TTM47AC Laboratory in construction of self-configuring systems

Jini Introduction

September 1, 2003

Fakultet for informasjonsteknologi, matematikk og elekroteknikk
INSTITUTT FOR TELEMATIKK

Contents

1. Jini Overview

1.1 What is Jini?

Jini is a programming model/infrastructure that enables dynamic deployment and configuration of distributed systems. It provides simple mechanisms, which enable components to plug together to form a community -- a community put together with minimal planning, installation, and human intervention. Components may be physical devices or software objects (written in Java) that provide Services over published interfaces. In Jini terminology such objects and devices are called Services. Services can be connected to a network and announce their presence to a Lookup Service. Clients that wish to use the Services can then locate them and call them to perform tasks. Jini is intended for networks where clients and Services come and go, as for instance in a mobile computing environment, but it can be used in any network where there is some degree of dynamic changes.

1.2 How Jini works?

The Java programming language is the key to making Jini technology work. The ability to dynamically download and run code is central to a number of the features of the Jini architecture.
In a running Jini system, there are three main types of players which are connected to a network. There is a *Service*, such as a printer, a digital camera, a coffee machine, etc., a *client* that would like to use this Service, and a *Lookup Service* (Service locator) that acts as a broker between Services and clients. Figure 1.1 shows the basic components of Jini.

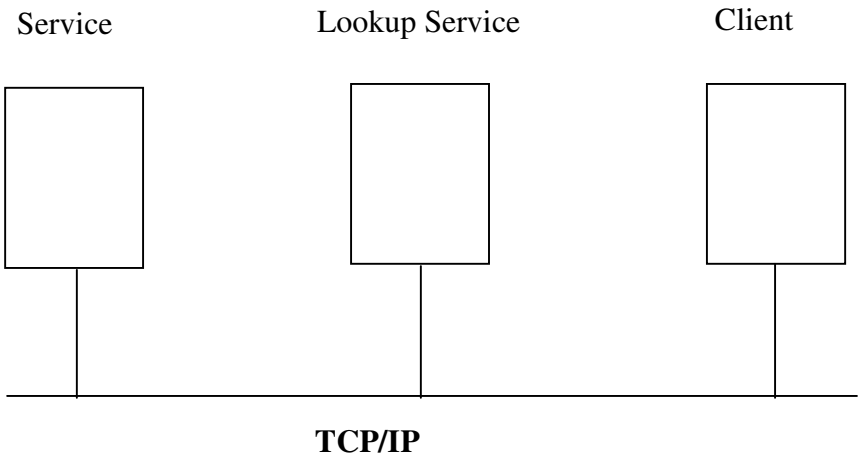

Figure 1.1: Components of a Jini system

Jini technology uses a Lookup Service with which Services register. When a Client (possibly a device) plugs in, it goes through an add-in protocol, called *discovery* and *join*: the device first locates the Lookup Service (discovery) and then uploads a *Service Registrar object* that it uses to access the Lookup Service. The Service Registrar object will serve as a local proxy for the Lookup Service. It implements the Lookup Service's interface and the protocol for communication between the Client and the Lookup Service. To use a Service, a Client first locates it using the Lookup Service. A Service Object (a proxy) representing the Service is then uploaded from the Lookup Service to the requesting Client. The Service Object implements the Service's interface and takes care of communication with the remote Service. The Client may now use the (remote) Service by invoking methods on the local Service Object. Once the connection is made, the Lookup Service is not involved in any of the resulting interactions between the Client and Service. See Fig 1.2.

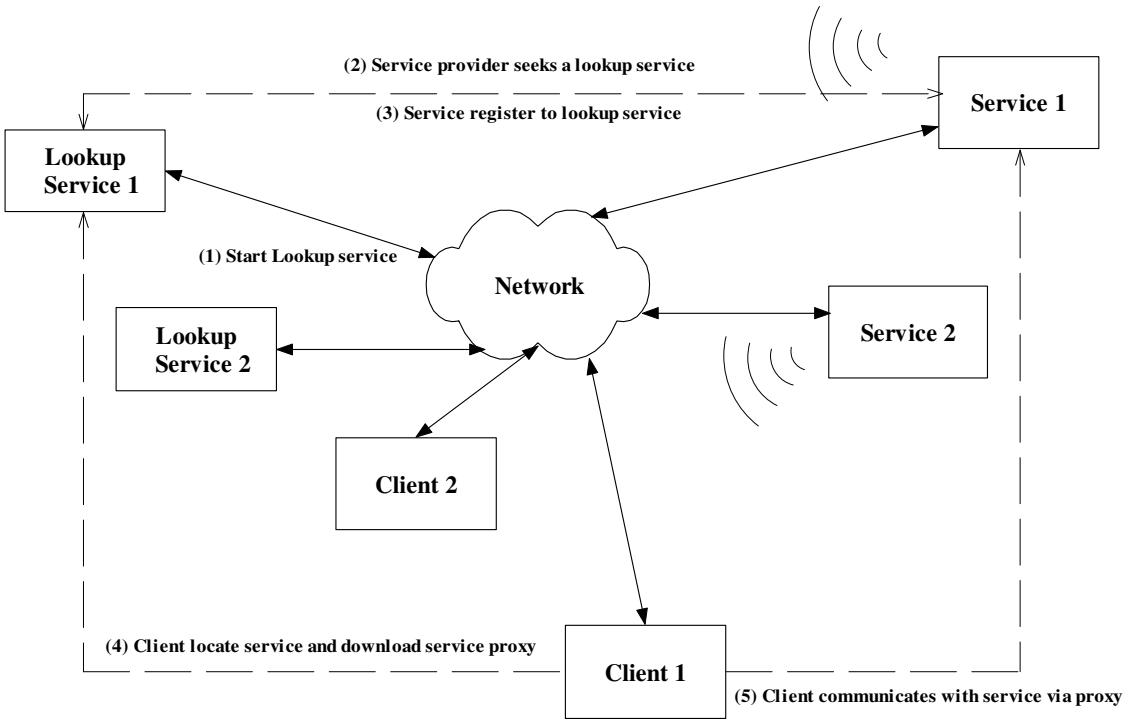

Figure 1.2: A typical Jini system

It doesn't matter where a Service is implemented -- compatibility is ensured because each Service provides everything needed to interact with it. There is no central repository of drivers, or anything else for that matter.

1.3 Jini versus classical distributed systems technology

Jini emerges from a tradition where sequential programming is in focus and where networking and concurrency is something new and difficult that must be hidden. This is apparent in the use of proxies/Service Objects that enable Clients to communicate with

Services using method calls as if there were no network in-between. The same principle is applied in other client-server middleware for distributed systems such as RPC (Remote Procedure Calls), CORBA, Java RMI and Microsoft DCOM.

- **Simplicity**

Jini is, at its heart, about how Services and Clients connect to one another – not about what those Services are or how they work. In fact, Jini Services can even be written in a language other than Java; the only requirement is that there exists a bit of code that is written in Java that can participate in the mechanisms Jini uses to find other Jini devices and Services. Jini lets programs use Services in a network without knowing anything about the protocol used by the Service. For example, one implementation of a Service might be SOAP-based, another one might be RMI-based, and a third one might be CORBA-based. In effect, each Service provides its clients with the mechanism to talk to it. A Service is defined by its programming API, declared as a Java language interface. Jini technology is claimed to be the only solution that makes the details of how a Service uses the network into an implementation detail that can differentiate between implementations of the same Service, without changing the client code.

- **Reliability**

Jini supports serendipitous interactions among Services and users of those Services. That is, Services can appear and disappear on a network in a very lightweight way. Jini allows Services to come and go without requiring any static configuration or administration.
Another feature of Jini, which make Jini networks more reliable, is that the Jini community is self-healing: it doesn't make the assumption that networks are perfect, or that software never fails. It can adapt when Services (and consumers of Services) come and go and it also supports redundant infrastructure in a very natural way. This is supported by the Lease mechanism in Jini, see Chapter 2.4.

- **Scalability**

Jini addresses scalability through federation. Federation is the ability for Jini communities to be linked together, or federated, into larger groups. Each new device and Service could join a community easily by employing Jini technology and make that community more valuable for users who use that community.

It is claimed that Jini technology enable developers to:

- Create applications and Services that interact without preinstalled drivers
- Take advantage of Jini technology's self-healing nature; no need to reconfigure upon failure
- Clean up is automatic, so garbage collection is eliminated
- Pay only for Services that are used
- Ensure that applications are always current, without downtime or upgrading
- Change vendors at any time
- Modify equipment configurations easily, including legacy hardware and software

2. Jini Technology Infrastructure

2.1. Lookup Service

Jini technology uses a Lookup Service with which devices and Services register. A Client locates a Service by querying a Lookup Service. Thus, the initial phase of both a Client and a Service is to discover a lookup Service. Lookup Services will usually have been started by some independent mechanism. The search for a Lookup Service can be done either by unicast or by multicast.

2.1.1 Multicast Request Protocol

The multicast request protocol is a Service-initiated protocol that is used when a Service needs to discover all the Lookup Services that may exist on a local network. The process starts when an initiating Service sets itself up to both send multicast request and receive unicast replies. This is done by creating a couple of sockets, one for outgoing multicast UDP messages and one for incoming TCP messages. Likewise, the Lookup Service will set up two sockets too, but in the reverse way, as Fig 2.1 shows.

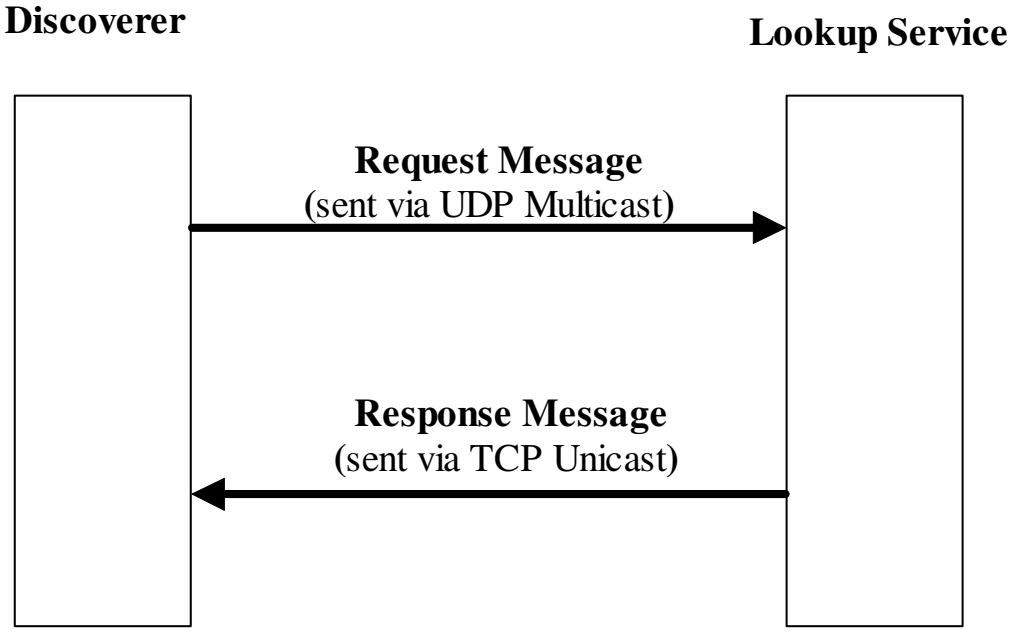

Figure 2.1: The multicast request protocol

2.1.2 Multicast Announcement Protocol

The multicast announcement protocol is used by Lookup Services to announce their presence to any interested parties that may be listening and within the range of the broadcast scope. When a new Lookup Service connects to a local network, it will send out a "multicast presence announcement" message and it will periodically sends out the same message during its life time. Fig 2.2 shows the communication flow.

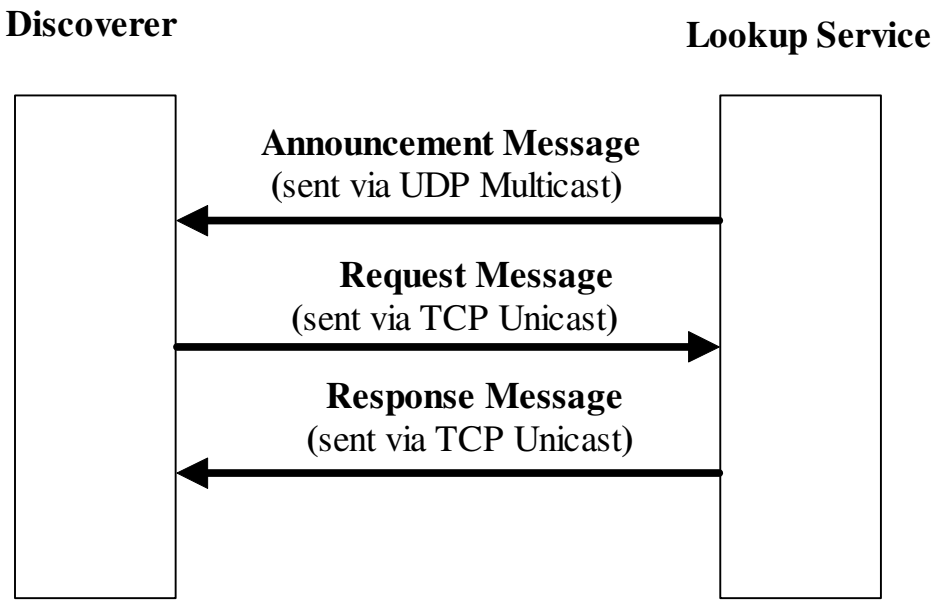

Figure 2.2: The multicast announcement protocol

2.1.3 Unicast Discovery Protocol

Unicast discovery can be used when the discoverer (Service or Client) already knows the machine on which the Lookup Service resides, and thus can ask for it directly. More commonly, it is used to contact non-local Lookup Services (beyond multicast radius) or after multicast protocols result in contact. It starts when the discovering entity sends a unicast request and waits for response from the Lookup Service. Unlike multicast, unicast requires explicit information about the location of the Lookup Service. If the Lookup Service received the request, it will respond with a Lookup Service Registrar Object (an RMI Marshaled Object).

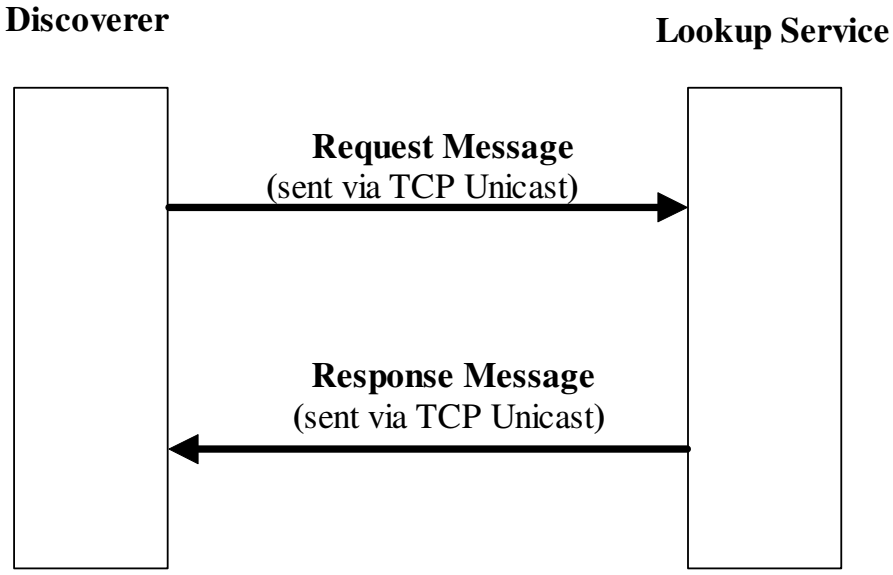

Figure 2.3: The unicast discovery protocol

2.1.4 Using Discovery in Applications

2.1.4.1 Interface DiscoveryListener

The basic interface for Jini-aware applications that need to work with discovery is called *DiscoveryListener.* This is a very simple Java interface with only two methods:discovered(DiscoveryEvent ev) and discarded(DiscoveryEvent ev). Any object that needs to find lookup Services via the high-level discovery APIs that Jini provides must implement this interface. The first method, discovered(), will be called whenever a new lookup Services is found via the discovery protocols. The second method, discarded(), will be called when a previously found lookup Service is no longer relevant or there are problems when communicating with a previously found lookup Service.

2.1.4.2 Class DiscoveryEvent

Note that both discovered() and discarded() in interface *DiscoveryListener* take an instance of DiscoveryEvent. The most important method of this class is getRegistrars(). This method returns an array of *ServiceRegistrar* instances. Each of these is a Service object/proxy for a Lookup Service somewhere on the network.

2.1.4.3 Class LookupDiscovery

Both multicast requests and announcements are implemented in the *net.jini.discovery.*`LookupDiscovery` class. `LookupDiscovery` first attempts to locate Lookup Services using multicast requests. After a few moments, it should find all the Lookup Services on the network. It then switches to passively listening to multicast announcements in case a new Lookup Service is added to the network.
`LookupDiscovery` takes an instance of *`DiscoveryListener`* in its constructor. It sends the discovered and discarded messages when it discovers or looses track of lookup Services respectively.

2.1.4.3 Class LookupLocatorDiscovery

The *net.jini.discovery.LookupLocatorDiscovery* class implements only the unicast protocols. This class encapsulates the functionality required of an entity that wishes to employ the unicast discovery protocol to discover a Lookup Service. This utility provides an implementation that makes the process of finding specific lookup Services much simpler for both Services and clients.
Because this class participates in only the unicast discovery protocol, and because the unicast discovery protocol imposes no restriction on the physical location of the entity relative to a lookup Service, this utility can be used to discover lookup Services running on hosts that are located far from, or near to, the host on which the entity is running.

2.1.4.4 Class LookupDiscoveryManager

The most convenient way to control the discovery process in Jini is via the class *net.jini.discovery.LookupDiscoveryManager*. This class provides a complete suite of functionality for controlling both multicast and unicast forms of discovery, and for doing general "housekeeping" on the discovery process. It implements three of general discovery interfaces, called *Discoverymanagement* (which defines bookkeeping and listener control), *DiscoveryGroupManagement* (which defines APIs for controlling

multicast discovery), and *DiscoveryLocatorManagement* (which defines APIs for controlling unicast discovery).

2.1.5 A Discovery Example

The discovery process could be demonstrated by the following example:

- A Jini-enabled digital camera is plugged into the network
- The camera sends out a multicast presence announcement
- The Lookup Service receives the camera's announcement packet
- The Lookup Service contacts the digital camera
- The camera sends a unicast discovery request packet to the Lookup Service
- The Lookup Service sends a Service Registrar object
- The camera can register itself (via join) through the Service registrar

2.2 The Service Join Protocol

The most important concept within the Jini architecture is a *Service*. A Service is an entity that can be used by a person, a program, or another Service. A Service may be a computation, a storage, a communication channel to another user, a software filter, a hardware device, or another user. Members of a Jini system federate in order to share access to Services. A Jini system consists of Services that can be collected together for the performance of a particular task. Services may make use of other Services, and a Client of one Service may itself be a Service with Clients of its own. The dynamic nature of a Jini system enables Services to be added or withdrawn from a federation at any time according to demand, need, or the changing requirements of the workgroup using it.
A *Service* is usually defined by a Java interface, and commonly the Service itself will be identified by this interface. Each Service can be implemented in many ways, by many different vendors.

In order for the Service to register the Service Object with a Lookup Service, the Service must first *find* the Lookup Service. This can be done in two ways: unicast or multicast requests, as described in the Lookup Service protocol. When the Lookup Service gets a request on this port, it sends an *object* back to the server. This object, known as a *registrar*, acts as a proxy to the Lookup Service, and runs in the Service's JVM (Java Virtual Machine). Any requests that the Service needs to make use 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 will be Java RMI (Remote Method Invocation).

What the Service does with the Registrar is to *register* the Service with the Lookup Service. This involves taking a copy of the Service Object, and storing it on the Lookup Service as showed in Fig 2.4.

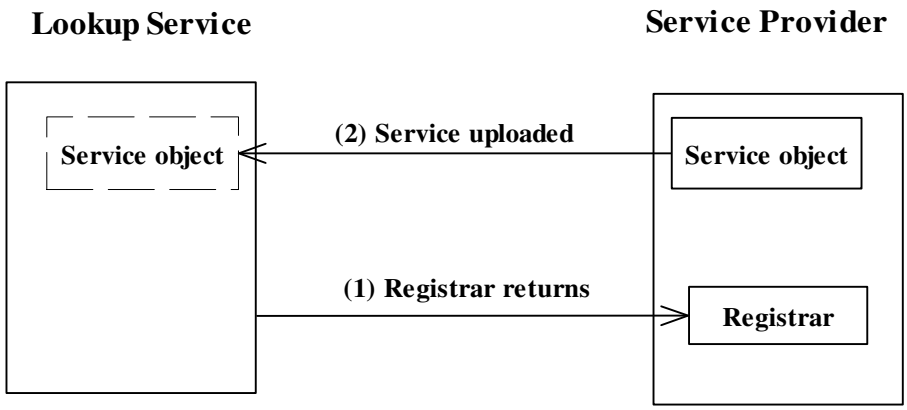

Figure 2.4: The Service join protocol

2.3 Client Lookup

Clients begin their search for Services in the same way that Services begin their lives: They use discovery to find Lookup Services. There are a few ways clients can search for Services in a Lookup Service depending upon which particular search API is used. The Lookup Service will correspondingly either return to the client a single Service proxyObject, or a set of ServiceItems, each of them contains a Proxy as well as any attributes associated with the proxy. Once a client has a proxy in its hands, it can invoke methods on the proxy to interact with the Service. This process is showed in Fig 2.5.

At this stage there is the original Service Object running back on the Service. There is a copy of the Service Object stored in the Lookup Service, and a copy of the Service Object running in the Client's JVM. The Client can now invoke methods on the Service Object running in its own JVM.

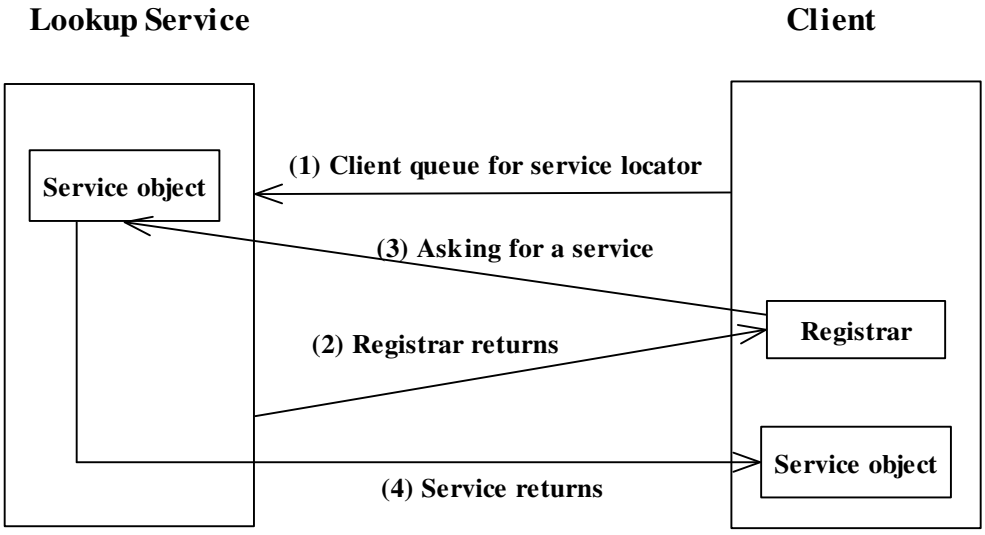

Figure 2.5: Client search for Service

In addition to searching, Clients can also ask a Lookup Service to send notifications when new Services appear, existing Services change, or previously registered Services go away.

2.4 Leasing

Access to many of the Services in the Jini system environment is *lease* based. A lease is a grant of guaranteed access over a time period. Each lease is negotiated between the user of the Service and the provider of the Service as part of the Service protocol: A Service is requested for some period; and access is granted for some period, presumably taking the request period into account. If a lease is not renewed before it is freed--either because the resource is no longer needed, the client or network fails, or the lease is not permitted to be renewed--then both the user and the provider of the resource may conclude the resource can be freed. Leasing is probably the most important feature when it comes to the robustness of the Jini infrastructure. With leasing, the Lookup Service is able to get rid of outdated entries for Services that did not or could not clean up after themselves. Without leasing, the Lookup Service would eventually be bogged down with Service Objects that are no longer valid. Clients would have to waste time when they received invalid Service Objects.

Leases are either exclusive or non-exclusive. Exclusive leases insure that no one else may take a lease on the resource during the period of the lease; non-exclusive leases allow multiple users to share a resource.

3. A Simple Example

To get started with Jini, you will need the following to be installed on your machine:

- The Java 2 compiler and API – JDK 1.3 or higher
- The Jini 1.2 Starter Kit

Follow the instructions in the course's website to download and install all the necessary software.

In this example, a simple Jini Time Service and a client application to use that Service will be implemented. The Jini Service returns the current time in milliseconds when it's getTime() method is invoked by a client.

The Steps involved in developing the example are:

1. Code the Jini Service Interface
2. Code the Jini Service
3. Code the Jini Client
4. Compile the Service and Client code
5. Run the Service and Client application

1. Code the Jini Service Interface

```
/*************************************************************************
 MyServerInterface.java
 *************************************************************************/

public interface MyServerInterface{
  public long getTime( );
}
```

2. Code the Jini Service

```
/*************************************************************************
 MyServer.java
 *************************************************************************/

import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.discovery.LookupDiscovery;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceRegistration;
import java.util.HashMap;
import java.io.IOException;
import java.io.Serializable;
```

```
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;

/**
 * This is the proxy object that will be downloaded
 * by clients.  It's serializable and implements
 * MyServerInterface.
 */
class MyServerProxy implements Serializable, MyServerInterface {
    public MyServerProxy() {
    }
    public long getTime() {
        long currentTime = System.currentTimeMillis();
        return currentTime;
    }
}

public class MyServer implements Runnable {
    // 10 minute leases
   protected final int LEASE_TIME= 10 * 60 * 1000;
    protected HashMap registrations = new HashMap();
    protected ServiceItem item;
    protected LookupDiscovery disco;

    // Inner class to listen for discovery events
    class Listener implements DiscoveryListener {
        // Called when we find a new Lookup Service.
        public void discovered(DiscoveryEvent ev) {
            System.out.println("discovered a Lookup Service!");
            ServiceRegistrar[] newregs = ev.getRegistrars();
            for (int i=0 ; i<newregs.length ; i++) {
                if (!registrations.containsKey(newregs[i])) {
                    registerWithLookup(newregs[i]);
                }
            }
        }

 /**
   * Called ONLY when we explicitly discard a Lookup Service, not "automatically"
   * when a Lookup Service goes down.  Once discovered,
   * there is NO ongoing communication with a Lookup Service.
   */
 public void discarded(DiscoveryEvent ev) {
            ServiceRegistrar[] deadregs = ev.getRegistrars();
            for (int i=0 ; i<deadregs.length ; i++) {
                registrations.remove(deadregs[i]);
```

```
        }
    }
}

  public MyServer() throws IOException {
    item = new ServiceItem(null, createProxy(), null);

    // Set a security manager
    if (System.getSecurityManager() == null) {
      System.setSecurityManager(new RMISecurityManager());
    }

    // Search in the "ttm47ac" group
    disco = new LookupDiscovery(new String[] { " ttm47ac " });

    // Install a listener.
    disco.addDiscoveryListener(new Listener());
  }

  protected MyServerInterface createProxy() {
    return new MyServerProxy();
  }

  /**
   * This work involves remote calls, and may take a while to complete.
   * Thus, since it's called from discovered(), it will prevent us from responding
   * in a timely fashion to new discovery events.  An improvement would be
   * to spin off a separate short-lived thread to do the work.
   */
  synchronized void registerWithLookup(ServiceRegistrar  registrar) {
    ServiceRegistration registration = null;

    try {
      registration = registrar.register(item, LEASE_TIME);
    } catch (RemoteException ex) {
      System.out.println("Couldn't register: " + ex.getMessage());
      return;
    }

    /**
     * If this is our first registration, use the Service ID returned to us.
     *  Ideally, we should save this ID so that it can be used after
     *  restarts of the Service
     */
    if (item.ServiceID == null) {
      item.ServiceID = registration.getServiceID();
```

```
            System.out.println("Set ServiceID to " + item.ServiceID);
        }

        registrations.put(registrar, registration);
    }

    // This thread does nothing but sleep, but it makes sure the VM doesn't exit.
    public void run() {
        while (true) {
            try {
                Thread.sleep(100000000);
            } catch (InterruptedException ex) {
            }
        }
    }

    // Create a new MyServer and start its thread.
    public static void main(String args[]) {
        try {
            MyServer myserver = new MyServer();
            new Thread(myserver).start();
        } catch (IOException ex) {
            System.out.println("Couldn't create Service: " +
                               ex.getMessage());
        }
    }
}
```

3. Code the Jini client

```
/*************************************************************************
 MyClient.java
 *************************************************************************/

// A simple Client to exercise MyServer

import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.discovery.LookupDiscovery;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceTemplate;
import java.io.IOException;
import java.rmi.RemoteException;
import java.rmi.RMISecurityManager;
```

```
public class MyClient implements Runnable {
   protected ServiceTemplate template;
   protected LookupDiscovery disco;

 // An inner class to implement DiscoveryListener
   class Listener implements DiscoveryListener {
      public void discovered(DiscoveryEvent ev) {
         ServiceRegistrar[] newregs = ev.getRegistrars();
         for (int i=0 ; i<newregs.length ; i++) {
            lookForService(newregs[i]);
         }
      }
      public void discarded(DiscoveryEvent ev) {
      }
   }

   public MyClient() throws IOException {
      Class[] types = { MyServerInterface.class };

      template = new ServiceTemplate(null, types, null);

      // Set a security manager
      if (System.getSecurityManager() == null) {
         System.setSecurityManager(
               new RMISecurityManager());
      }

      // Only search the “ttm47ac” group
      disco = new LookupDiscovery(new String[] { " ttm47ac " });

      // Install a listener
      disco.addDiscoveryListener(new Listener());
   }

   // Once we've found a new lookup Service, search for proxies that implement
   // MyServerInterface
   protected Object lookForService(ServiceRegistrar lusvc) {
      Object o = null;

      try {
         o = lusvc.lookup(template);
      } catch (RemoteException ex) {
         System.err.println("Error doing lookup: " + ex.getMessage());
         return null;
      }
```

```
        if (o == null) {
            System.err.println("No matching Service.");
            return null;
        }

        System.out.println("Got a matching Service.");
        System.out.println("The time at server is: " +
                           ((MyServerInterface) o).getTime());
        return o;
    }

    // This thread does nothing--it simply keeps the
    // VM from exiting while we do discovery.
    public void run() {
        while (true) {
            try {
                Thread.sleep(1000000);
            } catch (InterruptedException ex) {
            }
        }
    }

    // Create a MyClient and start its thread
    public static void main(String args[]) {
        try {
            MyClient client = new MyClient();
            new Thread(client).start();
        } catch (IOException ex) {
            System.out.println("Couldn't create client: " +
                               ex.getMessage());
        }
    }
}
```

4. Compile the Service and Client code

Execute the *compile_example.bat* script. The client and service classes will be placed in the *ClientJini* and *Service* directories.

5. Run the Service and Client application

Start up the Service by executing the *run_example_service.bat* script. If everything goes well, you should get a message similar to the following:
discovered a lookup Service!

Set ServiceID to 9be927b8-c9a1-4ef1-9cc7-e3ed6b596c23

Now you ca run the client. To do it, just execute the *run_example_client.bat* script. The client should contact a Jini community, search for a Service that implements the Time Server interface, and download the code that implements that interface. If the Client succeeds in its search the following message will appear:

Got a matching Service.
The time at server is: 996702196213