Jini by Example - Whitepaper

Technical Expertise Level: Intermediate
Requires knowledge of: Java

May 1, 1999

Download this document
- MS Word format [95k Zipped]
- Adobe Acrobat PDF format [109k Zipped]

Index
- Jini Overview
- Discovering a lookup service
- Unicast Discovery
- Multicast Discovery
- Joining the Lookup Service
- Joining through the JoinManager utility class
- A closer look at the example
- Joining through the ServiceRegistrar’s register() method
- Managing the lease time for services/events
- Looking up and accessing a Jini Service
- A Client for the calculator service
- A clean and simple approach for running/accessing Jini services in your network.
- Compilation of our Service and Client classes and Grouping into Jar files
- Starting the necessary services
- Possible problems you may run into

Jini Overview
The Jini technology makes a network more dynamic by allowing the plug-and-play of devices. It provides mechanisms for devices to join and detach from network dynamically without the need for configuring each device.

Fig 1. Jini as top of Java runs on any OS

The central mechanism of a Jini system is the Lookup service that registers devices and services available on the network. It is also the major point of contact between the system and the users of the system.

When a device is plugged into the network, it locates the lookup service (discovery) and registers (join) its service there. While registering, the device provides a callable interface to access its functionality and attributes those may be useful while querying the
lookup from another Jini client perspective. Now, the service offered by the device may be located and used by clients on the network. This basically involves discovering lookup, querying it for the specific service and invoking the callable interface of the service required. The callable interfaces are exposed and accessed through Java RMI (Remote Method Invocation). The proxy code that is used to access an interface is stored within the lookup services during registration and downloaded by clients automatically. Thus the need for specialized drivers is eliminated.

Thus Jini allows building up clusters of services that know about one another and cooperate - creating a "federation" of devices. The term federation may be understood to be a collection of co-operating, but autonomous entities. However, with Jini, Java on the devices is almost mandatory, and the interaction mechanisms - the interfaces, the methods, what to call, when to call must be well defined for devices to join a federation that you create. In the next few sections, we try to explain Jini from the perspective of implementing a Calculator service that a Jini federation makes use of.

Discovering a lookup service

Our calculator service will first have to register itself as a JINI service. To accomplish this, the calculator should discover a lookup service and register it. There are two ways of discovery, unicast discovery and multicast discovery.

Unicast Discovery

For unicast discovery, the device must know the IP address and the port number where the lookup service is running. The device sends a unicast discovery protocol packet to the pre-known IP and port. In response, the lookup service will send a unicast announcement packet to the device after the discovery packet is recognized and accepted. The Jini Framework uses the net.jini.core.discovery.LookupLocator class to specify the pre-known IP and Port using a string called Jini URL.

The Jini URL format is jini://MachineIP:PORTNO. After creating a LookupLocator with required Jini URL, we just need to call its getRegistrar() function which give you a lookup service object representation known as a ServiceRegistrar. The rest of the operations on the lookup service can be done through the ServiceRegistrar object. All the Unicast discovery protocol fundamentals are encapsulated within the LookupLocator class, so we no need to worry at all.

The code fragment for getting the ServiceRegistrar object is given below. The LookupLocator object is created based on the Jini URL string passed. If the Jini URL string does not have a port number mentioned, a default port number of 4160 is assumed.

```java
// Instantiate a LookupLocator object for the
// lookup service indicated
// Uses the IP and port number passed in.
// (If no port number is specified,
// the default port of 4160 will be used.)
```
LookupLocator lookupLocator = new LookupLocator(JiniURL);
// Invoke getRegistrar() on the LookupLocator to perform a
// unicast discovery on the lookup service. The lookup service
// is indicated by the host and port number passed
// to the constructor.
// The result is a reference to the
// registrar object representing the lookup service.
ServiceRegistrar registrar = lookupLocator.getRegistrar();

The complete code for unicast discovery can be found in
UnicastDiscovery. It should be invoked with the Jini URL string
specific to your machine address and Jini port.

**Multicast Discovery**

Multicast discovery is used when the locations of Jini lookup services
are not known beforehand. The network must be multicast enabled by
configuring the router settings to locate Jini lookup services outside of
your network domain.

Multicast is something like a controlled broadcast; the multicast
discovery packets can cross gateways and reach designated multicast
"groups" of machines. Setting a TTL (Time-to-Live) parameter can
control the range of multicast discovery packets. This TTL parameter
specifies the number of gateways the packets may cross. By default
the TTL value is 15. By sending out multicast discovery packets over a
designated network range, a device can detect Jini lookup services
running within that range, within specific multicast groups or even all
groups if necessary. Once the discovery packets are detected, the
lookup services will send multicast announcement packets into the
network, which might reach your application that acts as
DiscoveryListener.

A DiscoveryEvent will be sent to the DiscoveryListener, which in turn
provides a set of ServiceRegistrar object arrays. Note that the
device may get responses from more than one Jini lookup service. It
is upto the device to decide when and where to register itself.

The code for multicast discovery can be found as LookupServiceFind
and it can be used readily by any service providers and service
requesters.

As can be seen in the program for multicast discovery, first it creates
a LookupDiscovery object and a DiscoveryListener implementation
object. Then, addDiscoveryListener method is invoked to make the
DiscoveryListener listen for announcement packets from the lookup
services. LookupDiscovery, on receiving the announcement packets,
calls the discovered method of appropriate DiscoveryListener. We,
here, are not bothered much about the discarded method, which will
be called when a lookup service go out of the network.

**Joining the Lookup Service**

After locating the lookup service, the calculator service would need to
join the lookup service. Join occurs when a service has located a
lookup service and wishes to contribute itself. The device first locates
the Jini lookup service under a specific group and then registers its
service in it by loading its service object. A group identifies and/or
organizes the set of services available under a category. For example,
printer groups registers print services. The default group is public.
However, the present Jini implementation has the drawback that it
locates all lookup services pertaining to the specific service, not just
public alone. If we request for a specific service, both specific and
public lookup services will be found.

Some important classes relevant in our context of join protocol are
discussed below. For further information on Join protocol, please refer
the Jini API documentation.
net.jini.core.lookup.ServiceItem:
This is an encapsulation for any service objects the device wants to provide. The device can provide N number of services, as it needs. Registration/Lookup is possible only in terms of a ServiceItem when it is via ServiceRegistrar.

net.jini.core.entry.Entry:
As far as jini is concerned, a device may register its service along with a set of objects, which would best describe the device characteristics. Those are called Device Attributes. Those attributes have to signal to the LookupService that they are characteristics of the registering device by implementing the net.jini.core.entry.Entry, interface. For example, a set of attributes may describe the type of the service, service location, service provider, service usage, etc. For simplicity Jini provides an abstract class known as net.jini.entry.AbstactEntry which in turn implements the Entry interface and provides some meaningful functions like equals(), toString() and so on. Just extend it and add up device specific attributes - fields. A small example for defining Attribute goes here.

```java
public class DiskAttr extends net.jini.lookup.AbstractEntry {
    // An integer value representing the disk capacity
    public Integer capacity = null;
    // A string representing the product brand name
    public String brandName = null;
    // A boolean representing the current state of
    // the disk locked/unlocked
    public Boolean status = null;
}
```

* Please take a careful look at the above code snippet

As you can find all member fields in the above class are public. JINI stores your service objects along with any attribute fields using serialization. This implies that any attribute fields you want serialized will be declared as public for direct access - not static, final or transient. More over we can't have any primitive data types as fields of attributes, that's why the first and third fields in the previous example are wrapped around the equivalent Wrapper classes.

net.jini.core.lookup.ServiceRegistration
For any service registration with jini, the lookup service returns a ServiceRegistration object, which provides lot of useful facilities and information about our registration.

1. A method to get the related Lease object, with which a service can do a lease renewal and lease cancellation easily.
2. A method to get the allotted ServiceID, which uniquely identifies any service on the lookup service. This ID makes the maintainability easy for our service. More over this ID confirms that there are no redundant service in the lookup service, which is improves the jini system integrity.
3. A set of methods to add/modify/replace with new bunch of attributes.

net.jini.core.lookup.ServiceTemplate
This is a template class with which we can frame up a jini service search criterion to find matching services available in the lookup services. The below diagram illustrates the various parts of a ServiceTemplate.
There are primarily two ways of joining a lookup service. One is by using the JoinManager utility class in com.sun.jini.lookup package, which comes in handy. The other method is by enumerating services on the network explicitly and registering with them. Before we take a look at both, we need to note one more point about service implementation and uploading.

There are various possible methods to make our service available in the Jini lookup service. When you implement your service interface without RMI compliance, the server side service application has to be kept alive. So the client can’t be sure about it. On the other hand, when you create your interface as a Remote interface your server side application creates a stub class for your implementation class via the RMIC tool and registers either with Rmiregistry or Activation daemon. After this, your server application can simply quit after registering a stub with RMI. In case you decide to register your stub with Rmiregistry, you will need to extend your implementation class with java.rmi.UnicastRemoteObject, otherwise extend your implementation class with java.rmi.activation.Activatable. For further references on this issue, please go through the RMI specification in your JDK1.2 docs.

For simplicity, we make use of UnicastRemoteObject model in our examples.

**Joining through the JoinManager utility class**

JoinManager is a utility class, which discovers and keeps track of which lookup services to join, registers with them, keeps the registration leases renewed, and keeps the attributes up to date.

There are a number of possible constructors for JoinManager. The simplest and first is:

```java
JoinManager(java.lang.Object obj,
            Entry[] attrSets,
            ServiceIDListener callback,
            LeaseRenewalManager leaseMgr)
```

This specifies the service to be managed and its entry attributes. The callback will have its serviceIDNotify() method called when a new service is discovered. This argument can be null if the application has no interest in this. The leaseMgr can also be set to null and will then be created as needed later. Using this constructor will initiate a search for service locators belonging to the group "public", which is defined by a group value of the empty string "". The applications will need to follow this up immediately with a call to search for locators belonging to any group.

```java
JoinManager joinMgr =
new JoinManager(obj, null, null, null);
joinMgr.setGroups(LookupDiscovery.ALL_GROUPS);
```

The second constructor is:
JoinManager (java.lang.Object serviceobject,
       Entry[] attrSets,
       java.lang.String[] groups,
       LookupLocator [] locators,
       ServiceIDListener callback,
       LeaseRenewalManager leaseMgr)

The constructor includes multicast group names and lookup locators.
This allows multicast search for locators belonging to certain groups,
and also unicast lookup for known locators. A multicast only search
for any groups would use

JoinManager joinMgr = new JoinManager
       (serviceobject, null,
       LookupDiscovery.ALL_GROUPS,
       null, null, null);

Where both additional parameters are set to null.

JoinManager(net.jini.core.lookup.ServiceID ServiceID,
       java.lang.Object obj,
       net.jini.core.entry.Entry[] attrSets,
       java.lang.String[] groups,
       net.jini.core.discovery.
       LookupLocator[] locators,
       LeaseRenewalManager leaseMgr)

This constructor differs from the previous one in specifying a
ServiceID object. We can use this constructor when we already have
registered a service and want to update the service. Selection of any
one the JoinManager constructor depends on the situation and
requirement.

A closer look at the example
Lets take a closer look at implementing our calculator service and
registering its service on the network. The steps involved are a)
define a Calculator interface b) Create the GUI Frame class for it c)
Create the object that does the actual service d) Use the JoinManager
for easy registration.

Here is our Calculator interface code. Note that Calculator extends
java.rmi.Remote interface in order to make it available to the clients
via RMI protocol.

- an interface Calculator

    /**
     * This is the contract interface, which is
     * being provided
     * by the CalculatorService class
     **/
    import java.awt.Frame;
    import java.rmi.*;
    public interface Calculator
    extends Remote {
    public Frame getCalculator()
    throws RemoteException;
    }

- A Calculator Frame class, which is the calculator GUI - CalcFrame
  This CalcFrame is the service provided by the Calculator Service.
  Whenever any Client asks for a service of type Calculator, an
  instance of the Calculator frame is sent from the Lookup Service
to the client application.
- An implementation of the Calculator interface as well as of the
  Jini lookup interaction is provided by the class CalculatorService
The function `getCalculator()` is implemented as follows.

```java
public Frame getCalculator()
throws RemoteException {
    Frame f = new CalcFrame();
    f.setSize(300,150);
    return f;
}
```

The `CalculatorService` is registered with the lookup service as follows. In the following code segment we make use the first JoinManager constructor we discussed above.

```java
CalculatorService calculatorService =
    new CalculatorService();
String hostName = "chittu.cswl.com";
    Naming.rebind("rmi://"+hostName+"/CalculatorService",
    calculatorService);
System.out.println("Bound with RMIRegistry");
Object serviceStub = Naming.lookup
("rmi://"+hostName+"/CalculatorService");
System.out.println("Trying to Join Manager");

// Join the manager so that it
// will become a part of federation
// JoinManager joinManager =
//    new JoinManager(serviceStub,
//    attributes,calculatorService, new LeaseRenewalManager());
```

After this step the `CalculatorService` object is available to all Jini clients on the network. In the later part of the tutorial we shall see how to gain access to this `CalculatorService` as a Jini client.

Simply passing a `LeaseRenewalManager` object to the JoinManager constructor doesn't confirm the renewal of Lease for our service. We also need to make our application alive till we want the renewal.

How can I take control over the `LeaseRenewalManager`? For sake of simplicity we took the null argument constructor for `LeaseRenewalManager`. There is one other form of the constructor that helps in various ways.

```java
LeaseRenewalManager ( net.jini.core.lease.Lease lease,
    long expiration_in_milli_seconds,
    LeaseListener listener)
```

Here the first two parameters are self-explanatory except the `LeaseListener` parameter. `LeaseListener` is a listener object, which has a callback function `notify()` which is called with a `LeaserenewalEvent` when the related Lease expires or Lease is denied. We can create and keep alive a separate thread for the `LeaseListener` purpose and take some corrective measures when the Lease fails. In real time Jini programming this would be a very important consideration.

### Joining through the `ServiceRegistrar`'s `register()` method

Note: Here, we implement the same calculator service by the name AnotherCalculatorService, as the registration approach is different.

The approach of the join method here is quite different from that used in the Join Manager case. In this case we explicitly enumerate lookup services on the network, create a `ServiceItem` that encapsulates our service object and a set of attributes, and register this with each lookup service we are interested in. Please refer to
AnotherCalculatorService.java for coding details.

Lookup services on the network can be discovered using the LookupServiceFind class as discussed in the discovery section. Simply call its getLocators() method to get return an array of LookupLocator objects, each representing a lookup service on the network.

```
LookupLocator[] lookupLocators =
    LookupServiceFind.getLocators();
// Attributes for the service is
// initialized as follows.
// You can use attributes like ServiceType,
// ServiceInfo, Location etc. as desired
Name name = new Name("AnotherCalculatorService");
// e.g. a Name attribute of the Service
Entry [] attribs = new Entry[1];
attribs[0] = name;
// make the Name attribute as part of
// the attribute set

Note: The attribute set facilitates Service
query/Lookup simple and appropriate from a
client point of view.

// A Service item is an encapsulation
// of the actual service object and a set
// of attributes which describe the service.

AnotherCalculatorService calculatorService =
    new AnotherCalculatorService();
String hostName = "chittu.cswl.com";
Naming.rebind
("rmi://"+hostName+"/AnotherCalculatorService",
calculatorService);
System.out.println("Bound with RMIRegistry");
Object serviceStub =
    Naming.lookup
("rmi://"+hostName+"/CalculatorService");
ServiceItem serviceItem =
    new ServiceItem(null, serviceStub,attribs);

for(int i=0;i<lookupLocators.length;i++)
{
    ServiceRegistrar registrar =
        lookupLocators[i].getRegistrar();
    ServiceRegistration registration =
        registrar.register(serviceItem , Lease.ANY);
    Lease lease = registration.getLease();
    System.out.println("The allotted lease time is :");
    + lease.getExpiration()+" milli secs");
    System.out.println("The allotted Service ID is :";
    + registration.getServiceID());
}
When you register a service with a lookup service, the registration will be available for a certain period only, known as the lease (a returned object from Jini lookup) period. It is a must to renew the lease before it expires. Otherwise the service will be garbage collected automatically. It is also possible to cancel a lease following which the service will be removed from the lookup service. Granting a lease and its duration is fully up to the Lease granter to decide upon. In the present implementation of Jini, when you start a lookup service using `com.sun.jini.tck.reggie.CreateLookup` class, the service object will be available for only 5 minutes. But this can be overridden as explained in the next session.

### Managing the lease time for services/events

At some times, when you deal with Jini application development, it becomes important to administer the Lease time for services and events to be registered. If you take a look at the source code of the `Reggie` service implementation, `com.sun.jini.tck.reggie.RegistrarImpl`, which implements the `RegistrarAdmin` interface, you will find that the lease time for services has been hard coded to 5 minutes. It may be worthwhile to change the default(hard-coded) values. However, the interface does provide methods to change the lease time. The only thing is how to obtain the interface? A code snippet is shown below, which gives you the remedy. The complete listing can be found in `LeaseTimeChanger.java`.

```java
try{
    LookupLocator locator =  new LookupLocator("jini://"+args[0] );
    ServiceRegistrar registrar =
        locator.getRegistrar();
    // Changing the lease time - use RegistrarAdmin
    // Your registrar object is an implementation of
    // Administrable interface
    // So just cast to it.
    net.jini.admin.Administrable admin =
        (net.jini.admin.Administrable)registrar;
    com.sun.jini.tck.reggie.RegistrarAdmin adminProxy = null;
    // Get the Registrar administrator object by
    // calling the getAdmin() method
    // Be careful of your RegistrarAdmin import.
    // Make sure you use com.sun.jini.tck.reggie.CreateLookup
    // for starting the lookup service
    adminProxy = (com.sun.jini.tck.reggie.RegistrarAdmin)
        admin.getAdmin();
    // setMinMaxServiceLease() will set the lease time for services
```
long serviceLease = Long.parseLong(args[1]);
long eventLease = Long.parseLong(args[2]);

adminProxy.setMinMaxServiceLease(serviceLease);
// setMinMaxServiceLease() will set the lease
// time for event registrations
// if you don't understand what is Event Registrations,
// no problem - it wont affect your
//understanding of the tutorial.
adminProxy.setMinMaxEventLease(eventLease);
}
catch(Exception ie){
// You may get a ClassCastException if the
// RegistrarImpl is not from the package
// com.sun.jini.tck.reggie.
System.out.println("*****Class Cast Exception*****");
//in case if you are using RegistrarImpl.class
//from com.sun.jini.reggie package
//then you need to replace com.sun.jini.tck.reggie
//with com.sun.jini.reggie
//after you change, recompile it , run it.
}

Looking up and accessing a Jini Service

Looking up involves querying Jini lookup services using a service template. This service template, represented by a ServiceTemplate object, can specify fully or partially an attribute set. This is matched against the attributes of registered services and any matching ServiceItem objects are returned. It is possible to ask for a single matching service or N matching services, as you would choose. The following calls respectively can be used:

- serviceRegistrar.lookup(net.jini.core.lookup.ServiceTemplate templ)
  Returns java.lang.Object, cast the result to the desired & appropriate object type
- serviceRegistrar.lookup (net.jini.core.lookup.ServiceTemplate templ, int nMatches )
  Returns a ServiceMatches object from which ServiceItems can be extracted

A Client for the calculator service

Here we finally create a client class to test and access the JINI service that we have developed and registered with Jini.

// Specify the name of the service you need to search for
attributes[0] = new Name("CalculatorService");

// Creating ServiceTemplate with the given attributes
ServiceTemplate template = new ServiceTemplate(null, null, attributes);

// Perform a discovery to locate a lookup service and
// get its ServiceRegistrar Object. See code for details
// Execute a lookup based on the template
// Here, although I search by a Name attribute, I’m assuming that
// the object that comes back implements the Calculator interface
calculator = (Calculator) registrar.lookup(template);

Now we have access to the Calculator interface. Using this interface,
the Calculator GUI is obtained by invoking the function
getCalculator().

Frame f = calculator.getCalculator();
f.setVisible(true);

Fig 6 A Jini Client invoking service on the service provider machine

To summarize what we have done so far:

1. We located a Lookup Service on the Network
2. Registered a Service to it (CalculatorService and
AnotherCalculatorService)
3. Obtained the Service (CalculatorService) from the Lookup
Service
4. Used the RegistrarAdmin for setting the lease time

A clean and simple approach for running/accessing Jini
services in your network.

First you will need to have downloaded the software from SUN site
http://www.sun.com/jini After which, you will have the following:

- The Jini™ Technology Core Platform (JCP) -- provides the core
  Jini technology. It includes the "core" specifications and
  corresponding interfaces and classes for: Entry; Lookup;
  Discovery and Join; Distributed Events; Leasing; and
  Transactions.

- The Jini™ Technology Extended Platform (JXP) -- provides
  standard Jini software extensions, utility specifications and
  software that are useful in writing Jini applications and services.

- The Jini™ Software Kit (JSK) -- provides implementations of the
  Lookup and Transaction Manager services specified in the Jini
  Technology Core Platform (JCP) and the Jini Technology Extended
  Platform (JXP).

- In case you want to use JavaSpace technology, then you will
  need to download JSTK1.0DC
- The Jini Technology Compatible Kit (Jini TCK) also provides
  implementations of the Lookup service specified in the Jini
  Technology Core Platform (JCP) and the Jini Technology Extended
  Platform (JXP). Since this contains various utilities for testing our
  jini environment and developing Jini classes, we have used this
  particular kit in our example. In case if your application involves
  Transaction management, then it becomes must to use JSK jars
  instead of TCK jars.

Make sure that all your binary downloaded SDKs have a /lib
subdirectory from the installation subdirectory. This lib directory
contains the necessary jar files. For your convenience we have
provided the jar file table.

<table>
<thead>
<tr>
<th>Server Side</th>
<th>Client Side</th>
<th>Description</th>
</tr>
</thead>
</table>
### Jini Technology - Network-Centric Services: Java Solutions & Java Proc...

<table>
<thead>
<tr>
<th>JAR File</th>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jini.jar</td>
<td>None</td>
<td>Core Jini technology interfaces and classes</td>
</tr>
<tr>
<td>Reggie.jar</td>
<td>reggie-dl.jar</td>
<td>Jini Lookup service implementation</td>
</tr>
<tr>
<td>Mahalo.jar</td>
<td>mahalo-dl.jar</td>
<td>Jini Transaction implementation</td>
</tr>
<tr>
<td>Outrigger.jar,</td>
<td>outrigger-dl.jar</td>
<td>JavaSpaces technology implementation</td>
</tr>
<tr>
<td>transient-outrigger.jar</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jini-examples.jar</td>
<td>jini-examples-dl.jar</td>
<td>Lookup Browser and Service GUI examples</td>
</tr>
<tr>
<td>space-examples.jar</td>
<td>space-examples-dl.jar</td>
<td>JavaSpaces technology examples</td>
</tr>
<tr>
<td>tools.jar</td>
<td>None</td>
<td>ClassServer, DependencyChecker, SerializationChecker</td>
</tr>
<tr>
<td>sun-util.jar</td>
<td>None</td>
<td>Development utilities</td>
</tr>
<tr>
<td>pro.zip</td>
<td>None</td>
<td>ODI classes needed by Outrigger</td>
</tr>
</tbody>
</table>

When you develop your classes and starting services, it may be bit tedious to specify the necessary jar files in the `-classpath` option when you compile and `-cp` option when you run. So you can simply extract all the class files from Jini jars into a specific dir. Afterwards you can simply specify the absolute dir name in the classpath variable. For extreme simplicity, you move/copy the jini jars from JCP, JSK and JiniTCK into one directory, let us say **E:\jinijars\** and extract all the classes into one directory, let us say **E:\jiniclasses\**. The above table does not list the jars from JiniTCK. Those are

<table>
<thead>
<tr>
<th>Server Side</th>
<th>Client Side</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tck.jar</td>
<td>Tck-dl.jar</td>
<td>Jini TCK utilities and extension</td>
</tr>
<tr>
<td>TckReggie.jar</td>
<td>TckReggie-dl.jar</td>
<td>Jini Lookup service implementation</td>
</tr>
</tbody>
</table>

### Compilation of our Service and Client classes and Grouping into Jar files

1. Set the classpath to include our jini class path. In our example it is **E:\jiniclasses**. You don't need to include the Java core class path i.e. `/jdk1.2/jre/lib/rt.jar`, which will automatically loaded into the JVM when it is started.
2. Compile all the Java files using javac `-d` output-dir `filename.java` pattern

   **Example:** `javac -d E:\jiniclasses CalculatorService.java`

   Note :- The Java files in our tutorial are contained in a package **(package calc)**; So the javac tool will create a directory called `calc` under which all the compiled classes will be stored. So, in the tutorial all of our classes starts with an annotation `calc.` Example the CalculatorService class will be `calc.CalculatorService`

3. Compile the `calc.CalculatorService` and `calc.AnotherCalculatorService` classes using `rmic` tool to
create the stub classes for the clients to contact the remote server implementation.

Example: `rmic -v1.2 -d E:\jiniclasses calc.CalculatorService` which creates `calc.CalculatorService_Stub.class`. The `-v1.2` option in the `rmic` command eliminates the creation of Skeleton class, which are no longer needed in Java 1.2 platforms.

Now you should have the following classes under `calc` sub directory:

- `Calculator.class`
- `ClientFrame.class`
- `CalculatorService.class`
- `AnotherCalculatorService.class`
- `CalculatorServiceStub.class`
- `AnotherCalculatorServiceStub.class`
- `CalculatorClient.class`
- `AnotherCalculatorClient.class`

After we create all the necessary classes, we need to create two jar files:

- One consists of Server side classes (Red colored ones) name it as `CalcServer.jar`
- One consists of Client side classes (Blue colored ones) name it as `CalcClient.jar`

Is it a must to group the classes into jar files? May be a valid question. But we shouldn't forget the important truth about Jini and RMI, i.e. Jini supports dynamic downloading of Stub and required classes from the Server side to the Client. However, Dynamic downloading is not possible for the remote interfaces as `RMI supports implementation binding, not interface binding`. This dynamic downloading has to be taken care of through an HTTP server. Since the HTTP server will provide the classes from within a jar file, it becomes must to put the Server classes into a jar file and provide the jar file path to the HTTP server.

Note: The (Pink colored) `Calculator.class` should be part of both the Server and Client. This is the service agreement/interface between the Jini Server and the Jini Client as we discussed earlier in the tutorial.

Now we should proceed to the important aspect of the tutorial - how to run our services successfully. Once again, for better clarity let's check that the Jini jar files are available in `E:\jinijars` and jini classes has been extracted into `E:\jiniclasses`.

**Starting the necessary services**

1. Start an HTTP server. You can use any web server that can send documents. For an example it may be Microsoft Peer Web Server. For your convenience JSK comes with a ClassServer class which has been put in the `tools.jar` file. You can use this as follows

   ```
   java -jar e:\jini_0\lib\tools.jar -port 8000 -trees -verbose
   ```

   `-trees` causes the HTTP server to get the class file from any of the Jar files available in the document root : in this case it is `e:\jinijars`

   `-verbose` makes the HTTP server to display diagnoses information on the standard output.

   `-port` specifies the HTTPd port. By default it is 8080.

   It is possible to change the directory content of the document root even after starting the HTTP server. When you do this, you do not need to stop and re-start your HTTP server every time you want to change the directory content.
update your Jar file. However, make sure that the Server side jar is also part of document root you have specified in the command line for starting HTTP server.

2. Start RMID. You will need to remove the directory named /log if such a directory exists. Otherwise the RMID will try to reconstruct the Jini Lookup Service and Service objects you registered with the previous instance of the lookup service.

3. Start RMIREGISTRY. Make sure that the CLASSPATH variable doesn't include the path where you have stored your packages. This 3rd step could be done before the 1st and 2nd steps also.

4. Start the Jini Lookup Service. When you start the lookup service, make sure that you are specify a -Djava.server.rmi.codebase with the right URL.

A point about codebase property

- In case you have used the ClassServer (we have discussed earlier in this section) for creating your HTTP web server, then what you need to specify is:
  -Djava.server.rmi.codebase=http://host_address:port
  where the host_address is your IP and the PORT is the PORT no either the default 8080 or the number you specified when you started your web server.
- If you use a web server like Microsoft Peer Web Server, then you need to create a virtual directory in your HTTP service (Note the PORT number – default is 80) and the property has to be given like this.
  -Djava.server.rmi.codebase=http://host_address:port//virtual_dir_alise_name/

5. Go to the Server specific directory. Start your Server application, which does the following steps for you,
   - Create a service object and bind it with rmiregistry.
   - Obtain the service object from rmiregistry (get the stub).
   - Find a specific lookup service in the network and register your service stub with that.
   - Do a lease renewal through LeaseRenewalManager till you close your service application

6. Go to the Client specific directory i.e. where you have extracted your calcclient.jar and start your client application.

We have provided a set of batch files (a text version is here) in the downloadable, which serves the purpose of starting the necessary services.

start_http.bat

start java -jar e:\jini1_0\lib\tools.jar -port 8000 -dir e:\jinijars\ -trees -verbose

start_lookup.bat


start_server.bat

java
- -Djava.rmi.server.codebase=http://199.2.24.153:8000/calc.jar
- -Djava.security.policy=e:\policy
- -Djava.rmi.server.useCodebaseOnly=true calc.CalculatorService

start_client.bat

java -Djava.security.policy=e:\policy
- -Djava.rmi.server.useCodebaseOnly=true calc.CalculatorClient

Possible problems you may run into
Problem: When you try launching lookup service, some time you may receive an exception saying that "the Activation System is not running..." even though your RMID is running.

Solution: Increase the time interval between starting each of your services.

Problem: When you try launching lookup service, sometimes you may receive an exception saying that "can't create directory or directory exists".

Solution: You have to check whether you have permissions to create a directory and the reggie_log_dir is not already existing.

Problem: When you try launching server application, sometimes you may receive an exception saying that "can't find stub class".

Solution: Check if the stub class has been created via rmic tool for your service class. If this is ok, you will need to check whether the command line environment's CLASSPATH variable includes your Jini service path. If so would need to remove it. Re-start your rmiregistry and start the server application once again.

For More Information

California Software Laboratories helps companies develop business to consumer and business to business e-commerce solutions using state of the art technologies & tools. Visit the CSWL E-Commerce Services section for more information.

Want to learn more? Contact CSWL for Additional information.