Implementation of A Distributed Application Using Jini Technology

Yiwei Wang

A Project Report

Submitted to Committee Members
Dr. W. Homer Carlisle
Dr. Kai-Hsiung Chang
Dr. Wen-chen Hu

Department of Computer Science and Engineering
Auburn University

March 18, 2000
ACKNOWLEDGEMENT
CONTENTS

Chapter 1. Introduction..............................................................................................................1

Chapter 2. Overview of Jini Connection Technology.............................................................3
  2.1. Java & Jini are excellent tools for building distributed systems.................................3
  2.2. How does Jini work?.......................................................................................................5
  2.3. Jini Architecture............................................................................................................7

Chapter 3. Jini Specification......................................................................................................9
  3.1. Introduction...................................................................................................................9
  3.2. Discovery......................................................................................................................9
  3.3. Lookup.......................................................................................................................11
  3.4. Leasing.......................................................................................................................13
  3.5. Remote Events............................................................................................................15
  3.6. Transactions...............................................................................................................17
  3.7. JavaSpaces service.......................................................................................................20
  3.8. How to develop a simple Jini service & client?...............................................................21

Chapter 4. Comparison of Jini and other distributed system architectures.........................22
  4.1. Jini & RMI..................................................................................................................22
  4.2. Jini & Corba...............................................................................................................23
  4.3. Jini & EJB..................................................................................................................24
  4.4. Evolution of distributed computing.............................................................................25

Chapter 5. Implementation of a Translation and Account Service Using Jini.......................26
  5.1. Introduction...............................................................................................................26
FIGURES

Figure 2.1 An example of a distributed Jini system.................................................4
Figure 2.2 Service registration.................................................................6
Figure 2.3 Client requests Jini services.........................................................6
Figure 2.4 Jini Architecture Segmentation......................................................7
Figure 4.1 Evolution of distributed computing..............................................25
Figure 5.1 Login GUI..............................................................................41
Figure 5.2 Login GUI when Log In incorrect..................................................41
Figure 5.3 Translation Service GUI...............................................................42
Figure 5.4 Accounts Service GUI.................................................................42
Figure 5.5 Demo of translation result.............................................................43
Figure 5.6 Result Demo of Check Balance service...........................................44
Figure 5.7 Result Demo of Deposit Service....................................................44
Chapter 1. Introduction

Jini is a set of Application Programming Interfaces (APIs) and a runtime convention that simplifies the building and deploying of distributed systems. It enables all types of distributed services and devices to work together in a community organized without extensive planning, installation and human administration. A service can be discovered and used by any other services in the community.

Jini is a network-centric computing architecture. Jini Technology is based on the Java Technology. Jini clients request services by their Java interfaces and Jini services supply code into their clients to provide the requested services. The properties of a Java virtual machine such as platform-independence, serialization, safety and security, and code downloading, enable a system like Jini in which objects can be moved around the network in a consistent and trustable manner.

Jini technology enables connections between services at any time, anywhere. Devices do not need prior knowledge of any other devices in order to participate in a community of services. Once becoming a part of this community, devices are instantly on. No one needs to install them or configure them. The network is resilient, it adapts to changes quickly. As users come and go, the community lives on. Services in the network are available on demand, whenever they are needed. Communities of Jini services are largely self-healing. Given time, the system can repair damage to itself during network or software failure. Jini’s ability to support redundant infrastructures further reduces the possibility of service unavailability, and enhances the reliability. Jini also erases the
distinction of hardware and software by publishing services on a network in the same way under the same rules regardless of the hardware or software nature of these services.

As computing evolves from mainframes to smaller, yet more powerful processors, connecting devices becomes the dominant factor in determining computing value. Jini Technology supports this trend for future computing.

Jini technology has a very wide variety of usages. Almost anything that passes digital information in and out is a candidate for connection to a Jini network. Not only traditional computer hardware and software, but also consumer appliances such as personal digital assistants (PDAs), digital cameras, VCRs, TVs, DVD players, cell phones, and CD players, can all publish their services in a Jini network and be utilized by any other users in this community. Jini technology works in the home-networking environment as well as in the enterprise and in the internet. In the home, it can help simplify the job of "system administration". In the enterprise, it can make it easier for traditional systems administrators to manage a dynamic network environment. Basically, anywhere there is a network, Jini could potentially provide the plumbing for the distributed system running on that network. It is not surprising that over 30 high-tech companies have already announced their interest in Jini. Such companies include AOL, Ericsson, IBM, Motorola, Nokia, Xerox, Bosch-Siemens Hausgerate, 3Com and Philips.
Chapter 2. Overview of Jini Connection Technology

2.1. Java & Jini are excellent tools for building distributed systems

Distributed computing allows heterogeneous components and operating systems to coordinate in harmony to achieve business tasks, and is the trend of future computing.

But distributed designs are much more difficult than standalone systems, not only because you are building multi-user services, but also because you have to consider network latency, concurrency issues, memory management (different model of memory access), inevitable partial failure, network security, network load and performance issues. Distributed systems like RPC and CORBA do not consider issues such as performance and latency as part of their programming models. Such systems actually fail to support basic requirements of robustness and reliability for distributed system. These failures have been masked in the past by the small size of the distributed systems that have been built. In the enterprise-wide distributed systems foreseen in the near future, however, such a masking will be impossible.

The Java programming language makes some strong progress on addressing the above issues in building distributed systems. The Java virtual machine has several important features: homogeneity, a single type system, serialization, code downloading, safety, and security. These properties mean that objects can be moved around the network in a more trustable manner. Java's distributed system approach, Remote Method Invocation (RMI), takes full advantage of these properties and provides a powerful tool for addressing the problems in building distributed systems. RMI uses the Java type system, but makes a clear semantic distinction between local and remote objects. It forces developers to handle the failure modes that can occur in a distributed setting, and requires
them to decide about distribution "up front" rather than hiding it as an implementation detail. RMI extends Java’s strong typing to the network domain, allows the benefits of object-oriented programming to work across the network boundary, and supports flexible and evolvable remote code.

Jini technology, built on top of a Java 2 Platform, further addresses some difficult parts of distributed computing and makes the building of distributed systems simpler, more reliable and scalable. Jini simplifies the building of distributed systems by providing APIs to enable spontaneous networks of devices and software services to assemble into working groups of objects, or federations (Figure 2.1 shows what a Jini system might be). The programming model of Jini ensures that a Jini community is self-healing when one or more devices are removed from the community, and it provides an administration-free and reliable system.

![Figure 2.1. An example of a distributed Jini system](image-url)
Jini is the approach to building robust and reliable distributed systems. These issues will be discussed in depth in Chapter 3.

2.2. How does Jini work?

In a running Jini system, there are three major components: services, clients and service locators. Service is an entity that can be used by a person, program or another service. It can be a computation, storage, a communication channel to another user, a software filter, a hardware device, or another user. Client is anything that would like to make use of the Jini service. A service locator, which is called lookup service in Jini, acts as a broker/trader/locator between services and clients. It is where services advertise their ability, and clients go to find services. There may be one or more lookup services running in a Jini system.

Code will be moved around between these three components. This is done by serializing the objects and using Java's socket support to send and receive objects. In a situation where objects in one JVM needs to invoke methods on an object in another JVM, RMI (Remote Method Invocation) is used.

When a service is booted on the network, it uses a process called discovery to find the local lookup services. The service then registers its proxy object with each lookup service and joins the community (Figure 2.2). The proxy object contains a Java object, and its types define the service it is providing.
The client, on the other hand, uses the same kind of discovery process to find the local lookup services, except that it asks for services that they want to use. Client requests services by the Java language type. When the lookup service returns the proxy object, the client will automatically download the code for that object if it doesn't have it already. (Figure 2.3)

The client issues service requests by invoking methods on the proxy object. The proxy communicates with the service as it needs to in order to execute the requests. The Jini system does not define the protocol between the proxy and its service. This can be
done in a large number of ways, such as RMI, raw sockets, IIOP or any other desired protocol.

2.3. Jini Architecture

A Jini system consists of a runtime infrastructure, a programming model, and services (Figure 2.4).

<table>
<thead>
<tr>
<th>Infrastructure</th>
<th>Programming Model</th>
<th>Services</th>
</tr>
</thead>
<tbody>
<tr>
<td>Java VM</td>
<td>Java APIs</td>
<td>JNDI</td>
</tr>
<tr>
<td>RMI</td>
<td>JavaBeans</td>
<td>Enterprise Beans</td>
</tr>
<tr>
<td>Java Security</td>
<td>...</td>
<td>JTS</td>
</tr>
<tr>
<td>Java</td>
<td>Discovery/Join</td>
<td></td>
</tr>
<tr>
<td>+ Lookup</td>
<td>Leasing</td>
<td>Printing</td>
</tr>
<tr>
<td>Jini</td>
<td>Distributed Security</td>
<td>Transactions Manager</td>
</tr>
<tr>
<td></td>
<td>+ Events</td>
<td>JavaSpaces Service</td>
</tr>
</tbody>
</table>

Figure 2.4. Jini Architecture Segmentation

Jini infrastructure defines the minimal Jini technology core. It includes a distributed security system integrated into RMI that extends the Java platform's security model to the world of distributed systems, the discovery and join protocols, and the lookup service. The infrastructure provides mechanisms for adding, locating, accessing, and removing devices and services on the network.
The Jini programming model includes the leasing interface, event and notification interfaces, and transaction interfaces. The leasing interface defines a way of allocating and freeing resources using a renewable, duration-based model. The event and notification interfaces enable event-based communication between Jini services. The transaction interfaces enable entities to cooperate in such a way that either all of the changes made to the group occur atomically or none of them occur. The programming model helps build distributed systems that are reliable.
Chapter 3. Jini Specification

3.1. Introduction

Jini's ability to support spontaneously created, self-healing communities of services is based around several key concepts: discovery, lookup, leasing, remote events, and transaction. This chapter discusses these concepts more deeply.

3.2. Discovery:

Discovery is the process by which entities that wish to participate in a distributed Jini system (known as *djinn*) locate lookup services on the network and obtain references to them. The reference obtained from the lookup service is a *ServiceRegistrar* object, and is sent back to run in the client or service by the lookup service. Its role is to act as a proxy for the lookup service. There are three discovery protocols, each designed with different purposes. *The multicast request protocol* is used by a new Jini service to call anonymous lookup services on a LAN using multicast UDP. The *Multicast announcement protocol* is used by lookup services to announce their presence on a LAN using multicast UDP. *The unicast request protocol* is used by Jini services using unicast TCP to contact lookup services that reside outside the LAN and with known IP address.

*Unicast request protocol:* Unicast discovery is done using the *LookupLocator* class located in the net.jini.core.discovery package.

```java
public class LookupLocator {
    LookupLocator(java.lang.String url)
    throws java.net.MalformedURLException;
    LookupLocator(java.lang.String host, int port);
    String getHost();
    int getPort();
    public ServiceRegistrar getRegistrar();
}
```
throws java.io.IOException,
java.lang.ClassNotFoundException;
}

Each constructor takes parameters that allow the object to determine what IP address and TCP port number it should connect to. The getRegistrar() method performs unicast discovery and returns an instance of the ServiceRegistrar object, the proxy for the specified lookup service.

*Multicast announcement protocol:* Whenever a lookup service is started or a new one becomes available after a network failure, it can use the multicast announcement protocol to announce their presence on the network. The message the lookup service sends out with an announcement contains the service ID of this lookup service, the IP address, the port number, and the list of groups it manages.

*Multicast request protocol:* Entities using this broadcast search to locate lookup services and to obtain lookup service references (ServiceRegistrar) make use of the LookupDiscovery class in the net.jini.discovery package.

```java
public class LookupDiscovery {
    LookupDiscovery(java.lang.String[] groups)
    public void addDiscoveryListener(DiscoveryListener l)
}
```

The LookupDiscovery class relies on the DiscoveryListener class:

```java
public abstract interface DiscoveryListener {
    public void discovered(DiscoveryEvent e);
    public void discarded(DiscoveryEvent e);
}
```

The LookupDiscovery class also relies upon the DiscoveryEvent class:

```java
public class DiscoveryEvent {
    public net.jini.core.lookup.ServiceRegistrar[]
    getRegistrars();
}
```
The constructor of the LookupDiscovery class takes a Java String object group as a parameter. In Jini, a group is a logic name by which a group of djinns is identified. Each lookup service has a set of zero or more groups associated with it.

Using this protocol, a multicast call is broadcast across the network. Lookup services are expected to reply to it as they receive it. Doing so may take time, and there will generally be an unknown number of lookup services that can reply. To handle this indeterminacy, the LookupDiscovery object can have a listener registered with it that is invoked as each reply comes in. The discovered() method is invoked whenever a lookup service has been discovered. The discarded() method is invoked whenever the application discards a lookup service by calling discard() on the registrar object.

The parameter to the discover() method is a DiscoveryEvent object. This has one public method, getRegistrars(), which returns an array of ServiceRegistrar objects.

3.3. Lookup

Lookup refers to the things you can do to the lookup service after you find them through discovery processes, including Jini services publishing their services and Jini clients finding the services they want to use.

As mentioned in the previous chapter, the Jini Lookup service is a fundamental part of a djinn system. It provides a central registry of services available with the system, and keeps track of the shared resources of that community. The lookup service maintains a flat collection of service items. Each service item represents an instance of a service available with the djinn. The item contains a proxy object that other participants in the
community can download to use that service, and a list of attributes that are used to
describe the service. The proxy object can be an RMI stub if the service is implemented
as a remote, or another object if the service makes use of a local proxy.

**Services join Jini communities:** Jini services join the community and publish their
services through a protocol called join. Once the Jini services join the community, the
lookup services in that community will obtain a reference to the services, which is
actually a `ServiceItem` object which have 3 fields: a `ServiceID`, a service object that
acts as proxy, and an `Entry` object to describe the attributes of the service. This join
process makes use of the `ServiceRegistrar` object obtained from its discovery.
`ServiceRegistrar` has a method called `register()` that takes a `ServiceItem` object as
a parameter.

```java
public ServiceRegistration register(ServiceItem item,
long leaseDuration)
throws java.rmi.RemoteException
```

The service proxy that resides in the lookup service can be downloaded by clients
that need to use the service. Even though a given client may know nothing about the
implementation of a particular service, it can use the service’s proxy as a “front end” for
interacting with the service. This idea of downloadable service proxies is the key idea
that gives Jini its ability to use services and devices without doing any explicit driver or
software installation. The details of how the proxy object interacts with the actual service
are completely determined by the programmer. The most used ways are: the
downloadable proxy object performs the service; the downloaded object is an RMI stub
for talking to some remote service; the downloaded object uses a private communication
protocol for talking to the service.
If a community has several lookup services, a service that needs to join the
community will typically join all the lookup services that support that community. So if
one lookup service fails, others can stand in for it. This redundancy is one big benefit of
Jini that is used to deal with partial failure.

*Clients finding a service:* Once a client discovers a lookup service and has a
reference to the lookup service, it can search all the service items to find services of
interest. Clients specify the services of interest through a ServiceTemplate object:

```java
public class ServiceTemplate {
    public ServiceID serviceID;
    public java.lang.Class[] serviceTypes;
    public Entry[] attributeSetTemplates;

    ServiceTemplate(ServiceID serviceID,
                    java.lang.Class[] serviceTypes,
                    Entry[] attrSetTemplates);
}
```

A search for services can be based on the type of the downloadable proxy object,
the unique serviceID, the attributes contained in each service item, or any combinations
you want. The ServiceRegistrar interface has a `lookup()` method to do this work:

```java
Object lookup(ServiceTemplate tmpl)
    throws java.rmi.RemoteException;
```

If there is a service item that matches the client’s search, the proxy object of that
service item will be returned to the client. The client will use the proxy downloaded from
the lookup service as a “front end” to communicate directly with the service’s “back
end”.

3.4. Leasing

Leasing is a protocol for managing resources using a renewable, duration-based
model. It is this protocol that ensures building Jini communities are stable, self-healing,
and resilient in the face of (inevitable) network failures, machine crashes, and software errors.

The traditional approach to resource management in distributed systems lets the party hold services or resources forever until this party explicitly cancels those resources, or until some human system administrators go through the logs and clean out stale services. In this approach, partial failures are not recognized and cleaned out. Over the (comparatively) long life of a distributed system, such failures may spread over the network and grow without upper bound, taking up resources and compromising the performance of the overall system. Thus, it requires explicit human intervention to administer the system.

Leasing, the time-based resource management used in Jini, provides a successful approach to avoid those problems in the traditional approach. Resources can only be held for a period time, and when the time for the lease has expired, the service ends or the resource is freed. The time period for the lease is determined when the lease is first granted. Leases may be denied by the grantor of the lease. Leases may be renewed or cancelled before they expire by the holder of the lease, but in the case of no action (or in the case of a network or participant failure), the lease simply expires. Leases can be negotiated, but the grantor has the final word on the terms of the lease that is offered.

Leases provide a consistent means to free unused or unneeded resources throughout Jini: If a service goes away, either intentionally or unintentionally, without cleaning up itself, its leases will eventually expire and the service will end. Thus, they make it very difficult to disrupt the entire system and make the persistent storage used by
the members of a Jini community virtually maintenance free. Given a bit of time, the community will identify unused resources and free them.

Leases are extensively used in lookup services. During service registration to the lookup services, it passes a long integer representing the number of milliseconds, for which the service wants its lease to last, to the register() method on the ServiceRegistrar. The lookup service responds to a type which represents the just-registered service, ServiceRegistration. One of the members of the ServiceRegistration is a Lease object. From this object, services can know whether the lookup service, which is the lease grantor, has granted the lease, and if granted, how long the duration is. The lease granter can deny the lease entirely, grant the lease for the requested duration, or grant the lease for a shorter duration. Leases are always done in terms of time duration, in order to avoid un-synchronous time problems in the distributed system.

Jini also allows a third party, an agent, to take out a lease on behalf of a lessor. The agent goes through the mechanisms of acquiring the lease, but the lease is actually in the “name” of the lessor. This approach is more convenient for the lessor, all the work concerning the lease is handed to the agent, and the lessor does not need to worry about it.

Jini also provides various packages that manage leases from the lease holder side or the lease granter side.

3.5. Remote Events
The Jini remote event programming model specifies how to allow an object in one Java virtual machine (JVM) to register interest in the occurrence of some event occurring in an object in some other JVM, and to receive a notification when an event of that kind occurs.

Remote events are very different from local events, where events are delivered locally. In a distributed environment with possible network delays or failures, remote events cannot be guaranteed to be delivered in a timely order, or even cannot be delivered at all. The time it takes to deliver remote events may be long and the cost for such delivery may be very high. Given all the delivery patterns that might happen in a distributed environment and given the wide variety of kinds of events and various ways in which interest in such events can be indicated, Jini realizes it is hard to provide all the detailed solutions for each case. Thus, it takes a generic approach which allows various degrees of assurance on delivery of a notification, supports different policies of scheduling notification, and explicitly allows the interposition of objects that will collect, hold, filter, and forward notifications. Each application makes its own decisions about what makes sense for it.

Jini typically uses events of one type, the RemoteEvent or a small number of subclasses. It does not contain complex object state information, but just enough information to allow state information to be found if needed. A remote event is

serializable and can be moved around the network to its listeners.

```java
public class RemoteEvent {
    public long getID();
    public long getSequenceNumber();
    public java.rmi.MarshalledObject getRegistrationObject();
}
```
The key interface used by objects that wish to receive events, RemoteEventListener, is an RMI Remote interface. It has only a single method, notify(), which has a RemoteEvent object as a parameter.

```java
public interface RemoteEventListener
    extends java.rmi.Remote, java.util.EventListener {
    public void notify(RemoteEvent theEvent)
        throws UnknownEventException,
                java.rmi.RemoteException;
}
```

There is no single interface that defines how to register interest in such events, such as an interface that provides an addRemoteEventListener() method. Rather, each component that may be a source of event decides the circumstances under which it will fire off events, and provides its own way for recipients to express interest.

The Jini remote event programming model is designed to allow a third party, an agent, to be used to deal with distributed events. Agents can work with any event source, regardless of what that event is or where it comes from. The third-party objects can use, forward, and store Jini remote events without having to have specialized knowledge of a class that the third parties understand; everything is simply a RemoteEvent. The third parties don’t have to understand more specific types, and they only need to implement the one notify() method to be able to receive all Jini events, now and forever. The constraints and behaviors needed to guarantee application-based event delivery can be added to the process of sending, storing, and delivering events during implementation.

The fact that the listener interface is so simple also means that agents can be composed together. By combining multiple agents together you can create a “pipeline” of event processing in which the results of one stage are forwarded on to the next.

3.6. Transactions
Transaction is a tool that allows a set of operations to be grouped in such a way that either all operations succeed, or all operations fail. From outside the transaction, these operations appear to occur simultaneously. Transactions are especially important in distributed computing, where partial failures may occur and lead to inconsistency in the system. Transactions provide a means to enforce consistency over a set of operations on one or more remote participants.

All transactions must ensure ACID properties, which are:

- Atomicity: All the operations of a transaction take place, or none of them do.
- Consistency: The completion of a transaction must leave the participants in a “consistent” state.
- Isolation: The activities of one transaction must not affect any other transactions.
- Durability: The results of a transaction must be persistent.

The two-phase commit protocol is a protocol designed to enable objects to provide ACID properties. The protocol has two phases, and all participants must go through those two phases before the transaction as a whole either succeeds or fails. In the first phase, each participant in a transaction is asked to “vote” on a transaction. In the second phase, answers from all participants are collected. If all agree to go ahead, then the transaction “commits”, which is binding on all the participants. If any “abort” during the voting stage then it forces abortion of the transaction on all participants.

Jini has adopted the syntax of the two-phase commit method to deal with transaction. It is up to the clients and services within a transaction to observe the ACID properties if they desire. Jini essentially supplies the mechanism of a two-phase commit, and leaves the policy of meaning to the participants in a transaction.
In Jini, the two-phase commit protocol is defined using three primary types: 

TransactionManager, TransactionParticipant and NestableTransactionManager.

**TransactionManager:** A transaction manager is what looks after the two-phase commit protocol for all the participants in a transaction.

The TransactionManager interface is implemented by servers that manage the two-phase commit protocol:

```java
public interface TransactionManager {
    Created create(long leaseFor) throws ...
    void join(long id, TransactionParticipant part,
              long crashCount) throws ...
    void commit(long id) throws ...
    void abort(long id) throws ...
    ...
}
```

A client obtains a reference to a TransactionManager object via a lookup service, and then creates a new transaction by invoking the manager’s `create()` method. The first time a client tells a participant to perform an operation under a given transaction, the participant must invoke the transaction manager’s `join()` method. Any of the participants may force the transaction to abort by calling `abort()`, or can force it to the two-phase commit stage by calling `commit()`.

**TransactionParticipant:** Transaction participants implement the TransactionParticipant interface.

```java
public interface TransactionParticipant ... {
    int prepare(TransactionManager mgr, long id) throws ...;
    void commit(TransactionManager mgr, long id) throws ...;
    void abort(TransactionManager mgr, long id) throws ...;
    int prepareAndCommit(TransactionManager mgr, long id) ...
}
```

These are triggered by calls made upon the transaction manager. For example, if one client calls the transaction manager to abort, then it calls all the participants to abort.
The "normal" mode of operation (that is, when nothing goes wrong with the transaction) is for a call to be made on the transaction manager to commit. It then enters the two-phase commit stage where it asks each participant to firstly `prepare()` and then to either `commit()` or `abort()`.

_NestableTransactionManager:_ Jini uses this type to support Nested transactions, also known as subtransactions.

_Mahalo:_ Mahalo is a transaction manager supplied by Sun as part of the Jini distribution.

### 3.7. JavaSpaces service

The JavaSpaces service is a special service built atop the basic Jini substrate. It provides a ubiquitous, natural way to store and use Java objects. It likes a filesystem for objects. But rather than storing simple untyped data, JavaSpaces leverages Java to store whole objects, with all the benefits that come from being a Java object: strong typing, mobile code, secure execution, and so forth. Each object stored in a JavaSpace must implement the `net.core.entry.Entry` interface. Searching for objects is based on matching entries.

Sun has two different implementations of JavaSpaces: a transient implementation and a persistent implementation. The Transient JavaSpaces service only holds on to the data that it stores as long as it is running. Persistent JavaSpaces log their stored objects to the disk, so they can recover their state after they are restarted.

Types of operations that you can do on a JavaSpace are: write a new object into a JavaSpace, read an object that's in a JavaSpace, take an object out of a JavaSpace, or ask
JavaSpace to notify you when objects that match a certain template are written into the space.

JavaSpaces assists not only in providing dynamic object persistence, but also in the design of new distributed computing patterns. Based on JavaSpace, rather than building custom, remote communication interfaces or protocols for each new distributed application, applications can be defined in terms of the set of objects they write into the tuple space and the set of objects they retrieve from the tuple space.

3.8. How to develop a simple Jini service & client?

The basic steps to register a Jini service are:

- Discover a lookup service (static port, or broadcast)
- Create information about this Jini service
- Service joins the lookup service. Passes remote interface proxy
- Renew leasing periodically

A Jini client that wishes to use a service will:

- discover a lookup service
- prepare a template for lookup search
- lookup a service
- call the service
Chapter 4. Comparison of Jini and other distributed system architectures

4.1. Jini & RMI

RMI, which stands for Remote Method Invocation, enables you to invoke methods on Java objects in remote virtual machines. RMI is used extensively in Jini, particularly its facilities for mobile code, but it is not the same thing as Jini. Jini is a set of services and conventions built atop RMI and Jini-enabled services can enjoy the full benefits of Jini’s spontaneous networking and self-healing abilities.

Table 4.1 below shows the detailed difference between Java RMI and Jini.

<table>
<thead>
<tr>
<th>Java/RMI</th>
<th>Jini</th>
</tr>
</thead>
<tbody>
<tr>
<td>The service storing information about other service providers is the RMI registry.</td>
<td>In Jini, the service storing information about other service providers is called Jini Lookup service.</td>
</tr>
<tr>
<td>RMI clients use the class Naming.Lookup() for locating the requested RMI service.</td>
<td>Jini clients use the discovery process to locate Jini Lookup services.</td>
</tr>
<tr>
<td>The RMI client must know the RMI registry host explicitly. The same rule applies to RMI servers.</td>
<td>The Jini clients search for the Jini service without any service hosting knowledge.</td>
</tr>
<tr>
<td>The approach is more rigid because the client is dependent on a particular service provider.</td>
<td>The approach is more tolerant to service provider’s faults and maximizes client independence on a particular service provider.</td>
</tr>
<tr>
<td>The RMI proxy-stub approach is strictly adhered to.</td>
<td>The Jini proxy concept is more protocol-independent because it does not rely on generated fixed-protocol stubs. The proxy fulfills requests by itself or uses either an RMI call or an internal proxy provider to fulfill a request.</td>
</tr>
<tr>
<td>No concept of built-in support for transactions, distributed events, or leasing.</td>
<td>Programming model provides for support for transactions, distributed events, and leasing</td>
</tr>
</tbody>
</table>
4.2. Jini & Corba

Common Object Request Broker Architecture (CORBA) enables you to invoke methods on remote objects written in any programming language. CORBA is perhaps more akin to RMI than Jini, but CORBA includes a service called the trader service that is somewhat reminiscent of the Jini lookup service.

CORBA defines an Interface Definition Language (IDL) to create interfaces to objects and subsequent objects can be implemented in any language. RMI/Jini is Java only. With CORBA, you can only pass object references around between machines, and the actual object and its execution is bound to the server, with remote execution mediated by the proxy stubs, while RMI/Jini allows complete objects to be passed across the net, i.e., attributes and behavior (class files). This mechanism allows 'smart proxies' to be passed, allowing the service to partition the work between the client and the service in any way the service implementers choose. CORBA has a naming service which has the same functionality as the RMI registry.

The CORBA trader service allows you to seek the remote object by type. Similarly, you can look up a Jini service by type. However, the Jini lookup service offers a bit more flexibility in searching, because you can also search by a globally unique service ID and by attributes. But the most important difference between the CORBA trader service and the Jini lookup service lies in what they return as a result of the query. The CORBA trader service returns a remote reference to a matching remote object. The Jini lookup service returns a proxy object by value. Thus, when you get a remote reference back from the CORBA trader service (assuming you have the stub definition), you can talk to the remote object by invoking methods on the local stub. The local stub
will talk across the network to the remote object via CORBA. In the case of Jini services, since the server provides both the service and the proxy which knows how to talk to it, any protocol can be used, and even CORBA can be used to communicate over the network. The implementation is completely hidden from the client.

Above the RMI and CORBA mechanism, Jini adds support for a dynamic lookup service as well as a programming model encouraging the use of distributed events and time based leases for managing server resources. These new features enable the building of truly robust, dynamic, and reliable distributed systems using Jini.

4.3. Jini & EJB:

Enterprise JavaBeans (EJB) is a component-model specification for transaction-oriented and server-based components written in Java. It provides the notion of services on the network. Enterprise beans can, and usually do, live in different address spaces.

EJB is designed to hook together legacy enterprise systems, covered by Java wrappers, to form the back-end business logic of enterprise applications. It is designed to support easy construction of this logic, and leverages the transaction, messaging, and database services already on the enterprise network. As such, EJB is largely used to configure relatively static pathways between enterprise software components. As long as the logic of the system doesn’t change, there’s probably little need to reorganize the connections between the beans. Again, defining how these connections will take place happens mostly at design time. EJB does not provide a mechanism for one application to locate another application; it relies on JNDI to perform the location services.
In contrast, Jini is about dynamic, run-time discovery of services and run-time connectivity between them. Jini technology works in the home-networking environment as well as in the enterprise environment.

4.4. Evolution of distributed computing

Distributed systems have evolved in a way such that remote more and more looks like local. Figure 4.1 below demonstrates such an evolution process.

Figure 4.1 Evolution of distributed computing
Chapter 5. Implementation of a Translation and Account Service

Using Jini Technology

5.1. Introduction

In this application, two Jini services, an English-Spanish translation service and an account service, were implemented to demonstrate the use of Jini technology. The English-Spanish translation service provides a two-way translating service to network clients. English and Spanish words were kept in an Oracle database, and the translation service accesses the database information via JDBC. The Account service is used to manage account information of both the translation service and other clients. Account information was also kept in an Oracle database and accessed via JDBC. The account service provides checking balance, deposit, and withdraw services. When requesting a translation service, clients give their account IDs to the translation service. Upon receiving a client's request and its account ID, the translation service passes the client's account ID and its own ID to the account service and calls the credit-debit method. The Service fee for using the translation service is then credited to the translation service's account and deducted from the client's account. After that, the translated word is given to the client. The Jini Transaction Manager was used to handle money transfers between accounts. In addition translation services can act as clients to use the account service's credit-debit service. All account clients can directly access the account service to check balances, make deposits and withdrawals. The translation service was implemented as an activatable service using the RMI activation framework, so that services can be activated on demand and unnecessary network resource consumption can be avoided.
5.2. Translation service

Translation service interface: This is the Translation service API, which clients will use to communicate with the service. The Translation class implements this interface with four methods.

```java
public interface Translation extends Serializable {
    public String getEnglish(String word, int clientID)
        throws ...;
    public String getSpanish(String word, int clientID)
        throws ...;
    public int getCost() throws ...;
    public void credit(int amount, int clientID,
                        TransactionManager mgr,
                        long transactionID) throws ...;
}
```

getEnglish() and getSpanish() are the actual methods that provide translation services. Using the getCost() method, clients can find out the service fee of the translation service. credit() contacts the Account service to make transactions between the Translation service account and its client account when clients use the translation service.

Translation service remote interface: This is the interface that proxy objects will use to communicate back with the translation service running on some remote machine on the network. The RemoteTranslation class does nothing except extend the Translation and Remote interfaces:
public interface RemoteTranslation extends Translation, Remote {
}

**Translation service proxy**: TranslationProxy implements the proxy class, which will be sent to clients of the Translation service. This proxy can be considered as code that mediates between the local Translation service APIs provided by the Translation interface, and the remote Translation service APIs provided by the RemoteTranslation interface. The methods of this proxy object do some simple checking locally, without having to utilize the service on remote machines.

```java
public class TranslationProxy implements Translation, Serializable {
    RemoteTranslation server = null;
    public TranslationProxy(RemoteTranslation serv) {
        this.server = serv;
    }

    public String getEnglish(String word, int clientId)
            throws java.rmi.RemoteException {
        if (server==null)
            System.err.println("server is null");
        if (word==null)
            System.err.println("word is null");
        return server.getEnglish(word, clientId);
    }

    ...
}
```
Translation service implementation: TranslationImpl provides the actual implementation of the four methods of the Translation service APIs.

In the implementation, the RMI activation framework is used, so the service is only activated on demand. The RMI activation framework will be discussed in detail later in this chapter. As a transaction participant, TranslationImpl also implements the TransactionParticipant interface:

```java
public class TranslationImpl extends Activatable
    implements Remoteltransation, TransactionParticipant {
    ...
}
```

In the constructor, three issues are addressed. First, the object is registered with the RMI activation system. Second, a SecurityManager is installed, which is needed by Jini. Third, it is connected to an Oracle database, so that it can access the database information through JDBC.

```java
public TranslationImpl(ActivationID id, MarshalledObject data)
    throws java.rmi.RemoteException {
    //Register the object with the activation system
    super(id, 0);
    // needed by Jini
    System.setSecurityManager(new RMISecurityManager());
    // connect to an Oracle database
    try{
        DriverManager.registerDriver(new
        oracle.jdbc.driver.OracleDriver());
        con =DriverManager.getConnection
            ("jdbc:oracle:thin:@sb104pc2.cse.eng.auburn.edu:1521:
            web", "internal", "oracle");
    }catch (Exception e) {
```

29
getEnglish() and getSpanish() take two parameters: a word needed to be translated and clientID. A string query is created based on the language and word that need translations. Then, string query and clientID are passed to Translate(), which accesses the Oracle database containing the corresponding English and Spanish words, and retrieves the useful information. A typical access to the database is implemented as:

```java
try {
    // Create a Statement object so we can submit SQL
    // statements to the driver
    Statement stmt = con.createStatement();
    // Submit a query, creating a ResultSet object
    ResultSet rs = stmt.executeQuery(query);
    if (rs.next())
        TranslatedWord = rs.getString(1);
    // Close the result set
    rs.close();
    // Close the statement
    stmt.close();
} catch (Exception e) {
    System.out.println(e);
}
```

The database connection is closed in the finalize() method.

```java
public void finalize() throws java.rmi.RemoteException {
    // Close the connection
    try{
```
con.close();
}

} catch(Exception e) {
    System.out.println(e);
}

Method credit() calls the Accounts Service to manage account information when clients ask for the translation service. It will first join the transaction manager, then finds the Accounts service available in the Jini community, and then calls the Accounts service to deal with the transaction.

Translation service server: The TranslationServer discovers the lookup service and joins the Jini community by registering the Translation Service. The multicast request protocol is used to locate lookup services in all groups.

This class implements the DiscoveryListener and LeaseListener interface.

Public class TranslationServer implements DiscoveryListener,
    LeaseListener {...

The constructor of this class sets up the RMI activation framework and locates lookup services.

public TranslationServer() throws
    java.rmi.activation.ActivationException,
    java.rmi.RemoteException {
    //activation framework setup
    ...
    //find lookup services
    LookupDiscovery discover = null;
    try {

discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
}

} catch(Exception e) {
    System.err.println(e.toString());
    System.exit(1);
}

discover.addDiscoveryListener(this);

The discovered() method in the DiscoveryListener interface is implemented as below:

public void discovered(DiscoveryEvent evt) {
    ServiceRegistrar[] registrars = evt.getRegistrars();
    for (int n = 0; n < registrars.length; n++) {
        System.out.println("found registrars");
        ServiceRegistrar registrar = registrars[n];
        new RegisterThread(registrar).start();
    }
}

This method deals with discovering lookup services only. Service registration is handled in an inner class in its own thread. This is because registration may take some time, and it may hinder the discovery process for other lookup services. Thus, it is preferable to do registration in a separate thread.

class RegisterThread extends Thread {
    ServiceRegistrar registrar;

    RegisterThread(ServiceRegistrar registra) {
        this.registrar = registra;
    }

    public void run() {

32
// export the proxy service

ServiceItem item = new ServiceItem(serviceID, proxy, null);

ServiceRegistration reg = null;
try {
    reg = registrar.register(item, Lease.FOREVER);
} catch(java.rmi.RemoteException e) {
    System.err.print("Register exception: ");
    e.printStackTrace();
}

if (serviceID==null)
    serviceID = reg.getServiceID();
try {
    System.out.println("service registered at " +
                        registrar.getLocator().getHost() + " with serviceID " + serviceID);
} catch(Exception e) {}

// Lease renewal
leaseManager.renewUntil(reg.getLease(),
                        Lease.FOREVER, TranslationServer.this);
}
}

In main(), the program will create an instance of a TransactionServer object. A thread will start and sleep for sometime, so when JVM finishes executing code in main(), the JVM will not exit before getting any reply from remote machines and re-registering the lease.

public static void main(String argv[]) {
    try{
new TranslationServer();
Thread.currentThread().sleep(Lease.FOREVER);

} catch (Exception e) { }

Translation service clients: TranslationClient uses the multicast request protocol to locate lookup services in all groups, and then does a lookup for the translation service. The discovery process of lookup services is the same as those implemented in the TranslationServer, except that it uses an inner class LookupThread with a separate thread to find the service using the template.

Class [] classes = new Class[] {Translation.class};
template = new ServiceTemplate(null, classes, null);
try {
    translation = (Translation) registrar.lookup(template);
} catch(java.rmi.RemoteException e) {
    e.printStackTrace();
    System.exit(2);
}

In this LookupThread, after finding a translation service, it will try to find an account service and a transaction manager, which will be responsible for dealing with transactions. As a transaction participant, LookupThread implements the TransactionParticipant interface, and implements all four methods in this interface.

public int prepare (TransactionManager mgr, long id) { ... }
public int commit (TransactionManager mgr, long id) { ... }
public int abort (TransactionManager mgr, long id) { ... }
public int prepareAndCommit(TransactionManager mgr, long id) {...}
5.3. Accounts Service

**Accounts service interface**: Accounts is the service API. It provides four kinds of services to clients.

```java
public interface Accounts {
    public void creditDebit(int amount, int creditorID, int debitorID, long transactionID, TransactionManager tm)
        throws java.rmi.RemoteException;
    public int getBalance(int accountID)
        throws java.rmi.RemoteException;
    public void withdraw(int accountID, int amount)
        throws java.rmi.RemoteException;
    public void deposit(int accountID, int amount)
        throws java.rmi.RemoteException;
}
```

`creditDebit()` is used by Translation service to make transactions. The other three methods, `getBalance()`, `withdraw()`, and `deposit()`, can be used by any client that has an account with an Accounts service. Clients can check their account balance by calling `getBalance()`, withdraw money from their account by calling `withdraw()`, and deposit money to their account using `deposit()`.

**Accounts service remote interface**:

```java
public interface RemoteAccounts extends Accounts, Remote {
}
```

**Accounts service implementation**: AccountsImpl

```java
public class AccountsImpl extends UnicastRemoteObject
```
implements RemoteAccounts, TransactionParticipant,
java.io.Serializable { ... }

Account information is saved in an Oracle database, so it is connected to the
database in the constructor and disconnected in the finalize( ) method, just as in the
TranslationImpl class.

In creditDebit( ), it joins the transaction with

// join transaction
try {
    System.out.println("Trying to join");
    mgr.join(transactionID, this, crashCount);
} catch(net.jini.core.transaction.UnknownTransactionException e) {
    e.printStackTrace();
} catch(java.rmi.RemoteException e) {
    e.printStackTrace();
} catch(net.jini.core.transaction.server.CrashCountException e) {
    e.printStackTrace();
} catch(net.jini.core.transaction.CannotJoinException e) {
    e.printStackTrace();
}

Because it is also a transaction participant, the four methods of the
TransactionParticipant interface were implemented.

The methods getBalance( ), withdraw( ), and deposit( ) just use JDBC to
retrieve and update the account information from the Oracle database.

//Retrieve data
Statement stmt = con.createStatement();
String query =
"Select AMOUNT from ACCOUNT where ACCOUNTID = " + accountId;

ResultSet rs = stmt.executeQuery(query);
if (rs.next())
    seedMoney = rs.getInt(1);
rs.close();
stmt.close();
// update data
Statement stmt=con.createStatement();
String update = "UPDATE ACCOUNT SET AMOUNT=" + seedMoney + 
    "WHERE ACCOUNTID = " + accountId;
stmt.executeUpdate(update);
stmt.close();

Accounts service server: The AccountsServer class makes use of the
JoinManager class to register the service. The JoinManager class is implemented by
Sun, and it takes care of usual tasks needed for service registration. These tasks include
finding all possible lookup services, registering the service with all lookup services,
keeping re-registering till the end of execution, and notifying all lookup services if any
changes of service state occur. There are several constructors associated with this class.
The one used in the AccountsServer class is:

JoinManager (java.lang.object obj,
    Entry[] attrSets,
    Java.lang.string[] groups,
    LookupLocator[] locators,
    ServiceIDLListener callback,
    LeaseRenewalManager leaseMgr)
The constructor of AccountsServer class is implemented as:

```java
public AccountsServer() {
    System.setSecurityManager(new RMISecurityManager());
    JoinManager joinMgr = null;
    try {
        joinMgr = new JoinManager(new AccountsImpl(),
            null, LookupDiscovery.ALL_GROUPS,
            null, this, new LeaseRenewalManager());
    } catch (Exception e) {
        e.printStackTrace();
        System.exit(1);
    }
}
```

By using the JoinManager class, the code needed to register a service is greatly reduced.

**Accounts service client:** The AccountsClient uses the multiple request protocol to locate lookup services and to find Accounts service. The implementation is the same as in the TranslationClient class.

5.4. Using RMI Activation framework in Translation service

In this application, the communication protocol used between the proxy and the back-end service is RMI. For the translation service, the RMI activation framework is used, so that this service can activated on demand.

Prior to the introduction of activation framework in Java 2, remote objects that were called from external JVMs had to be "live" (running and exported) all the time.
Activation brings a way for such objects to be “activated” (recreated as needed) on demand. The activation framework provides a way for a remote object to consume no memory or CPU resources until it is needed.

The RMI activation daemon (rmid) is the entity that handles activating objects on demand. It does this by spawning new JVMs as needed to run the objects it is activating. External JVMs are spawned as child processes of rmid, and multiple objects can share these JVMs. A group of objects that can share a single JVM, with its associated security policies and resources, is called an “activation group”.

To implement an activatable object, the implementation class extends

```
java.rmi.activation.Activatable:
```

```
public class TranslationImpl extends
    java.rmi.activation.Activatable ...
```

Then, declare a two-argument constructor in the implementation class:

```
public TranslationImpl(ActivationID id, MarshalledObject data)
    throws java.rmi.RemoteException {
    //Register the object with the activation system
    //then export it on an anonymous port
    super(id, 0);
}
```

And there is a “setup” class whose job is to create all the information necessary for the activatable class, without necessarily creating an instance of the remote object. This class is unlike the RMI server class, which must stay alive as long as the implementation needs to be made available. TranslationServer is the setup class in this example.

In the setup class, a SecurityManager is installed first:
// set RMI security manager
System.setSecurityManager(new RMISecurityManager());

Second, an ActivationGroup instance is created:

//Create a descriptor for a new activation group to run
//our object in
Properties props = new Properties();
props.put("java.security.policy",
   "c:/files/policy.all");
ActivationGroupDesc group = new
   ActivationGroupDesc(props, null);

//Register the group with activation group and get the
//ID
ActivationGroupID gid =
   ActivationGroup.getSystem().registerGroup(group);

//create the group
ActivationGroup.createGroup(gid, group, 0);

Third, an ActivationDesc instance needs to be created:

//create an activation descriptor for our object
String location = "http://beast:8081/";
MarshalledObject data=null;
ActivationDesc desc = new ActivationDesc
   ("txn.TranslationImpl", location, data);

// register with rmid
RemoteTranslation remote = (RemoteTranslation)
   Activatable.register(desc);

// make a proxy with the impl (will be made into an RMI
// stub)
proxy = new TranslationProxy(remote);
5.5. Graphic user interface (GUI)

Three GUIs are implemented, including the Log-in GUI, Translation service GUI, and Accounts service GUI. All the GUIs are implemented using Java Swing.

When the program starts, a Log-In GUI (figure 5.1) displays.

An account ID and a password are required to use the Translation and Accounts service. If the Log-in is wrong, a message saying "Log in incorrect" will show on the Log-in GUI as in figure 5.2.
If the Log-in is correct, then a translation service GUI (figure 5.3) and an account service GUI (figure 5.4) pop up. Now you can use the translation service or access your account information.
To use the Translation service, you can click on the radio button to select the original language of the word needed to be translated, type in the word you want to translate in the JTextField under the label “Original Word”, then click the “Submit” button to request the service. The result will be demonstrated in the JTextField of the translated word. A result demo is shown in Figure 5.5.
The Accounts service GUI provides three kinds of services: check balance, deposit, and withdraw. These three services are put in a JList. You can click on either one to request a service. In case of making deposits and withdrawals, the amount of money to be transferred is put in the JTextField under the label “Transferring Amount”. The service result will be displayed at the bottom of the GUI. Figure 5.6 is a result demo of the check balance service, and Figure 5.7 is the demo of the deposit service.

Figure 5.6 Result demo of the check balance service

Figure 5.7 Result demo of the deposit service
If you check your account balance both before and after you use the translation service, you will find the service fee is deducted from your account after the Translation Service.

5.6. Run the program

To run the program, you need to start web servers, rmid, a lookup service, and a transaction manager. The script file used to start these services is shown as below:

```
@echo off
rem ===========================
rem start web server
rem ===========================
echo Launching web server...
rem start web server to be used by lookup service
start java -jar -classpath .
    \files\jini1_0\lib\tools.jar -port 8080
    -dir \files\jini1_0\lib -verbose
rem start web server to be used by the application
start java -jar -classpath .
    \files\jini1_0\lib\tools.jar -port 8081
    -dir c:\files\classes -verbose

rem ===========================
rem start rmid
rem ===========================
set DIR=c:\files\tmp
echo Removing the RMID log directory...
del log\Logfile.*
del log\Snapshot.*
del log\Version_Number
rd log
echo Removing Reggie's log directory...
del reggie_log\Logfile.*
del reggie_log\Snapshot.*
del reggie_log\Version_Number
rd reggie_log
echo Removing Mahalo's log directory...
cd txn_log
del JoinAdminLog\Logfile.*
del JoinAdminLog\Snapshot.*
del JoinAdminLog\Version_number
del JoinAdminLog\service_object
del JoinAdminLog\service_id
rd JoinAdminLog
cd %DIR%
del txn_log\Version
rd txn_log
echo Starting the RMID daemon...
```
start rmid

rem ==========================
rem start Lookup Service
rem ==========================

rem ==========================
rem start Tansaction Manager
rem ==========================
java -jar -Djava.security.policy=\files\policy.all -Dcom.sun.jini.mahalo.managername=TransactionManager \files\jini1_0\lib\mahalo.jar http://sb105pc3:8080/mahalo-dl.jar \files\policy.all c:\files\tmp\txn_log public,MyGroup

Then you can run translation and account services:

set cp=...
rem start AccountServer
start java -classpath %CP%
   -Djava.rmi.server.codebase=http://sb105pc3:8081/
   -Djava.security.policy=c:\files\policy.all
   txn.TranslationServer

rem start AccountServer
start java -classpath %CP%
   -Djava.rmi.server.codebase=http://sb105pc3:8081/
   -Djava.security.policy=c:\files\policy.all
   txn.AccountsServer

and run the client:

start java -classpath %CP%
   -Djava.rmi.server.codebase=http://sb105pc3:8081/
   -Djava.security.policy=c:\files\policy.all txn.Login

This will display the Log In GUI.

5.7. Discussion

The above application is a simple demonstration of how to use Jini technology to implement a Jini community composed of two Jini services. This is a minimal
implementation, and some important issues are not considered in this implementation. An example is that the lease period is chosen to be forever, which is not practical and is a waste of network resources.
/**
 * Translation.java
 *
 * Translation service interface
 *
 */

package txn;

import java.io.Serializable;
import net.jini.core.transaction.server.TransactionManager;

public interface Translation extends Serializable {
    public String getEnglish(String word, int clientID) throws java.rmi.RemoteException;
    public String getSpanish(String word, int clientID) throws java.rmi.RemoteException;
    public int getCost() throws java.rmi.RemoteException;
    public void credit(int amount, int clientID, TransactionManager mgr, long transactionID) throws java.rmi.RemoteException;
}
/**
 * RemoteTranslation.java
 *
 * Translation service remote interface
 */

package txn;

import java.rmi.Remote;

public interface RemoteTranslation extends Translation, Remote {
}
package txn;

import java.io.Serializable;
import java.io.IOException;
import java.rmi.Naming;
import net.jini.core.transaction.server.TransactionManager;

public class TranslationProxy implements Translation, Serializable {
    RemoteTranslation server = null;

    public TranslationProxy(RemoteTranslation serv) {
        this.server = serv;
    }

    public String getEnglish(String word, int clientID)
        throws java.rmi.RemoteException {
        if (server==null) System.err.println("server is null");
        if (word==null) System.err.println("word is null");
        return server.getEnglish(word, clientID);
    }

    public String getSpanish(String word, int clientID)
        throws java.rmi.RemoteException {
        if (server==null) System.err.println("server is null");
        if (word==null) System.err.println("word is null");
        return server.getSpanish(word, clientID);
    }

    public int getCost() throws java.rmi.RemoteException {
        if (server==null) System.err.println("server is null");
        return server.getCost();
    }

    public void credit(int amount, int clientID,
            TransactionManager mgr, long transactionID)
        throws java.rmi.RemoteException {
        if (server==null) System.err.println("server is null");
    }
}
/**
 * TranslationImpl.java
 *
 * Activatable translation service implementation
 */

deprecated
package txn;

import java.sql.*;
import java.rmi.*;
import java.rmi.activation.*;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.discovery.LookupLocator;
import net.jini.core.transaction.server.TransactionManager;
import net.jini.core.transaction.server.TransactionParticipant;
import net.jini.core.transaction.server.TransactionConstants;
import net.jini.core.transaction.UnknownTransactionException;
import net.jini.core.transaction.CannotJoinException;
import net.jini.core.transaction.CannotAbortException;
import net.jini.core.transaction.server.CrashCountException;

public class TranslationImpl extends ActivatableTransactionParticipant {
    implements RemoteTranslation,

    protected Connection con;
    protected String query;
    protected TransactionManager mgr = null;
    protected Accounts accts = null;
    protected long crashCount = 1;
    private int cost = 5;
    protected static final int myID = 9999;
    protected int clientID;

    public TranslationImpl(ActivationID id, MarshalledObject data)
        throws java.rmi.RemoteException {
        //Register the object with the activation system
        //then export it on an anonymous port
        super(id, 0);
        System.out.println("Translation Service is activated");
        System.setSecurityManager(new RMISecurityManager());

        try{
            DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
            con = DriverManager.getConnection("jdbc:oracle:thin:@sb104pc2.cse.eng.auburn.edu:1521:web",
                "internal", "oracle");
        } catch (Exception e) {
            System.out.println(e);
        }
    }
}
public String getEnglish(String word, int clientID) throws java.rmi.RemoteException {
    System.out.println("Called with Spanish Word "+ word);
    query = "Select English from TRANSLATION where Spanish = "+ word +
    "");
    return Translate(query, clientID);
}

public String getSpanish(String word, int clientID) throws java.rmi.RemoteException {
    System.out.println("Called with English Word "+ word);
    query = "Select Spanish from TRANSLATION where English = "+ word +
    "");
    return Translate(query, clientID);
}

public String Translate(String query, int clientID) {
    String TranslatedWord=null;
    int clientBalance = 0;

    //check if clients have enough money to call the service
    findAccounts();
    try {
        clientBalance = accts.getBalance(clientID);
        System.out.println("ClientID is "+ clientID);
        System.out.println("Client balance is "+ clientBalance);
    }catch(java.rmi.RemoteException e) {
        e.printStackTrace();
    }

    if (clientBalance < cost){
        System.out.println("Sorry, you don't have enough money to use the service.");
    }
    else {
        try {
            // Create a Statement object so we can submit SQL statements to the driver
            Statement stmt = con.createStatement();

            // Submit a query, creating a ResultSet object
            ResultSet rs = stmt.executeQuery(query);

            if (rs.next ())
                TranslatedWord = rs.getString(1);

            // Close the result set
            rs.close();

            // Close the statement
            stmt.close();
        }catch (Exception e) {
            System.out.println(e);
        }
    }
}
return TranslatedWord;
}

public int getCost() {
    return cost;
}

public void credit(int amount, int accountID,
    TransactionManager mgr, long transactionID) {
    System.out.println("crediting");
    this.mgr = mgr;

    // join the transaction manager before findAccounts
    System.out.println("Joining txn");
    try {
        mgr.join(transactionID, this, crashCount);
    } catch(UnknownTransactionException e) {
        e.printStackTrace();
    } catch(CannotJoinException e) {
        e.printStackTrace();
    } catch(CrashCountException e) {
        e.printStackTrace();
    } catch(RemoteException e) {
        e.printStackTrace();
    }
    System.out.println("Joined txn");

    //find Accounts Service
    if (accts == null)
        findAccounts();

    if (accts == null) {
        try {
            mgr.abort(transactionID);
        } catch(UnknownTransactionException e) {
            e.printStackTrace();
        } catch(CannotAbortException e) {
            e.printStackTrace();
        } catch(RemoteException e) {
            e.printStackTrace();
        }
    }

    //call Accounts service
    try {
        accts.creditDebit(amount, accountID, myID,
            transactionID, mgr);
    } catch(java.rmi.RemoteException e) {
        e.printStackTrace();
    }
}
protected void findAccounts() {
    // find a known account service
    LookupLocator lookup = null;
    ServiceRegistrar registrar = null;

    try {
        lookup = new LookupLocator("jini://localhost");
    } catch (java.net.MalformedURLException e) {
        System.err.println("Lookup failed: " + e.toString());
        System.exit(1);
    }

    try {
        registrar = lookup.getRegistrar();
    } catch (java.io.IOException e) {
        System.err.println("Registrar search failed: " + e.toString());
        System.exit(1);
    } catch (java.lang.ClassNotFoundException e) {
        System.err.println("Registrar search failed: " + e.toString());
        System.exit(1);
    }
    System.out.println("Registrar found");

    Class[] classes = new Class[] {Accounts.class};
    ServiceTemplate template = new ServiceTemplate(null, classes, null);
    try {
        accts = (Accounts) registrar.lookup(template);
    } catch (java.rmi.RemoteException e) {
        System.out.println("Preparing...");
        return TransactionConstants.PREPARED;
    }

    public int prepare(TransactionManager mgr, long id) {
        System.out.println("Preparing...");
        return TransactionConstants.PREPARED;
    }

    public void commit(TransactionManager mgr, long id) {
        System.out.println("committing");
        try {
            System.out.println("updating DB");
            accts.deposit(myID, cost);
        }
        catch (Exception e) {} 
    }

    public void abort(TransactionManager mgr, long id) {
        System.out.println("aborting");
    }

    public int prepareAndCommit(TransactionManager mgr, long id) {
        int result = prepare(mgr, id);
    }
if (result == TransactionConstants.PREPARED) {
    commit(mgr, id);
    result = TransactionConstants.COMMITTED;
}
return result;

public void finalize() throws java.rmi.RemoteException {
    // Close the connection
    try{
        con.close();
    } catch(Exception e) {}
package txn;

import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceItem;
import net.jini.core.lookup.ServiceRegistration;
import net.jini.core.lease.Lease;
import net.jini.core.lookup.ServiceID;
import com.sun.jini.lease.LeaseRenewalManager;
import com.sun.jini.lease.LeaseListener;
import com.sun.jini.lease.LeaseRenewalEvent;
import java.rmi.RMISecurityManager;
import java.rmi.*;
import java.rmi.activation.*;
import java.util.Properties;

public class TranslationServer implements DiscoveryListener, LeaseListener {

    static final String serviceName = "TranslationService";
    protected TranslationProxy proxy;
    protected ServiceID serviceID=null;
    protected LeaseRenewalManager leaseManager = new LeaseRenewalManager();

    public static void main(String argv[]) {
        try{
            new TranslationServer();
            Thread.currentThread().sleep(Lease.FOREVER);
        }
        catch (Exception e) { }
    }

    public TranslationServer() throws java.rmi.activation.ActivationException,
            java.rmi.RemoteException {

        // set RMI security manager
        System.setSecurityManager(new RMISecurityManager());

        //Create a descriptor for a new activation group to run our object in
        Properties props = new Properties();
        props.put("java.security.policy", "c:/files/policy.all");
        ActivationGroupDesc group = new ActivationGroupDesc(props, null);

        //Register the group with activation group and get the ID
        ActivationGroupID gid = ActivationGroup.getSystem().registerGroup(group);

        //create the group

*/
ActivationGroup.createGroup(gid, group, 0);

// create an activation descriptor for our object
String location = "http://sb105pc3:8081/";
MarshalledObject data=null;
ActivationDesc desc = new ActivationDesc("txn.TranslationImpl",
location, data);

// register with rmid
RemoteTranslation remote = (RemoteTranslation)
Activatable.register(desc);

// make a proxy with the impl (will be made into an RMI stub)
proxy = new TranslationProxy(remote);

// now continue as before
LookupDiscovery discover = null;
try {
    discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
} catch( Exception e ) {
    System.err.println(e.toString());
    System.exit(1);
}

discover.addListener(this);

public void discovered(DiscoveryEvent evt) {
    ServiceRegistrar[] registrars = evt.getRegistrars();
    for (int n = 0; n < registrars.length; n++) {
        System.out.println("found registrars");
        ServiceRegistrar registrar = registrars[n];
        new RegisterThread(registrar).start();
    }
}

// an inner class to register the service in its own thread
class RegisterThread extends Thread {
    ServiceRegistrar registrar;
    RegisterThread(ServiceRegistrar registra) {
        this.registrar = registra;
    }
    public void run() {
        // export the proxy service
        ServiceItem item = new ServiceItem(serviceID, proxy, null);
        ServiceRegistration reg = null;
        try {
            reg = registrar.register(item, Lease.FOREVER);
        } catch( java.rmi.RemoteException e ) {
            System.err.print("Register exception: ");
            e.printStackTrace();
        }
        if (serviceID==null) {
            serviceID = reg.getServiceID();
        }
    }
}

} try {
    System.out.println("service registered at " + registrar.getLocator().getHost() + " with serviceID " + serviceID);
    } catch(Exception e) {
    }

    // Lease renewal
    leaseManager.renewUntil(reg.getLease(), Lease.FOREVER, TranslationServer.this);

}

public void discarded(DiscoveryEvent evt) {
    System.out.println("LookupDiscoveryListener: discarded ...");
    // Retrieve the discarded lookup service(s) from the event
    ServiceRegistrar[] regs = evt.getRegistrars();
    for(int i=0; i<regs.length;i++)
        System.out.println(" Discarded Lookup: "+regs[i]);
}

public void notify(LeaseRenewalEvent evt) {
    System.out.println("Lease expired " + evt.toString());
}

}
/**
 * TranslationClient.java
 *
 * Translation service client
 */

package txn;

import java.lang.*;
import java.lang.Integer;
import java.io.IOException;
import net.jini.core.discovery.LookupLocator;
import net.jini.discovery.LookupDiscovery;
import net.jini.discovery.DiscoveryListener;
import net.jini.discovery.DiscoveryEvent;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.lookup.ServiceID;
import net.jini.lookup.entry.Name;
import net.jini.core.entry.Entry;
import com.sun.jinilease.LeaseRenewalManager;
import net.jini.core.lease.Lease;
import java.rmi.RMISecurityManager;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import net.jini.core.transaction.server.TransactionManager;
import net.jini.core.transaction.server.TransactionConstants;
import net.jini.core.transaction.server.TransactionParticipant;

public class TranslationClient implements DiscoveryListener {

    public String word = null;
    private ServiceTemplate template;
    private Translation translation = null;
    private TransactionManager mgr = null;
    private String OriginalLanguage;
    private String aWord;
    static int clientID;
    int cost;

    public static void main(String[] args) {

        if(args.length != 3) {
            System.err.println("Usage: client.TranslationClient " +
                    "OriginalLanguage WordToBeTranslated AccountID");
            System.exit(1);
        }

        Integer account = new Integer(args[2]);
        clientID = account.intValue();
        new TranslationClient(args[0], args[1], clientID);

        // stay around long enough to receive replies
        try {
            Thread.currentThread().sleep(10000L);
        } catch(java.lang.InterruptedIOException e) {

        }
    }
}
public TranslationClient(String Origin, String Word, int accountID) {
    OriginalLanguage = Origin;
aWord = Word;
clientID = accountID;

    System.setSecurityManager(new RMISecurityManager());
    LookupDiscovery discover = null;
    try {
        discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
    } catch (Exception e) {
        System.err.println(e.toString());
        System.exit(1);
    }
    discover.addDiscoveryListener(this);
}

public void discovered(DiscoveryEvent evt) {
    ServiceRegistrar[] registrars = evt.getRegistrars();
    for (int n = 0; n < registrars.length; n++) {
        ServiceRegistrar registrar = registrars[n];
        new LookupThread(registrar).start();
    }
}

public void discarded(DiscoveryEvent evt) {
}

public class LookupThread extends Thread implements TransactionParticipant,
java.io.Serializable {
    ServiceRegistrar registrar;
    Accounts accounts = null;
    long crashCount = 1;
    int cost = 0;

    LookupThread(ServiceRegistrar registrar) {
        this.registrar = registrar;
    }

    public void run() {
        // find a translation service
        if (translation==null) {
            System.out.println("Searching for translation");
            Class[] classes = new Class[] {Translation.class};
            template = new ServiceTemplate(null, classes, null);
        }
    }
}
try {
    translation = (Translation) registrar.lookup(template);
} catch(java.rmi.RemoteException e) {
    e.printStackTrace();
}

if (translation == null)
    System.out.println("Translation Service not found");
else {
    System.out.println("Service found");
    try {
        cost = translation.getCost();
    } catch(java.rmi.RemoteException e) {
        System.out.println(e);
    }
    if(cost>20) {
        System.out.println("Costs too much: " + cost);
        translation = null;
    }
}

//try to find Accounts service
findAccounts();

// find a transaction manager
if (mgr == null) {
    Class[] classes = new Class[] {TransactionManager.class};
    ServiceTemplate template = new ServiceTemplate(null, classes, null);
    try {
        mgr = (TransactionManager) registrar.lookup(template);
    } catch(java.rmi.RemoteException e) {
        e.printStackTrace();
    }
    if (mgr == null) {
        System.out.println("Manager null");
        return;
    }
}

if (translation != null && mgr != null) {
    TransactionManager.Created tcs = null;

    // Creating transaction
    try {
        tcs = mgr.create(Lease.FOREVER);
    } catch(java.rmi.RemoteException e) {
        mgr = null;
        return;
    } catch(net.jini.core.lease.LeaseDeniedException e) {
        mgr = null;
        return;
    }
}
long transactionID = tcs.id;

// first, export ourselves since we don't extend
UnicastRemoteObject
try {
    UnicastRemoteObject.exportObject(this);
} catch (RemoteException e) {
    e.printStackTrace();
}

// join in transaction
try {
    mgr.join(transactionID, this, crashCount);
}
catch (net.jini.core.transaction.UnknownTransactionException e) {
    e.printStackTrace();
} catch (java.rmi.RemoteException e) {
    e.printStackTrace();
} catch (net.jini.core.transaction.server.CrashCountException e) {
    e.printStackTrace();
} catch (net.jini.core.transaction.CannotJoinException e) {
    e.printStackTrace();
} new LeaseRenewalManager().renewUntil(tcs.lease,
    Lease.FOREVER, null);
System.out.println("crediting...");
try {
    translation.credit(cost, clientID, mgr, transactionID);
} catch(OutOfRangeException e) {
    System.err.println(e.toString());
}
System.out.println("translating...");

try {
    if (OriginalLanguage.equalsIgnoreCase("Spanish")) {
        word = translation.getEnglish(aWord, clientID);
        System.out.println("Corresponding English Word is: " + word);
    }
    else if (OriginalLanguage.equalsIgnoreCase("English")){
        word = translation.getSpanish(aWord, clientID);
        System.out.println("Corresponding Spanish Word is: " + word);
    }
    else {
        System.out.println("Can not make translations for
the language you specified");
    }
} catch (java.rmi.RemoteException e) {
    System.err.println(e.toString());
}
if (word != null) {
    System.out.println("Translated word is " + word);
System.out.println("Calling commit");
try {
    System.out.println("mgr state " +
mgr.getState(transactionID));
    mgr.commit(transactionID);
} catch(Exception e) {
    e.printStackTrace();
}
}

else {
    try {
        mgr.abort(transactionID);
    } catch(java.rmi.RemoteException e) {
    } catch(net.jini.core.transaction.CannotAbortException e) {
        try {
            net.jini.core.transaction.UnknownTransactionException e) {
        }
    }
}

protected void findAccounts() {
    // find a known account service
    LookupLocator lookup = null;
    ServiceRegistrar registrar = null;

    try {
        lookup = new LookupLocator("jini://localhost");
    } catch(java.net.MalformedURLException e) {
        System.err.println("Lookup failed: " + e.toString());
        System.exit(1);
    }

    try {
        registrar = lookup.getRegistrar();
    } catch (java.io.IOException e) {
        System.err.println("Registrar search failed: " + e.toString());
        System.exit(1);
    } catch (java.lang.ClassNotFoundException e) {
        System.err.println("Registrar search failed: " + e.toString());
        System.exit(1);
    }
    System.out.println("Registrar found");

    Class[] classes = new Class[] {Accounts.class};
    ServiceTemplate template = new ServiceTemplate(null, classes, null);
    try {
        accounts = (Accounts) registrar.lookup(template);
    } catch(java.rmi.RemoteException e) {
        System.exit(2);
    }
}
public int prepare(TransactionManager mgr, long id) {
    System.out.println("Preparing...");
    return TransactionConstants.PREPARED;
}

public void commit(TransactionManager mgr, long id) {
    System.out.println("committing");
    try{
        accounts.withdraw(clientID, cost);
        System.out.println("DB updated");
    }
    catch(Exception e){
        e.printStackTrace();
    }
}

public void abort(TransactionManager mgr, long id) {
    System.out.println("aborting");
}

public int prepareAndCommit(TransactionManager mgr, long id) {
    int result = prepare(mgr, id);
    if (result == TransactionConstants.PREPARED) {
        commit(mgr, id);
        result = TransactionConstants.COMMITTED;
    }
    return result;
}

} //LookupThread

} //TranslationClient
/**
 * Accounts.java
 *
 * Accounts service interface
 */

package txn;

import net.jini.core.transaction.server.TransactionManager;

public interface Accounts {

    void creditDebit(int amount, int creditorID, int debtorID,
                      long transactionID, TransactionManager tm)
            throws java.rmi.RemoteException;

    public int getBalance(int accountID) throws java.rmi.RemoteException;

    public void withdraw(int accountID, int amount) throws
                         java.rmi.RemoteException;

    public void deposit(int accountID, int amount) throws
                       java.rmi.RemoteException;

}
/**
 * RemoteAccounts.java
 *
 * Accounts service remote interface
 */

package txn;

import java.rmi.Remote;

public interface RemoteAccounts extends Accounts, Remote {
}
package txn;

import java.sql.*;
import net.jini.core.transaction.server.TransactionManager;
import net.jini.core.transaction.server.TransactionParticipant;
import net.jini.core.transaction.server.TransactionConstants;
import java.rmi.server.UnicastRemoteObject;
import java.util.Hashtable;
import net.jini.core.lookup.ServiceTemplate;
import net.jini.core.lookup.ServiceRegistrar;
import net.jini.core.discovery.LookupLocator;

public class AccountsImpl extends UnicastRemoteObject
    implements RemoteAccounts, TransactionParticipant, java.io.Serializable {

    protected Connection con;
    protected long crashCount = 1;

    public AccountsImpl() throws java.rmi.RemoteException {
        try{
            DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
            con = DriverManager.getConnection("jdbc:oracle:thin:@sb104pc2.cse.eng.auburn.edu:1521:web",
                "internal", "oracle");
        }catch (Exception e) {
            System.out.println(e);
        }
    }

    public void creditDebit(int amount, int creditorID, int debtorID, long transactionID, TransactionManager mgr) {

        java.rmi.Remote stub = null;
        try {
            stub = toStub(this);
        } catch(Exception e) {
            System.out.println("To stub failed");
            e.printStackTrace();
        }
        System.out.println("To stub found");
        String annotate = java.rmi.server.RMIClassLoader.getClassAnnotation(stub.getClass());
        System.out.println("from " + annotate);
        try {
            Class cls = java.rmi.server.RMIClassLoader.loadClass(annotate,
                "txn.AccountsImplStub");
        } catch(Exception e) {
            System.out.println("To stub class failed");
        }
    }
}
public void createAccount(int accountID, int amount)
        throws java.rmi.RemoteException {
    try {
        Statement stmt = con.createStatement ();
        String update =
            "insert into ACCOUNT values (" + accountID + "," +
            amount + ")";
        stmt.executeUpdate (update);
        stmt.close();
    }
        catch (Exception e) {}
    }
*/

public int getBalance(int accountID) throws java.rmi.RemoteException {
    int seedMoney = 0;
    try {
        Statement stmt = con.createStatement ();
        String query =
            "Select AMOUNT from ACCOUNT where ACCOUNTID = " + accountID;
        ResultSet rs = stmt.executeQuery (query);
        if (rs.next ())
            seedMoney = rs.getInt (1);
        else {
            System.out.println("This account does not exist, you have to create an account first");
            System.exit (1);
        }
        rs.close();
        stmt.close();
    }
    catch (Exception e) {
        System.out.println (e);
    }
}
public void withdraw(int accountID, int amount) throws java.rmi.RemoteException {
    int seedMoney = 0;

    // obtain the seedMoney of the account
    try {
        Statement stmt = con.createStatement();
        String query =
            "Select AMOUNT from ACCOUNT where ACCOUNTID = " + accountID;
        ResultSet rs = stmt.executeQuery(query);
        if (rs.next ())
            seedMoney = rs.getInt(1);
        rs.close();
        stmt.close();
    }
    catch (Exception e) {
        System.out.println(e);
    }
    System.out.println("Current balance is" + seedMoney);
    seedMoney = seedMoney - amount;

    // update the Money
    try{
        Statement stmt=con.createStatement();
        String update = "UPDATE ACCOUNT SET AMOUNT=" + seedMoney + "WHERE ACCOUNTID = " + accountID;
        int n=stmt.executeUpdate(update);
        System.out.println(n + " rows updated");
        stmt.executeUpdate(update);
        stmt.close();
    }
    catch (Exception e) {
        System.out.println(e);
    }
}

public void deposit(int accountID, int amount) throws java.rmi.RemoteException {
    int seedMoney = 0;

    // obtain the seedMoney of the account
    try {
        Statement stmt = con.createStatement();
        String query =
            "Select AMOUNT from ACCOUNT where ACCOUNTID = " + accountID;
        ResultSet rs = stmt.executeQuery(query);
        if (rs.next ())
            seedMoney = rs.getInt(1);
        rs.close();
stmt.close();
}
catch (Exception e) {
    System.out.println(e);
}

seedMoney = seedMoney + amount;
System.out.println(\"Current balance is\" + seedMoney);

//update the Money
try{
    Statement stmt=con.createStatement();
    String update = \"UPDATE ACCOUNT SET AMOUNT=\" + seedMoney + \"WHERE ACCOUNTID = \" + accountID;
    int n=stmt.executeUpdate(update);
    System.out.println(n + \" rows updated\");
    stmt.executeUpdate(update);
    stmt.close();
} catch (Exception e) {
    System.out.println(e);
}

public int prepare(TransactionManager mgr, long id) {
    System.out.println(\"Preparing...\");
    return TransactionConstants.PREPARED;
}

public void commit(TransactionManager mgr, long id) {
    System.out.println(\"commiting\");
}

public void abort(TransactionManager mgr, long id) {
    System.out.println(\"aborting\");
}

public int prepareAndCommit(TransactionManager mgr, long id) {
    int result = prepare(mgr, id);
    if (result == TransactionConstants.PREPARED) {
        commit(mgr, id);
        result = TransactionConstants.COMMITTED;
    }
    return result;
}

public void finalize() throws java.rmi.RemoteException {
    try{
        con.close();
    } catch(Exception e) {}
/**
 * AccountsServer.java
 * Accounts service server
 */

package txn;

import com.sun.jini.lookup.JoinManager;
import net.jini.core.lookup.ServiceID;
import com.sun.jini.lookup.ServiceIDListListener;
import com.sun.jini.lease.LeaseRenewalManager;
import net.jini.discovery.LookupDiscovery;
import java.rmi.RMISecurityManager;

public class AccountsServer implements ServiceIDListListener {

    public static void main(String argv[]) {
        new AccountsServer();

        try {
            Thread.currentThread().sleep(1000000L);
        } catch(java.lang.InterruptedIOException e) {

        }
    }

    public AccountsServer() {

        System.setSecurityManager(new RMISecurityManager());
        JoinManager joinMgr = null;
        try {
            joinMgr = new JoinManager(new AccountsImpl(),
                                     null,
                                     LookupDiscovery.ALL_GROUPS,
                                     null,
                                     this,
                                     new LeaseRenewalManager());
        } catch(Exception e) {
            System.out.println(e);
        }
    }

    public void serviceIDNotify(ServiceID serviceID) {
        System.out.println("got service ID " + serviceID.toString());
    }
}


package txn;

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

public class AccountClient implements DiscoveryListener {

    private ServiceTemplate template;
    private Accounts account;
    private String service;
    private int accountid;
    private int transfer;
    public int seedMoney;


    public static void main(String[] args) {
        if(args.length !=3) {
            System.err.println("Usage: client.AccountClient " +
                    "Service AccountID Money");
            System.exit(1);
        }

        int account = Integer.valueOf(args[1]).intValue();
        int money = Integer.valueOf(args[2]).intValue();
        new AccountClient(args[0], account, money);

        try {
            Thread.currentThread().sleep(100000L);
        } catch(java.lang.InterruptedException e) {
        }
    }

    public AccountClient(String serv, int account, int money) {
        service = serv;
        accountid = account;
        transfer = money;

        System.setSecurityManager(new RMISecurityManager());
        LookupDiscovery discover = null;
        try {
            
        }
    }
}
discover = new LookupDiscovery(LookupDiscovery.ALL_GROUPS);
}
catch (Exception e) {
    System.err.println(e.toString());
}
discover.addDiscoveryListener(this);

public void discovered (DiscoveryEvent evt) {
    ServiceRegistrar[] registrars = evt.getRegistrars();
    Class[] classes = new Class[] {Accounts.class};
    account = null;
    template = new ServiceTemplate(null, classes, null);
    for (int n = 0; n < registrars.length; n++) {
        ServiceRegistrar registrar = registrars[n];
        new LookupThread(registrar).start();
    }
}

class LookupThread extends Thread {
    ServiceRegistrar registrar;
    LookupThread (ServiceRegistrar registrar) {
        this.registrar = registrar;
    }
    public void run() {
        try {
            account = (Accounts) registrar.lookup(template);
        } catch (java.rmi.RemoteException e) {
            System.out.println(e);
        }
        if (account == null)
            System.out.println("Accounts Service not found");
        else
            System.out.println("Service found");
        try {
            if (service.equalsIgnoreCase("CheckBalance")) {
                seedMoney = account.getBalance(accountid);
                System.out.println("account balance is: " + seedMoney);
            } else if (service.equalsIgnoreCase("Deposit")) {
                account.deposit(accountid, transfer);
                System.out.println("Processing succeed");
            } else {
                account.withdraw(accountid, transfer);
                System.out.println("Processing succeed");
            }
        } catch (java.rmi.RemoteException e) {
            System.err.println(e.toString());
        }
    }
}
public void discarded(DiscoveryEvent evt) {
}
}
/*
 * TranslationFrame.java
 *
 * Translation service GUI
 */

class TranslationFrame extends JFrame implements ActionListener {

    private JLabel title;
    private JTextField originW;
    private JTextField tranW;
    private JRadioButton englishButton;
    private JRadioButton spanishButton;
    private JButton okButton;
    private JButton cancelButton;
    int ID;

    public TranslationFrame(int accountid) {
        ID = accountid;
        setTitle("Translation Service");
        setSize(400, 300);
        setLocation(200, 200);
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });

        Container contentPane = getContentPane();

        JPanel P1 = new JPanel();
        P1.setBorder(BorderFactory.createEmptyBorder(10, 30, 20, 30));
        title = new JLabel("English-Spanish Translation Service");
        title.setFont(new Font("SansSerif", Font.BOLD, 16));
        P1.add(title);
        P1.add(new JLabel("You will be charged 5c per Service"));
        contentPane.add(P1, "North");

        JPanel P2 = new JPanel();
        P2.setBorder(BorderFactory.createEmptyBorder(30, 40, 30, 50));
        P2.add(new JLabel("Original Language "));
        englishButton = new JRadioButton("English", true);
        spanishButton = new JRadioButton("Spanish", false);
        ButtonGroup group = new ButtonGroup();
        group.add(englishButton);
        group.add(spanishButton);
        P2.add(englishButton);
        P2.add(spanishButton);
        P2.add(new JLabel("Original Word: "));
P2.add(originW = new JTextField("", 10));
P2.add(new JLabel("Translated Word: "));
P2.add(tranW = new JTextField("", 10));
contentPane.add(P2, "Center");

JPanel P3 = new JPanel();
okButton = new JButton(" OK ");
P3.add(okButton);
okButton.addActionListener(this);
cancelButton = new JButton(" Cancel ");
P3.add(cancelButton);
cancelButton.addActionListener(this);
contentPane.add(P3, "South");

public void actionPerformed(ActionEvent evt) {
    Object source = evt.getSource();
    if (source == okButton) {
        String word = originW.getText().trim();
        String language;
        if (englishButton.isSelected())
            language = "English";
        else
            language = "Spanish";
        System.out.println("Original Language is :" + language);
        System.out.println("Word is: " + word);
        TranslationClient TC = new TranslationClient(language, word, ID);
        try {
            Thread.currentThread().sleep(5000L);
        } catch (java.lang.InterruptedException e) {
        }
        System.out.println("word is: " + TC.word);
        tranW.setText(TC.word);
    } else if (source == cancelButton) {
        originW.setText(" ");
        tranW.setText(" ");
    }
}

public class TranslationGUI {
    public static void main(String[] args) {
        JFrame frame = new TranslationFrame(1111);
        frame.show();
    }
} */
package txn;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.sql.*;

class LoginFrame extends JFrame implements ActionListener {

    private JTextField accountid;
    private JPasswordField password;
    private JButton okButton;
    private JButton cancelButton;
    private JLabel label;

    public LoginFrame() {
        setTitle("Log In");
        setSize(300, 200);
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });

        Container contentPane = getContentPane();

        JPanel P1 = new JPanel();
        P1.setBorder(BorderFactory.createEmptyBorder(20, 30, 10, 30));
        P1.add(new JLabel("Account ID:"));
        P1.add(accountid = new JTextField("", 10));
        P1.add(new JLabel("Password:"));
        P1.add(password = new JPasswordField("", 10));
        label = new JLabel("Log in incorrect");
        P1.add(label);
        label.setVisible(false);
        contentPane.add(P1, "Center");

        JPanel P2 = new JPanel();
        okButton = new JButton(" OK ");
        P2.add(okButton);
        okButton.addActionListener(this);
        cancelButton = new JButton(" Cancel ");
        P2.add(cancelButton);
        cancelButton.addActionListener(this);
        contentPane.add(P2, "South");
    }

    public void actionPerformed(ActionEvent evt) {
        Object source = evt.getSource();
if (source == okButton) {
    String id = accountid.getText().trim();
    int num = Integer.valueOf(id).intValue();
    char[] pwd = password.getPassword();
    if(validate(id, pwd)) {
        label.setVisible(false);
        System.out.println("right");
        JFrame frame2 = new TranslationFrame(num);
        frame2.show();
        JFrame frame3 = new AccountFrame(num);
        frame3.show();
    }
    else {
        System.out.println("wrong password");
        label.setVisible(true);
    }
} else if (source == cancelButton) {
    accountid.setText("*");
    password.setText("*");
}

public boolean validate(String id, char[] password) {
    boolean valid = false;
    int number = Integer.valueOf(id).intValue();
    try {
        DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
        Connection con = DriverManager.getConnection("jdbc:oracle:thin:@sb104pc2.cse.eng.auburn.edu:1521:web", "internal", "oracle");
        Statement stmt = con.createStatement();
        String query = "SELECT PASSWORD FROM ACCOUNT WHERE ACCOUNTID = " + number;
        ResultSet rs = stmt.executeQuery(query);
        String pswd = null;
        while (rs.next()) {
            pswd = rs.getString(1);
            if (pswd.equalsIgnoreCase(new String(password)))
                valid = true;
        }
        rs.close();
        stmt.close();
        con.close();
    }
    catch (Exception e) {
        System.out.println(e);
    }
    return valid;
}

public class Login {
    public static void main(String[] args) {
}
JFrame frame = new LoginFrame();
frame.show();
}
/*
 * AccountFrame.java
 *
 * Accounts service GUI
 */

package txn;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.event.*;
import java.sql.*;
import java.lang.*;

class AccountFrame extends JFrame implements ActionListener {

    private JButton okButton;
    private JButton cancelButton;
    private JList service;
    private JTextField money;
    private JTextField result;
    int ID;

    public AccountFrame(int accountid) {
        ID = accountid;
        setTitle("Account Service");
        setSize(300, 200);
        setLocation(600, 200);
        addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });

        Container contentPane = getContentPane();
        GridBagLayout gbl = new GridBagLayout();
        contentPane.setLayout(gbl);

        service = new JList(new String[] {
            "Check Balance", "Deposit", "Withdraw"});
        service.setSelectedIndex(0);

        JLabel label = new JLabel("Transferring Amount: ");
        money = new JTextField("", 5);

        okButton = new JButton(" Submit ");
        cancelButton = new JButton(" Cancel ");

        result = new JTextField();
        result.setEditable(false);

        GridBagConstraints gbc = new GridBagConstraints();
        gbc.fill = GridBagConstraints.BOTH;
        gbc.weightx = 0;
        gbc.weighty = 100;
        add(service, gbc, 0, 0, 1, 3);
```java
gbc.weightx = 100;
gbc.fill = GridBagConstraints.NONE;
gbc.anchor = GridBagConstraints.CENTER;
add(label, gbc, 1, 0, 2, 1);
add(money, gbc, 1, 1, 2, 1);
add(okButton, gbc, 1, 2, 1, 1);
gbc.fill = GridBagConstraints.HORIZONTAL;
add(cancelButton, gbc, 2, 2, 1, 1);
gbc.anchor = GridBagConstraints.SOUTH;
gbc.weighty = 0;
add(result, gbc, 0, 3, 4, 1);
result.setText("Processing result is displayed here ");

//service.addListSelectionListener(this);
okButton.addActionListener(this);
cancelButton.addActionListener(this);
}

public void actionPerformed(ActionEvent evt) {
    Object source = evt.getSource();
    if (source == okButton) {
        String m = money.getText().trim();
        System.out.println("Money is: " + m);
        int index = service.getSelectedIndex();
        if (index == 0) {
            System.out.println("Call check Balance");
            AccountClient ac = new AccountClient("CheckBalance", ID, 0);
            try {
                Thread.currentThread().sleep(5000L);
            } catch (java.lang.InterruptedIOException e) {
            }
            int mon = ac.seedMoney;
            result.setText("Your Balance is: " + mon);
            System.out.println("seedMoney is: " + mon);
        }
        else if (index == 1) {
            int my = Integer.valueOf(m).intValue();
            AccountClient ac = new AccountClient("Deposit", ID, my);
            try {
                Thread.currentThread().sleep(5000L);
            } catch (java.lang.InterruptedIOException e) {
            }
            result.setText("Processing Succeed");
            System.out.println("Call Deposit");
        }
        else if (index == 2) {
            int my = Integer.valueOf(m).intValue();
            AccountClient ac = new AccountClient("Withdraw", ID, my);
            try {
                Thread.currentThread().sleep(5000L);
            } catch (java.lang.InterruptedIOException e) {
            }
            result.setText("Processing Succeed");
            System.out.println("Call Withdraw");
        }
    }
```
else if (source == cancelButton) {
    money.setText("");  
    result.setText("Processing Result: ");
}

public void add(Component c, GridBagConstraints gbc,  
    int x, int y, int w, int h) {
    gbc.gridx = x;
    gbc.gridy = y;
    gbc.gridwidth = w;
    gbc.gridheight = h;
    getContentPane().add(c, gbc);
}

} /*
public class AccountGUI {
    public static void main(String[] args) {
        JFrame frame = new AccountFrame(accountId);
        frame.show();
    }
} */