

Jini

Jini

- Jini is a new framework from SUN for developing distributed services
- Jini goals:
 - ⇒ Enable a service-based architecture
 - ⇒ Support spontaneous networking
 - ⇒ Provide network “plug-and-work”
 - ⇒ Erase the distinction between hardware and software
 - ⇒ Provide an infrastructure to make writing distributed programs easier
- Much of the work in Jini explicitly addresses the difficulties in developing a distributed computing application:
 - ⇒ Latency
 - ⇒ Synchronization
 - ⇒ Partial Failure

Jini Technology Structure

Jini Services	<ul style="list-style-type: none">• JavaSpaces™• Transaction Managers• Printing, Storage, Databases...
Jini Infrastructure	<ul style="list-style-type: none">• Discovery• Lookup Service
Jini Programming Model	<ul style="list-style-type: none">• Leasing• Distributed Events• Transactions
Java 2 Platform	<ul style="list-style-type: none">• Java RMI• Java VM

Jini Terminology

- A Jini system is organized as a set of *federations* (or *communities* or *groups*) of services
- A service is any network-available component, either hardware or software
- Jini uses a *lookup service* to allow services to advertise their availability to client
- A service *joins* a community by *discovering* an available lookup service and registering with it
- To use a particular service, a client performs a *lookup* in the lookup service

Jini Operations

- **Discovery**
 - ⇒ Services and clients locate and obtain references to lookup services
- **Join**
 - ⇒ Services register their availability with one or more lookup services
- **Lookup**
 - ⇒ Clients locate and contact services

Discovery - Multicast Request Protocol

- The Multicast Request Protocol is used by a service or client to locate a “nearby” lookup service
- The service/client sends a UDP multicast packet to a well-known port (default 4160). The contents of this packet include:
 - ⇒ Groups the service/client wants to join
 - ⇒ IP address/port where service/client can be contacted
 - ⇒ Lookup services that this service/client have already heard from

Discovery - Unicast Discovery Protocol

- The Unicast Discovery Protocol is used by a service or client to communicate with a specific lookup service, for example, a non-local lookup service outside the multicast radius
- The response part of this protocol is also used after a multicast request has discovered a lookup service
- A service/client sends a TCP request to a well-known port (default 4160) on a specific host
- The lookup service responds with a serialized MarshalledObject that contains the proxy object for the lookup service (a ServiceRegistrar object)

Discovery - Multicast Announcement Protocol

- The Multicast Announcement Protocol is used by Jini lookup services to announce their availability to services/clients within the multicast radius
- This is useful for new lookup services and after network failures
- A lookup service periodically sends a UDP multicast packet to a well-known port (default 4160). The contents of this packet include:
 - ⇒ IP address/port where Unicast discovery of this lookup service can be done
 - ⇒ List of groups this lookup service manages

LookupDiscovery Helper Class

- To facilitate discovery of lookup services, the Jini distribution includes a helper class called LookupDiscovery
- When a LookupDiscovery class is instantiated, a set of separate daemon threads are created which perform the Multicast Request Protocol and also handle the Multicast Announcement Protocol
- The LookupDiscovery object allows other objects to add themselves as listeners for DiscoveryEvents and to be notified when a lookup service is discovered
- Listener objects must implement the DiscoveryListener interface which has these methods:
 - ⇒ `public void discovered(DiscoveryEvent e)`
 - Called when one or more lookup service registrars has been discovered
 - ⇒ `public void discarded(DiscoveryEvent e)`
 - Called when one or more lookup service registrars has been discarded

Discovery Example

```
import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.discovery.LookupLocator;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceID;
import java.rmi.*;

/**
 * Class which discovers all available Lookup Services.
 */
public class FindAllLookupServices implements DiscoveryListener {

    public static void main (String[] args) {
        try {
            // Install a security manager.
            System.setSecurityManager (new RMISecurityManager());
```

Discovery Example (Continued)

```
// Create a LookupDiscovery object.
LookupDiscovery ld = new
    LookupDiscovery(LookupDiscovery.ALL_GROUPS);

// Add an object of this class as a DiscoveryListener.
ld.addDiscoveryListener(new FindAllLookupServices());

// Discover for 10 minutes.
Thread.currentThread().sleep(1000 * 60 * 10);

// Terminate discovery.
ld.terminate();
}
catch (Exception e) {
    System.out.println("Exception:" + e);
}
}
```

Discovery Example (Continued)

```
// Invoked whenever a new Lookup Service is discovered.
public void discovered(DiscoveryEvent de) {
    try {
        System.out.println( "Lookup service(s) discovered.");

        // Get the proxy objects for the services discovered.
        ServiceRegistrar lookupServices[] = de.getRegistrars();

        // Display info about each Lookup Service.
        for (int i = 0; i < lookupServices.length; i++) {
            ServiceID id = lookupServices[i].getServiceID();
            LookupLocator lookupLocator =
                lookupServices[i].getLocator();
            String host = lookupLocator.getHost();
            int port = lookupLocator.getPort();
            System.out.println("Lookup Service: jini://" + host + ":"
                + port + ", " + id.toString());
        }
    }
}
```

Discovery Example (Continued)

```
String[] groups = lookupServices[i].getGroups();
if (groups.length > 0 ) {
    System.out.println( "Groups of Lookup Service are: ");
    for (int k = 0; k < groups.length; k++ )
        System.out.println(groups[k]);
    }
}
}
catch (Exception e) {
    System.out.println("Exception:" + e);
}
}

// Invoked whenever a Lookup Service is discarded.
public void discarded(DiscoveryEvent de) {}

}
```

The Join Operation

- Once a service has discovered a lookup service(s), the service can *join* the Jini community by *registering* with one or more lookup services
- The service interacts with the lookup service through a proxy object, the ServiceRegistrar object
- The ServiceRegistrar proxy object is downloaded over the network from a location specified when the lookup service is started
- The service registers itself with the lookup service by invoking the register method of the ServiceRegistrar object:
 - ⇒ `public ServiceRegistration register(ServiceItem item,
long leaseDuration)`

The Join Operation (Continued)

- The ServiceItem object contains the following:
 - ⇒ A ServiceID object, or null if registering for the first time
 - ⇒ An instance of the service proxy object
 - ⇒ An array of attributes which we want to associate with the proxy object to aid in lookup
- The attributes can be used to locate a particular service, such as a color printer on the third floor of the engineering department
- The instance of the proxy object is serialized and sent to the lookup service encapsulated inside a MarshalledObject. The MarshalledObject class is new to Java 2 and allows the proxy object to be sent to the lookup service without the lookup service having to deserialize the actual proxy object and download its class file over the network.

Leases

- All resources in Jini are leased to clients. This is one way that Jini addresses the issue of service failures and self-healing.
- The service requests registration for the specified lease time. The lookup service may or may not grant this amount of time. Details on the actual lease granted can be obtained from the `ServiceRegistration` object.
- Leases need to be periodically renewed. The Jini distribution provides a `JoinManager` helper class to assist services in joining Jini communities and periodically renewing their leases in lookup services.

The Lookup Operation

- To use a service, a client must first discover a lookup service and then perform a *lookup* to obtain the desired service proxy
- The client interacts with the lookup service the same as a service does, through a ServiceRegistrar object
- The ServiceRegistrar proxy object is downloaded over the network from a location specified when the lookup service is started
- The client performs a lookup for a specific service by invoking the lookup() method of the ServiceRegistrar object:
 - ⇒ `public Object lookup(ServiceTemplate tmpl)`

The Lookup Operation (Continued)

- The ServiceTemplate object contains the following:
 - ⇒ A ServiceID to match, or null
 - ⇒ An array of service types to match, or null
 - ⇒ An array of attributes to match, or null
- The ServiceTemplate object is used to match against the ServiceItem objects maintained by a lookup service
- A null field of the template indicates a wildcard for the match A service item (item) matches a service template (tmpl) if:
 - ⇒ item.serviceID equals tmpl.serviceID (or if tmpl.serviceID is null) and
 - ⇒ item.service is an instance of every type in tmpl.serviceTypes and
 - ⇒ item.attributeSets contains at least one matching entry for each entry template in tmpl.attributeSetTemplates

The Lookup Operation (Continued)

- An entry matches an entry template if the class of the template is the same as, or a superclass of, the class of the entry, and every non-null field in the template equals the corresponding field of the entry. Every entry can be used to match more than one template. Note that in a service template, for serviceTypes and attributeSetTemplates, a null field is equivalent to an empty array; both represent a wildcard.

Jini Example

- The classic “Hello, World” Example using Jini!
- First, define the desired remote interface:

```
/**
 * Hello Interface.
 */
public interface IHello {
    public String sayHello();
}
```

- Note that this interface is not a remote interface and does not use RMI. The service proxy object which the client will download provides a local implementation of this interface. A proxy object may indeed use RMI to interact with a remote service, but in this example we’ll keep things simple.

Jini Example (Continued)

- Here's our server:

```
import net.jini.discovery.*;
import net.jini.core.lookup.*;
import java.io.Serializable;
import java.rmi.RMISecurityManager;

/**
 * The proxy object which clients will download.
 */
class HelloWorldServiceProxy implements Serializable, IHello {
    public HelloWorldServiceProxy() {}

    public String sayHello() {
        return "Hello, world!";
    }
}
```

Jini Example (Continued)

```
/**
 * The remote server object.
 * This is a very simple version which does NOT renew its leases!
 */
public class HelloWorldService implements DiscoveryListener {

    private ServiceItem item;
    private final int LEASE_TIME = 10 * 60 * 1000;

    public HelloWorldService() {

        // Create a ServiceItem to send to the lookup service.
        // The serviceID and attributes array are both null here.
        ServiceItem item = new ServiceItem(null,
            new HelloWorldServiceProxy(), null);

    }
}
```

Jini Example (Continued)

```
// Handle lookup service discoveries.
public void discovered(DiscoveryEvent de) {
    ServiceRegistration registration = null;
    try {
        ServiceRegistrar[] regs = de.getRegistrars();
        for (int i = 0; i < regs.length; i++) {
            registration = regs[i].register(item, LEASE_TIME);
        }
    }
    catch (Exception e) {
        System.out.println("Exception:" + e);
    }
}

// Handle lookup service discards.
public void discarded(DiscoveryEvent de) {}
```

Jini Example (Continued)

```
public static void main(String args[]) {  
  
    try {  
        // Install a security manager.  
        if (System.getSecurityManager() == null) {  
            System.setSecurityManager(new RMISecurityManager());  
        }  
  
        // Create a LookupDiscovery object.  
        LookupDiscovery ld = new  
            LookupDiscovery(LookupDiscovery.ALL_GROUPS);  
  
        // Add a DiscoveryListener.  
        HelloWorldService hws = new HelloWorldService();  
        ld.addDiscoveryListener(hws);  
    }  
}
```

Jini Example (Continued)

```
// Hang around.
synchronized (hws) {
    hws.wait();
}
}
catch (Exception e) {
    System.out.println("Exception:" + e);
}
}
}
```

Jini Example (Continued)

- Here's a simpler version of the server which uses a JoinManager:

```
import com.sun.jini.lookup.ServiceIDListener;
import com.sun.jini.lookup.JoinManager;
import com.sun.jini.lease.LeaseRenewalManager;
import net.jini.core.lookup.ServiceID;
import java.rmi.RMI SecurityManager;
import java.io.Serializable;

/**
 * The proxy object which clients will download.
 */
class HelloWorldServiceProxy implements Serializable, IHello {
    public HelloWorldServiceProxy() {}

    public String sayHello() {return "Hello, world!";}
}
```

Jini Example (Continued)

```
/**
 * The remote server object.
 * This version uses a JoinManager to make things simpler.
 */
public class HelloWorldService2 implements ServiceIDListener {

    public HelloWorldService2() {}

    // Handle notifies of the service ID.
    public void serviceIDNotify (ServiceID uniqueID) {}

    public static void main(String args[]) {

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

Jini Example (Continued)

```
// Create a proxy object.
HelloWorldServiceProxy proxy = new HelloWorldServiceProxy();

// Create a remote server object.
HelloWorldService2 hws = new HelloWorldService2();

// Create a JoinManager to facilitate registering.
// The attributes array is null here.
JoinManager myManager = new JoinManager(proxy, null, hws,
    new LeaseRenewalManager());
}
catch (Exception e) {
    System.out.println("Exception:" + e);
}
}
}
```

Jini Example (Continued)

- Here's a client which does Unicast Discovery:

```
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.discovery.LookupLocator;
import java.rmi.RMISeccurityManager;

/**
 * This client uses Unicast Discovery to access the service proxy
 * object.
 */
public class HelloWorldClient {
    public static void main (String[] args) {
        try {
            // Install a security manager.
            if (System.getSecurityManager() == null) {
                System.setSecurityManager(new RMISeccurityManager());
            }
        }
    }
}
```

Jini Example (Continued)

```
// Create a LookupLocator object to perform Unicast Discovery.
LookupLocator lookup = new LookupLocator("jini://serverhost");
ServiceRegistrar registrar = lookup.getRegistrar();

// Create a ServiceTemplate to lookup the
// HelloWorldServiceProxy.
// The serviceID and attributes array are both null here.
Class[] types = {IHello.class};
ServiceTemplate template = new ServiceTemplate (null, types,
    null);
IHello hwsp = (IHello) registrar.lookup(template);
```

Jini Example (Continued)

```
// Invoke a method on the proxy.  
if (hwsp == null)  
    System.out.println("No matching service.");  
else  
    System.out.println(hwsp.sayHello());  
}  
  
catch (Exception e) {  
    System.out.println("Exception:" + e);  
}  
  
}  
}
```

Jini Example (Continued)

- Here's a client which does Multicast Discovery:

```
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.discovery.LookupDiscovery;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceRegistration;
import java.rmi.RMI SecurityManager;

/**
 * This client uses Multicast Discovery to access the service proxy
 * object.
 */
public class HelloWorldClient2 implements DiscoveryListener {

    private ServiceTemplate template;
```

Jini Example (Continued)

```
public HelloWorldClient2() {  
  
    // Create a ServiceTemplate to lookup the  
    // HelloWorldServiceProxy.  
    // The serviceID and attributes array are both null here.  
    Class[] types = {IHello.class};  
    template = new ServiceTemplate (null, types, null);  
  
}  
  
// Handle lookup service discards.  
public void discarded(DiscoveryEvent de) {}
```

Jini Example (Continued)

```
// Handle lookup service discoveries.
public void discovered(DiscoveryEvent de) {
    ServiceRegistration registration = null;
    try {
        ServiceRegistrar[] regs = de.getRegistrars();
        for (int i = 0; i < regs.length; i++) {
            IHello hwsp = (IHello) regs[i].lookup(template);
            if (hwsp == null)
                System.out.println("No matching service.");
            else
                System.out.println(hwsp.sayHello());
        }
    }
    catch (Exception e) {
        System.out.println("Exception:" + e);
    }
}
```

Jini Example (Continued)

```
public static void main(String args[]) {  
  
    try {  
        // Install a security manager.  
        if (System.getSecurityManager() == null) {  
            System.setSecurityManager(new RMISecurityManager());  
        }  
  
        // Create a LookupDiscovery object.  
        LookupDiscovery ld = new  
            LookupDiscovery(LookupDiscovery.ALL_GROUPS);  
  
        // Add a DiscoveryListener.  
        HelloWorldClient2 hwc = new HelloWorldClient2();  
        ld.addDiscoveryListener(hwc);  
    }  
}
```

Jini Example (Continued)

```
// Hang around.  
synchronized (hwc) {  
    hwc.wait();  
}  
}  
catch (Exception e) {  
    System.out.println("Exception:" + e);  
}  
}  
  
}
```