a service may have many entries with different sets of attributes

**Transaction Interface**

A transaction is a grouping of operations
Whose nature dictates that they should either all succeed or all fail
Further, the operations within a transaction
Should appear to the outside as if they occur simultaneously
In distributed systems operations are particularly error prone
They may depend on several different remote participants
Therefore a transaction system must to maintain the integrity of the data

Traditional transaction systems control every aspect of the transaction
They implement a maximal set of interfaces
That ensures correctness of all possible transaction semantics
Jini chooses to implement a minimal set of interfaces
That allow services to implement transaction semantics
The Jini system designers call this
"philosophical" difference.
They choose a more flexible, more traditional object-oriented approach.

What Jini does define is the completion protocol
The completion protocol is *two-phase commit protocol* for distributed transactions
The two-phase commit protocols defines mechanisms
Which allow operations to be wrapped into a single operation
A transaction is created and overseen by a transaction manager. It is given a unique transaction ID during creation. The semantics of the transaction are represented by a semantic object.

Note
Although the transaction manager only cares about completion of the transaction, participants need to know the semantics of the transaction.

Therefore, clients and services usually create, pass, and manipulate semantic objects. Services which usually implement transactions accept semantic objects of certain types.

Typical steps in a transaction:

1) Client creates a transaction by making a call to the transaction manager (usually by calling TransactionFactory to create a semantic object).
2) The newly created semantic object is passed to the service as a parameter.
3) If the service accepts the transaction, it joins it as a participant.
4) All operations are then performed under the transaction.
5) To commit a transaction all the participants must vote. Participants can vote prepared, not-changed, or aborted.
6) If all vote prepared or not-changed the transaction is committed and all participants are told to roll-forward.
7) If one or more participants vote to abort, then participants told to roll-back.

The following diagrams are from the Transaction Specification help to show the mechanics of a transaction.

From the perspective of:
- The client (originator)
- A participant
- The service manager
Distributed Event Model

Event based programming is very common
Especially in window based systems
Microsoft Windows
MacOS
X-Windows
The same model of programming is useful in a distributed environment
Where objects in a Java Virtual Machine (JVM)
Can register interest in
- Occurrence of some state change (event)
- In another object
Running on a different JVM

- Can receive notification of when that event takes place.

The distributed event model has different characteristics and requirements from the local model.

In a local model:

- Events occur as a result of:
  - Mouse clicks, key presses, etc.
- This is made possible because notification is assumed to be:
  - Ordered, timely and reliable

In a distributed system these assumptions do not hold:

- Notification is:
  - Untimely, unreliable and can occur in any order

Therefore, the nature of events differs between:

- Local and distributed systems

The types of events that are useful in a distributed system would be:

- A printer out of paper
- Disk full
- A bug has been entered into the database

Furthermore it is often useful to have a third party object:

- Receive the notification:
  - Instead of the object that originally registered interest receiving it

This allows objects to:

- Offload some of its load
- Only retrieve the notifications when it is ready:
  - Think of a logging system where the events are:
    - Logged to a third party
    - Only retrieved when explicitly requested