6th International Conference on Applied Informatics
Eger, Hungary, January 27–31, 2004.

Mobile agents for the database management using Jini

Fabrice Mourlin, Jaouad Skaita

LACL University Paris 12, France
fabrice.mourlin@wanadoo.fr, skaita@univ-paris12.fr

Abstract

The management of large database application need complex strategies. The use of mobile agents is powerful, and provides a reliable approach. The Java language provides an ideal implementation platform, furnishing tools that help streamline complex software applications. Java's Jini framework facilitates mobile agent application development, providing key features for distributed network programming. We describe our technical approach of the distributed database management using Jini framework as a foundation for mobile agents. The presentation of this paper follows a step by step presentation which is close to training. The example which is the kernel of that work, can also be used as a base course on mobile programming course. The subject of the case study is a collect of SQL statement on several hosts on a local network which have to be applied onto a specific database.

1. Introduction

The concept of mobile agents is frequently used in today's software applications—such that e-commerce to network management to data warehousing. Mobile agent developers implement these solutions in Java language for several reasons: First, Java's built-in object-oriented language features are conducive to agent technology. Second, developers can be extremely productive using Java. Essentially, Java provides tools that simplify and manage complex software development tasks. Jini is one of Java APIs (Application Programming Interface) that allow designers to build distributed applications easily. Our work introduces a mobile agent framework based on the Jini architecture for the management of distributed database. Jini (Java Intelligent Network Interface) provides a powerful set of packages in a Java developer's toolbox. It automates and abstracts distributed applications' underlying details. These details include the low-level functionality (socket communication, synchronization, and so on) necessary to implement the high-level abstractions (such as service registration, discovery, and use) that Jini provides.

First we present what is an agent for our framework, and how its mobile features are taken into account. Secondly, we describe the power of mobile agent and how they move all around a network. Then we describe the service which are essential to our mobile framework.

2. What is an Agents in our framework ?

Agent means a robot which possesses a high-level functionality (in UML, Unified Modelling Language, it will be a use cas). Classically, the features of an agent are identified depending on the application domain *[1]* which is concerned such as intrusion detection, parallel computing, image rendering,or database management:

- Reactive : it receives stimuli and answer without any help,
- Autonomous : the most part of the resources are contained inside the agent itself,
- Goal-oriented : in an agent oriented application, each of them has its own goal or its own set of rules, it has to apply them to the node where it is,
- Temporally continuous: an agent is not a component which is waiting for a new mission and managed by a server of components; but a piece of code which answers continuously to its environment,
- Communicative: an agent can send data to another one and this data can be a continuous flow,
- Intelligent: a precise strategy can be implemented in an agent depending on the previous it did. Very often, a set of command is prepared and some of them are assigned to agent: it is a kind of mission,
- Mobile: dynamically, an agent can navigate through a set of node in a large network.

In the context of our work, we can consider an agent as a software entity that exhibits some combination of the previous properties.

What can we use mobile agents for? Mobile agents come in a variety of flavors and perform numerous functions:

- An **information agent** searches for information residing on remote nodes and reports back to the source : it is used for computing a diagnostic,
- A **computation agent** seeks under used network resources to perform CPU-intensive processing functions : it is powerful for a load balancing algorithm,
- A **communication agent** can send messages back and forth between clients residing on various network nodes :

In our framework, the mobility is an essential skill and this is possible because the agent are loosely coupled. All mobile agent in its mission consist of two main components.

1. the mobile agents themselves; that are, entities with some job to do.

2. the mobile agent host(s), the service that provides the mobile agents' execution platform.

In a distributed environment, we can have one-to-many agent hosts as well as one-to-many agents. To be an active agent platform, a given node in the system must have at least one active agent host. Figure 1 describes the framework components.

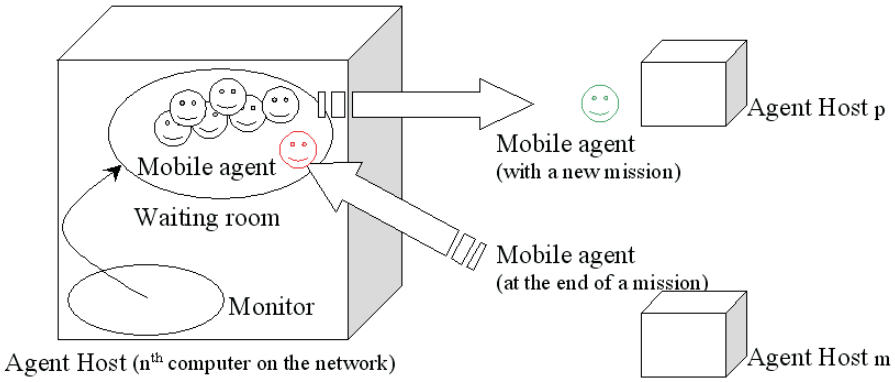


Figure 1 : software architecture of our framework

This component scheme can be mapped easily to the Jini model. Jini, at the highest level, provides the infrastructure that enables clients to discover and use various services. Jini also provides a programming model for developers of Jini clients and services. In the context of this mobile agent framework, the agent host(s) provides Jini collect services. The mobile agent(s) is the Jini client. It can have a lease for on particular node and this one can be modify during the action of the agent. This collect service provides a set of SQL statements which have to be executed later on a database.

Jini services register with one or more Jini lookup services by providing a service proxy for perspective clients. In turn, clients query the lookup service(s) for particular services, figure 2 depicts that process.

The first step in building an agent host is to create a remote interface, the service template that agents will look for via the Jini lookup service. The `AgentHostRemoteInterface` provides one method, `acceptAgent()`, which agents call to travel to the implementing agent host:

```
public interface AgentHostRemoteInterface extends Remote
{
      public void acceptAgent (AgentInterface ai) throws RemoteException;
}
```

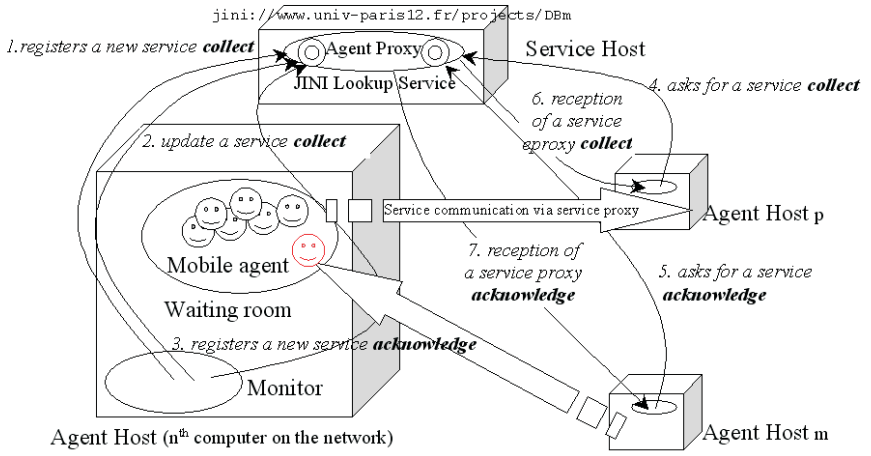


Figure 2 : registration and discovery of services

Because a class can implement several interfaces, a same objects can publish several interfaces, that means, provide multiple services. For instance, if we had a distributed data warehouse, we might have an agent host that provides a local data access service. In this instance, a data-mining agent might look for a host that provides the data access service and move to that host to perform localized mining operations. Therefore, we can have agents with different missions share hosts that provide multiple services.

The second phase in building the agent host is to provide an implementation of this remote interface that is the actual Jini service. The `MobileAgentHost` class implements the `AgentHostRemoteInterface`.

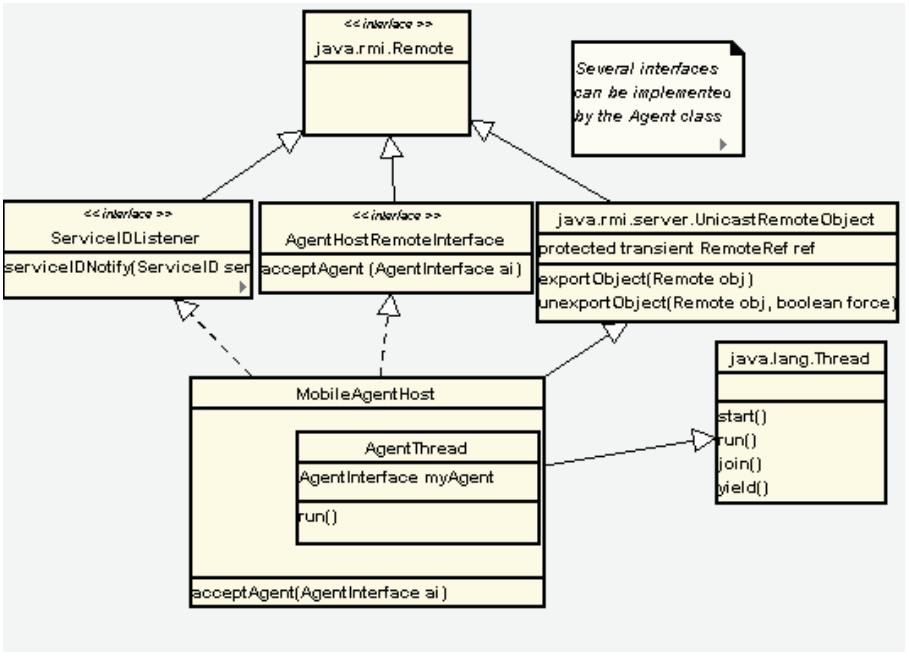


Figure 3: class diagram for the description of a mobile agent host.

The class extends the `UnicastRemoteObject` class of the `java.rmi.server` package, which allows clients to obtain a remote reference and call its methods during a collect of data. The `MobileAgentHost` also implements the `ServiceIDListener` interface, which is passed a unique `ServiceID` object via the `serviceIDNotify()` method when the service first registers with a Jini lookup service. The `MobileAgentHost` constructor is shown below:

```
public class MobileAgentHost extends UnicastRemoteObject
                             implements AgentHostRemoteInterface, ServiceIDListener
{
       private JoinManager myJoinManager = null;
       private ServiceID myServiceID = null;
       private LeaseRenewalManager myLeaseManager = null;
       private Object myAgentObject = null;
       private LookupDiscoveryManager myLDM = null;

       public MobileAgentHost(Object agentObject)
                             throws IOException
       {
              myAgentObject = agentObject;
              myLDM = new LookupDiscoveryManager(LookupDiscovery.ALL_GROUPS, null, null);
              myJoinManager = new JoinManager(this,   null,
                     this, myLDM, new LeaseRenewalManager());
       }
}
```

Specifies a lookup locator to perform unicast discovery at a known host/port.

Publish service to all groups

This service

Attribute sets

DiscoveryListener for callbacks when a lookup service is discovered or discarded. This service registration will be handled by the JoinManager.

ServiceIDListener implementor, which is notified when a serviceID is assigned.

DiscoveryManagement implementor

LeaseRenewalManager

This `MobileAgentHost` constructor saves the `agentObject` object reference stored as member data and passed to arriving agents via the `collect()` method. The constructor itself performs 2 mains functions :

1. it creates a `LookupDiscoveryManager` to locate a Jini lookup service(s).
2. it creates a `JoinManager`, with the `LookupDiscoveryManager` as a parameter, to add this lookup service to the Jini service federation.

In the `acceptAgent()` method's implementation, the `MobileAgentHost` binds an incoming agent to an `AgentThread`: this is a way to keep autonomy for all the agents

```
public void acceptAgent(AgentInterface ai) throws RemoteException
{
       AgentThread at = new AgentThread(ai);
       at.start();
}
```

In turn, an inner class instance, `AgentThread`, is created to run the bounded agent by calling its `collect()` method, passing the `LookupDiscoveryManager` and agent object references. This `collect()` method contains the specific action the agent has to do, it means operations on database using JDBC (Java DataBase Connectitity)

```
private class AgentThread extends Thread
{
      private AgentInterface myAgent = null;
      AgentThread(AgentInterface ai)
      {
            myAgent = ai;
      }
      public void run()
      {
            myAgent.collect(myLDM, myAgentObject);
      }
}
```

In this implementation, a new thread is created for each arriving agent. It is possible to enhance the implementation by binding incoming agents to an AgentThread from a pre-existing thread pool.

3. How work the mobile agents

The construction of mobile agents. The 1^{st} step in building an agent is to create an interface for agents. For this, we create an `AgentInterface` that extends the `Serializable` interface. The `Serializable` interface marks the class as a serializable entity, or one that can be sent across the wire, this is essential for the navigation through the network :

```
public interface AgentInterface extends Serializable
{
      public void collect(LookupDiscoveryManager ldm, Object dbRef);
}
```

The `AgentInterface` consists of a `collect()` method that is called when an agent arrives on a mobile agent host. This method takes 2 parameters:

1. a reference to the `LookupDiscoveryManager` maintained by the current host. The agent uses this reference if it decides to look for new service providers, such as when it wants to travel to a new mobile agent host for the next management operation.

2. an `dbRef` parameter, which contains data necessary for the agent to complete its action. For example, an agent that must communicate with other agents currently residing on this host might be passed a collection of agent references. A data-mining agent might be passed a reference to a local database.

An implementation of this interface is the abstract `MobileAgent` class. This class's constructor builds a service template that locates services of type `MobileAgentHostInterface`. It also provides 3 additional methods:

- `collect()`:to perform the task of gathering SQL statements, subclasses override the abstract `collect()` method. Each agent host has its own SQL files with its own priorities, also, there is one sub class per kind of agent host.

- `moveToRandomHost()`: when the agent wants to move, subclasses call the `moveToRandomHost()` method, which performs 3 steps:

    1. gets a list of the currently available mobile agent hosts with a call to `getMobileAgentHosts()`.
    2. randomly selects a host from the list of all the agent hosts, it is necessary to visit.
    3. moves to a new mobile agent host by calling the `acceptAgent()` method. If the call on the selected host fails, select a new mobile agent host.

- `getMobileAgentHosts()`: to obtain a list of currently available agent hosts, subclasses call the `getMobileAgentHosts()`. It requires the following steps which is a classical JINI pattern:

    1. Call `getRegistrars()` to obtain a current list of lookup services.
    2. Iterate through each lookup service to find services that match the desired template; in this case, `AgentHostRemoteInterfaces`. When the task is complex or when the amount of data is quite important, it could be useful to duplicates the same agent host registered with multiple lookup services. The `myMAHServiceTemplate` object, a `ServiceTemplate` class instance, passed to the `lookup()` method initialises in the `MobileAgent` constructor.

3. Add each matching service to a vector of `AgentHostRemoteInterfaces`. This part is essential because it keep the track of all the move of agents and then it is possible to visualize agent route map.

```
private static Logger logger = Logger.getLogger("dbms.mobile.mobileAgent");
private Vector getMobileAgentHosts(LookupDiscoveryManager ldm)
{
    // Find available lookup services
    ServiceRegistrar[] lookupServices = ldm.getRegistrars();
    // Find matching services called AgentHostRemoteInterfaces.
    Vector mobileAgentHosts  = new Vector(HOST_MAX);
    ServiceMatches agentHosts= null;
    for (int index = 0; index < lookupServices.length; index++)
    {
        try // Register with each lookup service.
        {
            agentHosts=lookupServices[index].lookup(myMAHServiceTemplate, Integer.MAX_VALUE);
        }
        catch (RemoteException re)
        {
            logger.fine(re.getMessage());
            continue;
        }
        for (int index2 = 0; index2 < agentHosts.totalMatches; index2++)
            mobileAgentHosts.addElement(agentHosts.items[index2].service);
    }
    return mobileAgentHosts;
}
```

An interaction diagram can model the sequence of events between the `MobileAgent`, `MobileAgentHost`, and the Jini lookup service which can be on a computer far from the local network. All the error message are saved into a log file which can be parsed in a post mortem analysis.

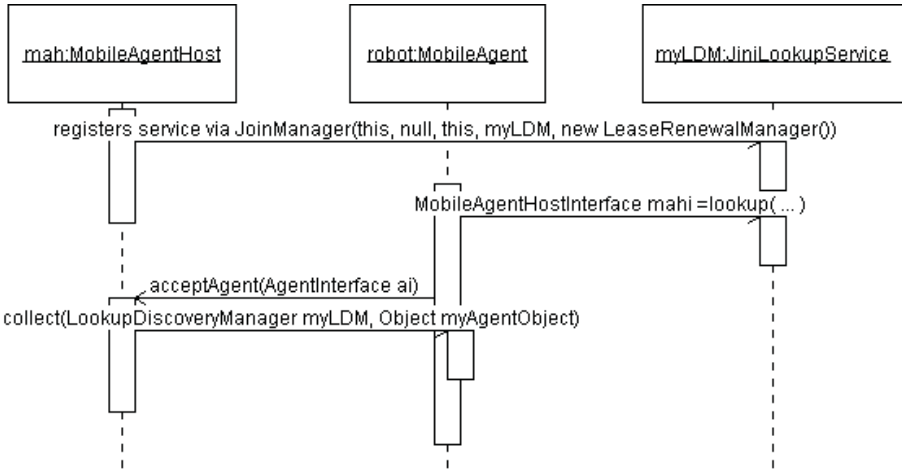


Figure 4: a scenario where the robot interacts with the lookup service

The final step in building the agent is the creation of a concrete mobile agent implementation. We chose to implement a `RouteMapAgent` that extends `MobileAgent`. This agent randomly travels to various agent hosts and logs its route via a `Logger` object. In this implementation, the object passed to the `collect()` method by the

`AgentHost` is the local hostname. The agent records this name in its route map. Figure 5 describes the `RouteMapAgent` class diagram.
`RouteMapAgent`'s `collect()` method, called when an agent turn up at a host, is implemented as follows:

1. Display the route taken to get to this host :this history operation allows the agent to decide what it has to do (to continue or to ask for some help),
2. After realized its own action, record the current host in the route table. It is important to mark all the location, where an agent is already gone,
3. Search for and travel to a new agent host with a call to `moveToRandomHost()` or directly to database host for a first update.

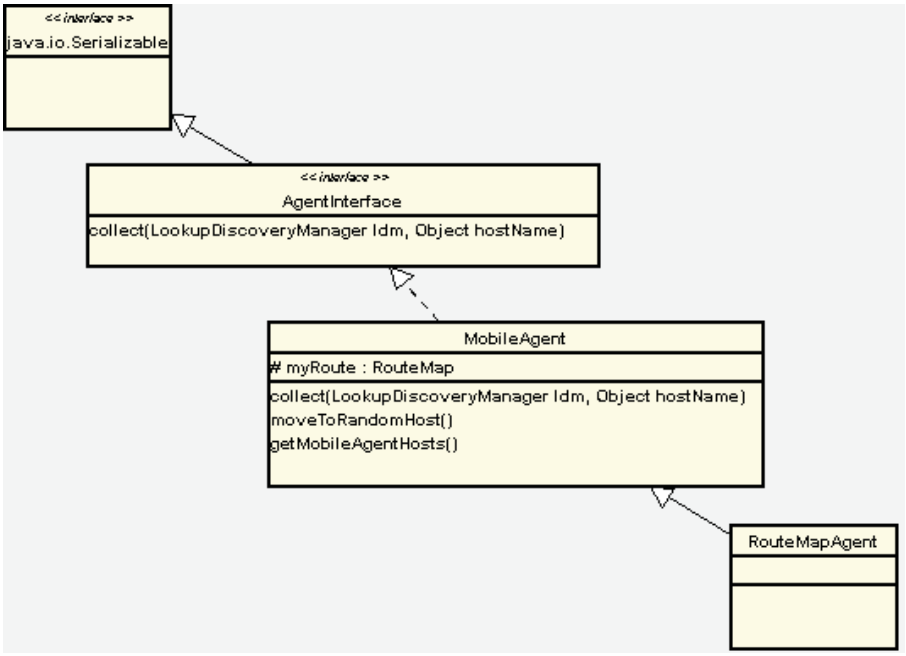


Figure 5: class diagram of the structure of the `RouteMapAgent` class

We illustrate these several steps with the code a the collect method:

```
public void collect(LookupDiscoveryManager ldm, Object hostName)
{
      // Trace the arrival in a log file
      logger.fine(this + " is present at "+ Calendar.getTime());
      // Display route in a log file
      for (Enumeration path = myRoute.elements(); path.hasMoreElements(); )
      {
            logger.fine ("Recent mobile agent hosts:");
            logger.fine ("Host " + i + " : " + iter.nextElement());
      }
      // Maintain a list of the last HOST_MAX hosts visited
      if (hostName != null)
      {
            myRoute.addFirst(hostName);
            if (myRoute.size() > HOST_MAX)
                  myRoute.removeLast();
      }
      // Rest for a while
      int sleepTime = (int)Math.floor(Math.random() * 5000);
      try
      { // import the information from a SQL file of the mobile agent host
            Class[] classes = new Class[] {SQLFileOrder.class });
            ServiceTemplate template = new ServiceTemplate(null, classes, null);
            sqlfo = (SQLFileOrder) registrar.lookup(template) ;
            SQLOrderSet sqlOS = sqlfo.importNewOrder("import.sql") ;
            addStatement(sqlso);
            Thread.sleep(sleepTime);
      }
      catch (InterruptedException ie)
      {
            logger.severe (ie);
      }
      // Move on to the next mobile agent host
      moveToRandomHost(ldm);
}
```

The `RouteMapAgent` performs a additional simple task. Alternate extensions of the `MobileAgent` class, it could provide more complex functionality.
The `RouteMapAgent` possesses several properties :

1. it is autonomous:
2. it is completely independent, coming and going as it pleases.
3. it controls its own actions by choosing where and when to move.
4. the agent is clearly goal-oriented. Its duty: execute the import of SQL orders into its bag, travel to various hosts and record its route.
5. the agent is temporally continuous, that is, a constantly running entity.
6. it is mobile, moving among various hosts in the network.

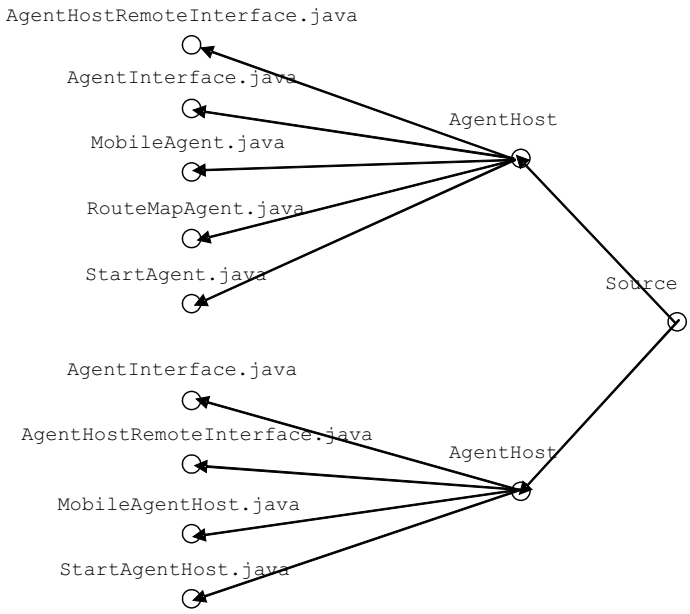
This agent clearly depends on the services which are available on the mobile agent host, but it does have the communicative property because the agent displays its route upon arrival at each host. Essentially, this agent talks, but cannot listen to.
Sometimes an agent might wish to specify a destination host, at the end of its travel for instance, when it has to go to the database front host. In this scenario, you can modify the `MobileAgentHost` to register its service and a corresponding globally

unique name. This is a modification of the `getMobileAgentHosts()` method to return a name/service template pair that each agent can select from.

4. A mobile framework

The most difficult part is always the deployment, because it depends on the current state of the services which are not only used by our application but also by all the code which take mobility as a basic feature [2], [3].
First it is essential to insure that all the source code are clearly separated from the bytecode files. If not, it involves some side effect such that new compilation of source code or set up of environment variables. We can divide the synopsis of a scenario into the following steps [4]:

1. We create 2 directories (`MobileAgent` and `AgentHost`) for the respective source codes of mobile agent and agent host .java files. To resolve compile-time dependencies, you must include the interfaces in both directories:



2. We define 2 environment variables called, `AGENT_DIR` corresponds to the absolute path of the agent directory you created. The `AGENT_HOST_DIR` corresponds to the agent host directory's absolute path.

3. Then we compile the Java files in the agent directory, where `JINI_HOME` is set to the Jini distribution's top-level directory : Jini 2.0:

```
javac -classpath %JINI_HOME%/lib/jini-core.jar;%JINI_HOME%/lib/
                  jini-ext.jar;
                 %JINI_HOME%/lib/sun-util.jar;%AGENT_DIR% -d..\..\..\
                  Class *.java
```

4. Then we compile the Java files in the agent host directory:

```
javac -classpath %JINI_HOME%/lib/jini-core.jar;%JINI_HOME%/lib/
                  jini-ext.jar;
                 %JINI_HOME%/lib/sun-util.jar;%AGENT_HOST_DIR% -d..\
                 ..\..\Class *.java
```

5. We create the rmi stub for the `MobileAgentHost` service:

```
rmic -v1.2 -classpath %JINI_HOME%/lib/jini-core.jar;%JINI_HOME%/lib/
                       jini-ext.jar;
                      %JINI_HOME%/lib/sun-util.jar;%AGENT_HOST\_DIR%
     -d %AGENT_HOST_DIR% MobileAgentHost
```

6. Several services have to be launched such that an HTTP server for clients to download Jini class files on the node where the Jini lookup service will run:

```
java -jar %JINI_HOME%/lib/tools.jar -port 8081 -dir %JINI_HOME%/lib
           -verbose&
```

7. We start an HTTP server that will provide the agent class files; this server must start on every node in which an agent initializes:

```
java -jar %JINI_HOME%/lib/tools.jar -port 8082 -dir %AGENT_DIR%
           -verbose&
```

8. Then we start an HTTP server that will provide the agent host class files; this server starts on every node in which an agent host runs:

```
java -jar %JINI_HOME%/lib/tools.jar -port 8083 -dir %AGENT_HOST_DIR%
           -verbose&
```

9. We start the RMI (Remote Method Invocation) activation daemon:

```
rmid -J-Djava.security.policy=%AGENT_HOST_DIR%/policy.all&
```

10. Then we start the Jini lookup service. Note that because Jini uses multicast for discovery, your network must therefore support it too. Also note that the policy file specified is wide open (no restrictions) which certain applications might not permit:

```
java -Djava.security.policy=%JINI_HOME%/example/lookup/policy.all
     -jar %JINI_HOME%/lib/reggie.jar http://localhost:8081/reggie
           -dl.jar
                    %JINI_HOME%/example/lookup/policy.all /tmp/
                     reggie_log public&
```

The hostname parameter, localhost, in my case, is the name of the server machine on which the Jini HTTP server started. The final parameter, `/tmp/reggie_log`, is a directory for the lookup service log files.

11. Start one or more `MobileAgentHosts`. Execute this command for each host:

```
java -Djava.security.policy=%AGENT_HOST_DIR%/policy.all
     -classpath %JINI_HOME%/lib/jini-core.jar;%JINI_HOME%/lib/jini
                 -ext.jar;
                %JINI_HOME%/lib/sun-util.jar;%AGENT_HOST_DIR%
     -Djava.rmi.server.codebase=http://localhost:8083/ StartAgentHost
      myHostName
```

The hostname parameter, localhost, in my case, is the name of the server machine on which the agent host HTTP server started; don't confuse the agent and agent host port numbers. The `myHostName` is a command-line parameter you can use to name the host whatever you like, which proves useful if you start multiple host processes on the same machine.

12. Start one or more `RouteMapAgents`. Execute this command for each agent:

```
java -Djava.security.policy=%AGENT_DIR%/policy.all
     -classpath %JINI_HOME%/lib/jini-core.jar;%JINI_HOME%/lib/jini
                 -ext.jar;
                %JINI_HOME%/lib/sun-util.jar;%AGENT_HOST_DIR%
     -Djava.rmi.server.codebase =http://localhost:8082/ StartAgent
      myHostName
```

The hostname parameter, localhost, in my case, is the name of the server machine on which the agent HTTP server started. The `myHostName` is a command-line parameter you can use to name this particular agent whatever you like.

When we work on a single node, start several `MobileAgentHosts` (one per process/JVM). If there is an access to a distributed network, we can start an agent host on each node. After starting several hosts, we start a `RouteMapAgent`. We should be able to watch the agent bounce randomly among the various hosts by examining `RouteMapAgent`'s output as it arrives at each host.

Conclusion

Mobile agents play an ever-increasing role in today's advanced software systems. With the advent of distributed computing and the power and flexibility it provides, complexity is at the forefront. You can incorporate Jini to mitigate complexity; it encapsulates the underlying functionality (sockets, messaging protocols, synchronization issues, fault tolerance, and so on) required for a successful distributed application. We like using Jini because We have less networking code to debug and hence am more productive because we can develop distributed applications faster. We can adhere to a flexible client-service model, concentrating my efforts on providing services and/or clients that use those services. Jini provides a powerful tool for distributed applications and maps quite nicely to a mobile agent architecture.

References

[1] David Kotz and Robert S. Gray, *ACM Operating Systems Review*, 33(3), August 1999, pages 7-13. It is an update of a position paper that appeared at the Workshop *Mobile Agents in the Context of Competition and Cooperation (MAC3)* at Autonomous Agents, May 1, 1999, in Seattle, Washington, USA.

[2] Joshua Fox's "Deploy Code Servers in Jini Systems"(*JavaWorld,* December 2001)

[3] Robert Flenner "Jini and Javaspaces application development", SAMS (September 2002)

[4] Scott Oaks « Jini in a Nutshell: A Desktop Quick Reference", Software Development, Computer Programming, Software Developer Books (2001)