

UNIVERSITY OF  
CALGARY

## **Unleashing the Jini: A Practical Guide to Using Jini**

Lance Titchkosky

Department of Computer Science, University of Calgary

December 18, 2002

[lance@alumni.ucalgary.ca](mailto:lance@alumni.ucalgary.ca)

### **Jini, what is it all about?**

In today's world, we are seeing the adoption an increasing number of pervasive computing devices. Things like wireless networking enabled personal digital assistants (PDA's), cell phones, printers and laptop computers are becoming more and more common in both the workplace and at home. There is a need for these network connected devices to be able to share their services with other devices in the hope of gaining synergy and making their use much easier. Jini technology is a standard released by Sun Microsystems that allows network devices to do just that.

Jini is built using Java technology and therefore is a cross-platform system. Jini is also network independent so it can be run on top of any type of network that is in use. It can be used with network technologies such as BlueTooth, Ethernet, IrDA (for infrared use), Firewire and many more [jini]. The main goal of Jini is to provide an environment for "creating dynamically networked components, applications, and services that scale from the device to the enterprise" [jini].

### **Services**

At the heart of Jini are services. Services can be "a computation, storage, a communication channel to another user, a software filter, a hardware device, or another user" [jini3]. In order for a device to offer a service, it is assumed that the device itself has enough memory and processing power to run Java and the implementation of Jini on it. This however could be a problem for some resource constrained devices. Sun's

implementation of Jini is probably too large (somewhere around 2.3 MB) for most of these resource-constrained devices [lenders]. This may not be that much of a problem in the future as technology becomes cheaper and faster. PsiNaptic, a company in Calgary, Alberta has designed a reference platform and a micro edition of Jini called JMatos that runs over a Bluetooth network. JMatos is a much smaller implementation of Jini (around 100k) and is able to run on fairly low-end inexpensive embedded systems [psi]. Another option for devices or services that cannot run Java is to have them connect to a proxy which can. This proxy can then act as a wrapper for that service.

## **Jini Lookup Service**

Jini relies on the use of a central server called the Jini Lookup Service (JLS) that maintains a list of services provided by devices it knows about [bettstetter]. In order to join a federation a JLS must first be found, this is done by multicasting a discovery message. Once the JLS has been found the device can register itself with the JLS. The registration process is known as joining a Jini federation. The registration consists of acquiring a lease from the JLS and letting the JLS know what services you have to offer (if any). Devices that have joined the federation must update their leases periodically to maintain the connection to the JLS. Anytime a device wants to use a service that is registered with the JLS, the device contacts the JLS to locate the service, and then obtains a lease for that service. The leasing procedure allows the JLS to cleanup services that are no longer in use or devices that have left the federation. A device using Jini can enter and leave federations at will; it also can be a member of multiple federations at the same time [psi] and may or may not host its own JLS that other devices can connect to.

## **How Services are Registered and Used**

Once the JLS is found a service provide can register its services by providing the JLS with a service proxy. The proxy defines the services that the device offers and is used by a client that wishes to use a service. The server proxy appears to a client basically as a Java object with various methods that can be invoked.

When a client wishes to use a service it connects to the lookup service, searches for a network service it wishes to use, and then is given access to that network service. This whole process described above is illustrated in Figure 1. As you can see it takes 6 steps from the initial discovery of a JLS to the use of that service by a client.

A real world example of this would be something like a user who has a Jini enabled digital camera with a wireless connection of some type wants to print a photo on a printer. The user would select the photo on the camera and select print. The camera would then send

out a broadcast message looking for a JLS. Once it found one it would search on the JLS for a photo printer. Let us assume that the JLS already has a photo printer in its federation. The camera would request a lease for the photo printer and the JLS would return a service proxy of the photo printer to the camera. Now that the camera has service proxy it can print directly to the printer [jini3].

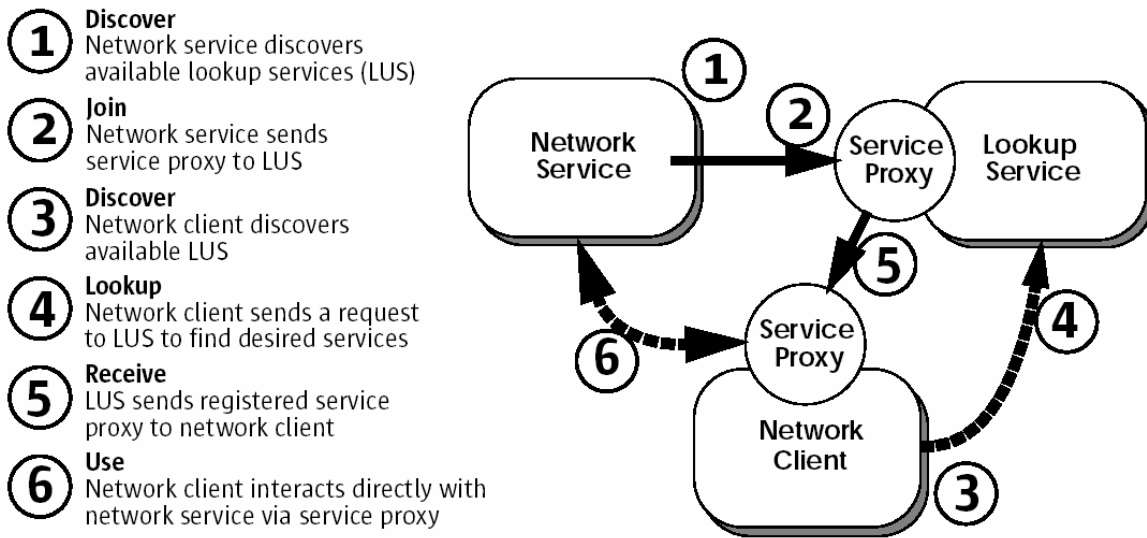


Figure 1 – Steps in the setup and use of a service on a Jini federation [jini2]

## Simple Code Example

In this section, the simplest Jini service will be explored. This of course is the ‘HelloWorld’ service; all this service does is return the string ‘HelloWorld’ whenever it is called. This example has three parts. The first section of code is an interface that defines what methods the HelloWorld service will offer, second is the code that that implements that interface, and lastly is code for a client that connects to the JLS to use the HelloWorld service. In order for this code to run, the machine you are running needs to be running a JLS service. For testing I used Sun’s sample JLS called reggie and an Apache web server. Since this code is using RMI, the RMI stub/skeleton code must also be generated for the HelloWorld class using the rmic command.

```
import java.rmi.Remote;
import java.rmi.RemoteException;

public interface HelloInterface extends Remote {
    public String getHelloString() throws RemoteException;
}
```

Code block 1 – The Interface for the HelloWorld service

```

import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import net.jini.core.*;
import net.jini.core.discovery.LookupLocator;
import net.jini.core.entry.Entry;
import net.jini.lookup.entry.Name;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceRegistration;

/**
 * This class implements a simple HelloWorld Jini Service
 */
public class HelloWorld extends UnicastRemoteObject implements HelloInterface{
    public HelloWorld() throws RemoteException {}

    public String getHelloString() throws RemoteException {
        return ("Hello Jini!");
    }

    public static void main (String args[]) throws Exception {
        System.setSecurityManager (new RMISecurityManager ());
        // Find the lookup service on the localhost
        LookupLocator lookupLocator = new LookupLocator("jini://localhost");
        ServiceRegistrar registrar = lookupLocator.getRegistrar();

        // We now create the service for the HelloWorld object
        // First we need a description of the service, we just give it a name
        Entry[] identifyingAttributes = new Entry[1];
        identifyingAttributes[0] = new Name("HelloWorld");

        HelloWorld helloWorld = new HelloWorld();
        ServiceItem myService = new ServiceItem(null,helloWorld,identifyingAttributes);

        // We register the service with the lookup service for a lease of 10 mins
        ServiceRegistration Service = registrar.register(myService, 600000);

        // That's it this service should now be active
        System.out.println ("Jini federation has been joined...");
    }
}

```

### Code block 2 – The implementation of the HelloWorld Service

```

import java.rmi.RMISecurityManager;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.discovery.LookupLocator;
import net.jini.core.entry.Entry;

```

```

import net.jini.lookup.entry.Name;

/**
 * This class implements a simple client that uses a Jini service
 */
public class HelloClient {
    public static void main (String args[]) throws Exception {
        System.setSecurityManager (new RMISecurityManager ());

        // Find the lookup service on the localhost
        LookupLocator lookupLocator = new LookupLocator("jini://localhost");
        ServiceRegistrar registrar = lookupLocator.getRegistrar();

        // Create a template for the service we want to find, the HelloWorld service
        Entry[] serverAttributes = new Entry[1];
        serverAttributes[0] = new Name("HelloWorld");
        ServiceTemplate template = new ServiceTemplate (null, null, serverAttributes);

        // We now can lookup the service and get an interface to it
        HelloInterface helloWorldInterface =
            (HelloInterface)registrar.lookup(template);

        // If that was successful then we can call methods
        if(helloWorldInterface != null) {
            System.out.println(helloWorldInterface.getHelloString());
        }
    }
}

```

**Code Block 3 – The client that uses the HelloWorld service**

## Jini's use in Multi-agent Systems

There are quite a few multi-agent frameworks out there that use Jini as their communication technology. The University of Maryland has two multi-agent projects which both use Jini; Ronin a Jini-based distributed agent development framework and JPES a Jini Prolog Engine Service [chen]. Another multi-agent framework based on Java and Jini is Paradigma which was developed at the University of Southampton by Ashri and Luck. Ashri and Luck believe Jini is an ideal technology for multi-agent systems as it resolves many of the problems that arise with highly dynamic network topologies like those that exist in multi-agent systems [luck]. Jini also has a fairly sophisticated lookup mechanism which is a fairly common necessity in a multi-agent system. From the example code here you can see how easy it is to use Jini. Its use in a multi-agent system can allow you to focus on other issues with the agents rather than just trying to figure out how to get your agents to connect to each other, in fact Ashri and Luck refer to Jini as the “plumbing” in their Paradigma framework [luck].

## References

- [bettstetter] C. Bettstetter, C. Renner, “A Comparison of Service Discovery Protocol and Implementation of the Service Location Protocol”,  
<http://www.tgs.cs.utwente.nl/eunice/summerschool/papers/paper5-1.pdf>,  
2000.
- [chen] H. Chen, Harry Chen’s Research homepage,  
<http://users.ebiquity.org/~hchen4/research.shtml>
- [jini] Sun Microsystems, Jini homepage, <http://www.sun.com/software/jini/>,  
2002.
- [jini2] Sun Microsystems, Jini Datasheet,  
<http://www.sun.com/software/jini/whitepapers/jini-datasheet0601.pdf>,  
2001.
- [jini3] Sun Microsystems, Jini Technology Architectural Overview,  
<http://www.sun.com/software/jini/whitepapers/architecture.html>, 2002.
- [lenders] V. Lenders et al., “Hybrid Jini for Limited Devices”,  
<http://www.tik.ee.ethz.ch/~lenders/publication/hybrid-jini-icwlhn01.pdf>,  
December 2001.
- [luck] R. Ashri and M. Luck, Paradigma: Agent Implementation through Jini,  
<http://www.ecs.soton.ac.uk/~ra00r/publications/dexa00.ps>, IEEE Computer  
Society Press, 2000.
- [psi] PsiNaptic, “A Jini Lookup Service for Resource-constrained Devices”,  
<http://www.psinaptic.com/oem/jinidoc.pdf>, January 2002,