IMPLEMENTATION OF AN AGENT MONITORING SYSTEM IN A JINI ENVIRONMENT WITH RESTRICTED USER ACCESS

Marietta A. Gittens
(Dr. Sadanand Srivastava, Dr. James Gil De Lamadrid)
{mgittens, ssrivas, gildelam}@cs.bowiestate.edu
Department Of Computer Science,
Center for Distributed Computing (CERDIC)
Bowie State University, MD, 20715.

Abstract

This paper deals with the implementation of our Agent Monitoring System in a Jini environment with restricted user access. Since most Jini applications rely heavily on Remote Method Invocation, our Agent and Group Services were modified to each exhibit an interface. The Agent interface object possessed the activity method implemented in all Agent servers, whilst the group interface object had to be modified to implement its interface. Once this was done the services had to register with reggie, which is a lookup service. Using the methods in reggie the Jini Lookup Service Browser can be invoked.

AMA Architecture

Our AMA System consists of two main parts: - an Agent Monitoring Agent (AMA) and an agent-based system. Agents can belong to groups or exist independently. The AMA consists of the **AgentMonito**r (our server) and its two associated user interfaces, **amaPage,** and **AgHistory**. Via its three canvases, amaPage shows agent activity, group hierarchies and agent communication. AgHistory displays the history of the agents on a timeline. The agent-based system consists of instances of communicating agents, which may or may not belong to groups. (see Fig.1)

Fig.1 THE AMA STRUCTURE

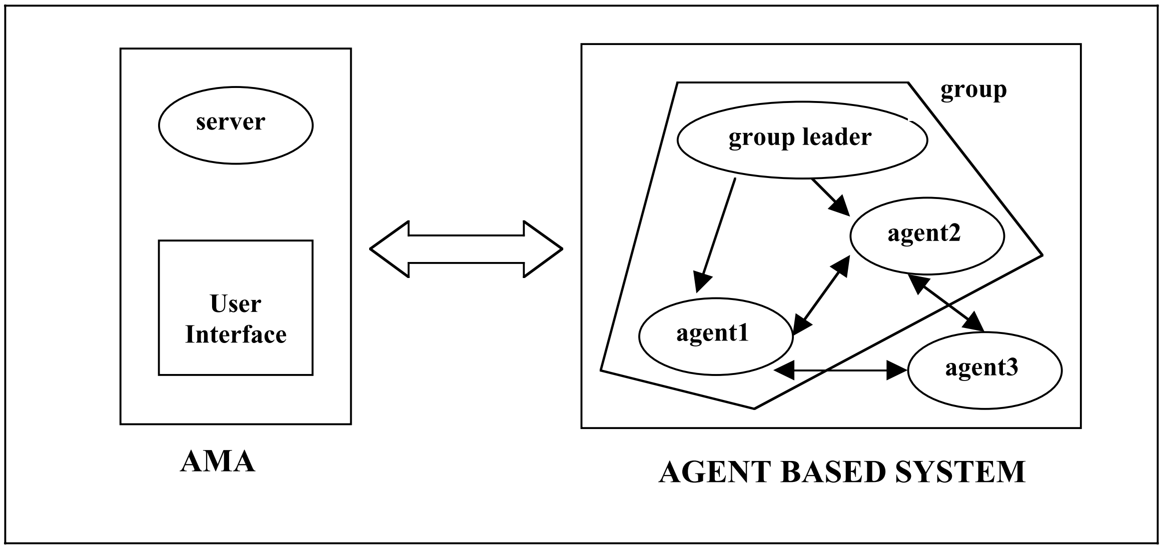
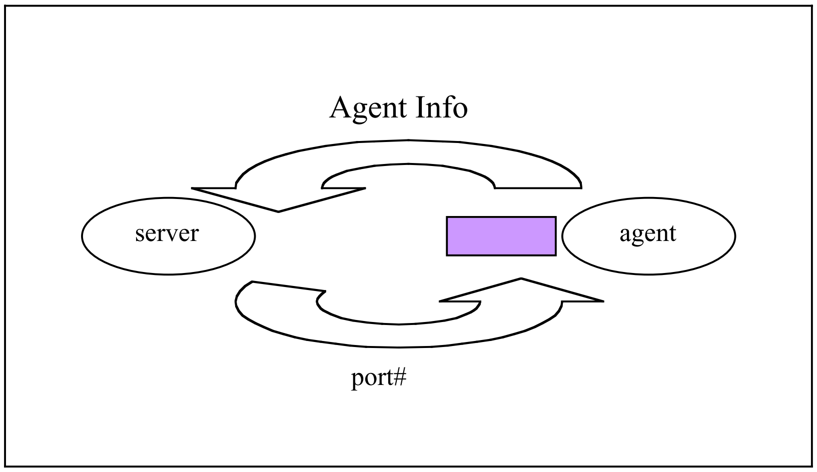


Fig.2 AGENT REGISTRATION



When an agent is created, it immediately registers at the **AgentMonitor** by passing on to it certain defining characteristics about itself. In return, the **AgentMonitor** issues a port number to the agent (Fig.2). Via this backport the server sends a ping message to check whether the agent is active or not (Fig.4). The **AgentMonitor** is not only capable of monitoring agent viability, but can also detect occurrences of inter-agent communication (Fig.3), and can track the path of a mobile agent as it moves from one host to the other. In order for each group and its members to relocate to a new host to continue operations there, a previously executing **AgentCatcher** must be there to receive them. This causes the agent to be reinitialized at the **AgentMonitor** (Fig.5).

Fig.3 MONITORING AGENT COMMUNICATION

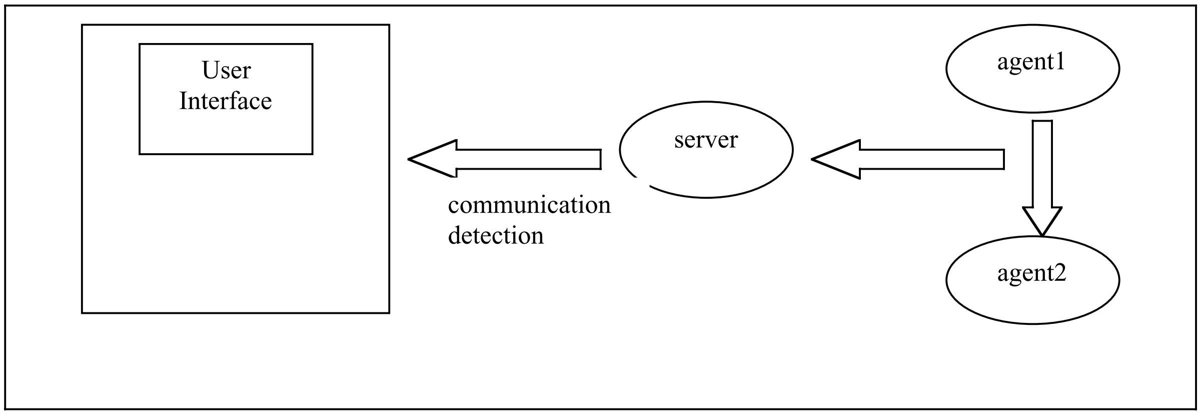


Fig.4 PING ACTIVE AGENTS

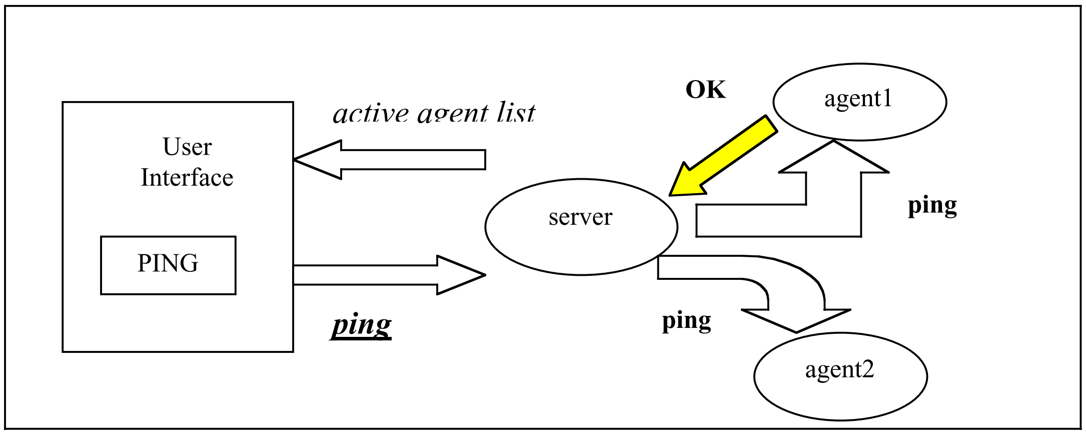
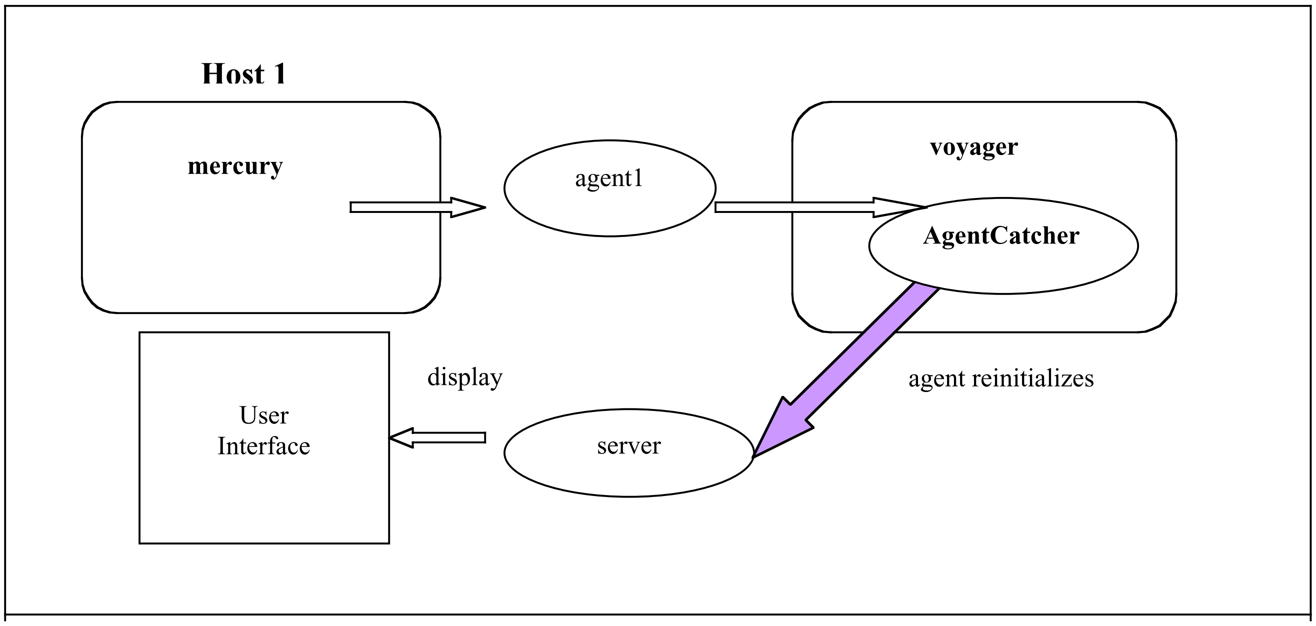


Fig. 5 MONITORING MOBILE AGENTS

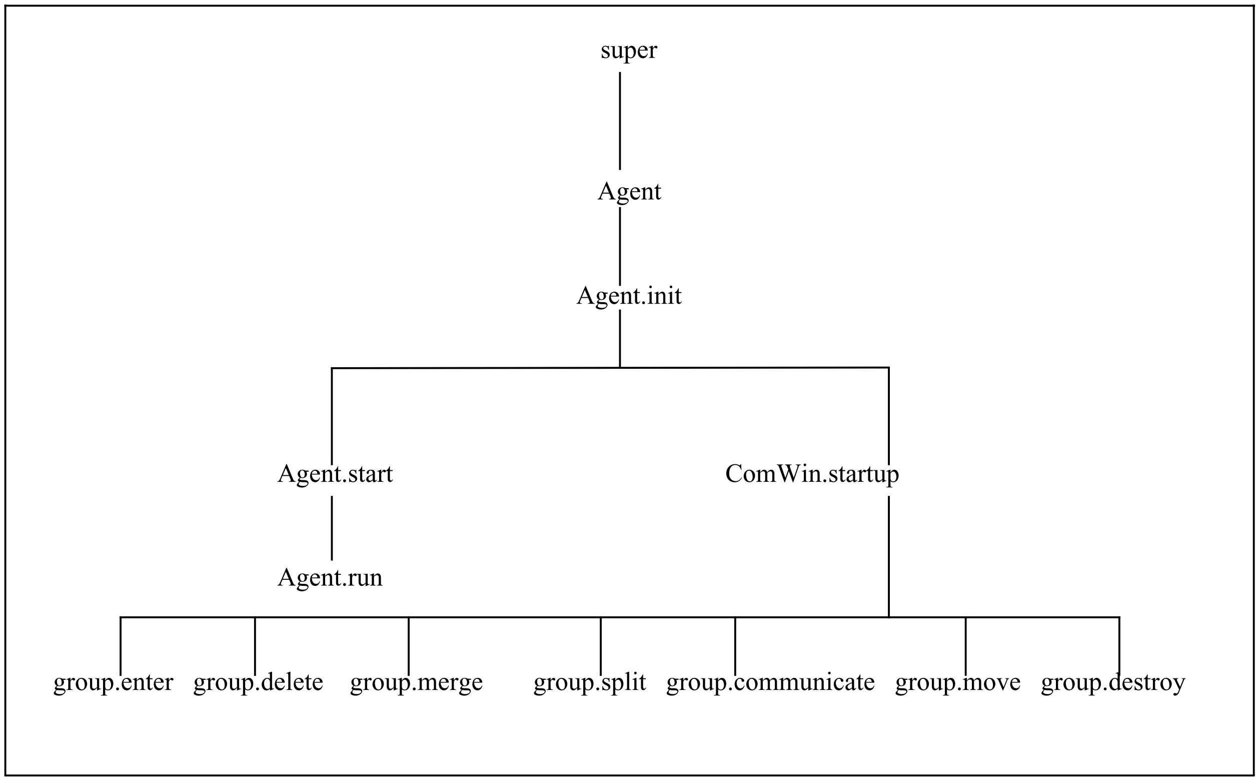


The function hierarchy chart below shows the common ancestry of agents and groups. The divergence occurs depending on whether the application calls Agent.start() or ComWin.startup(). Agents possess an activity method that is at present empty but can be made specific depending on the task the user wishes it to carry out. Group leaders activate their command windows via which they can accept groups of members, which may include agents or other groups. Groups possess the capability of

- **accepting** as many members as possible
- **deleting** members from their association
- **communicating** messages to one or all of their members
- **destroying** themselves or any or all of their members
- each accepting leadership of a group that is **merging** with it
- returning leadership to the former leader of a subgroup by **splitting** that group away
- **moving** their members to a new host to continue their operations there.

A file, containing user privileges that restrict interaction with groups, agents and the monitor itself, has been constructed. When a user attempts to start the AgentMonitor his access rights are first verified. If he is authorized to use the monitor the application runs. If he is not the application exits. A similar situation occurs with the execution of group and agent services.

Fig.6 FUNCTION HIERARCHY CHART FOR THE GROUP CLASS



The Jini Environment

Jini is a distributed computing framework that operates under the **client-server** paradigm. Clients and servers can communicate in a Jini community using any distributed computing protocol. The terms - **server** and **service** are differentiated in a Jini environment. A **service interface** or **service** is defined as the API that a server presents to the outside world. A **server** is recognized as the implementation of a service. In Jini a well-defined interface is of utmost importance since it is the object that is known and used by both the client and the server. Although many Jini applications are both clients and servers in their functionality, nevertheless all Jini services can be considered to be Jini clients because they use at least one other Jini service (the lookup service). The Jini framework allows services to locate each other on the network. The distinction between hardware and software is erased in a Jini environment. Jini is characterized by

- the absence of user intervention when bringing services on-line or offline (except when initially starting the service).
- Robustness — a result of its leasing and event notification methods. It can adapt to additions to or deletions from its community at any time.
- Dynamic loading of service implementations when they are needed.

Jini Prerequisites

In order for a developer to adequately utilize all of Jini s capabilities, certain prerequisites must be in place.

- Java must exist throughout the network since the lookup service runs on Java and all other services need to be either run directly on a Java VM or proxied to a Java VM.
- A network must exist. Jini services locate lookup services via the process of discovery. This process relies heavily on the fact that a network exists which supports the process of multicasting in its underlying infrastructure. Java VMs all require the presence of the TCP/IP protocol, which is only available in a networked environment.
- Devices that run the Java VM must be connected directly to the network. This requirement allows Jini to benefit from the fact that Java s VM provides a protocol-neutral interface to the underlying network.
- Participants in a Jini community must be aware of the identity of each other s service interface in order to locate instances of it on the network, via a lookup service or registrar.

Jini and RMI

RMI or Remote Method Invocation forms an integral part of Jini. It is very hard to separate the two. A Jini community benefits most from other Jini services if the underlying platform or VM upon which it runs supports RMI. Two major reasons for this is that

- all distributed events in Jini use objects that implement the RMI remote interface.
- Sun s implementation of the lookup service, reggie, gives an object that contains an embedded RMI reference. In order to receive this object successfully one s underlying VM must support RMI.

While RMI is generally the most obvious choice by Jini developers they are not at all limited to its use. If Jini is used in an environment that doesn t support RMI, the strategies used will require the availability of a proxy Java Virtual Machine. Jini services must be able to communicate with the Jini lookup service and they will have to do so through a combination of native code and proxies on the network.

RMI is based on a **client-server** paradigm. The **server** provides an implementation of service methods and runs on a particular host at a particular port. The **client** seeks these methods whenever it needs to perform a specific task. A list of the methods that the client can use, as well as their input parameter types and return value types, constitutes the **remote interface** or **stub** object that accompanies each server. The client invokes these methods using a **remote procedure call (RPC)** mechanism, which appears no different from any local procedure call. The RMI infrastructure, however, packages the parameters and ships them via the network to the VM (host and port) where the server is running. These parameters are passed to the appropriate method on the server. The server runs the invoked method in its VM and the results are again packaged and sent via the RMI infrastructure back to the waiting client.

Jini technology utilizes this RMI infrastructure to accomplish its distributed events. The client - an application - needs to accomplish a specific task. In order to accomplish this task there are servers on the network that the client can call. RMI necessitates that the client should know where the server is, how to reach the server, and what the server can do. However Jini sets out with different goals in mind. To accomplish these goals, it adds services to the RMI infrastructure that RMI does not support.

- It provides a **discovery** mechanism so that foreknowledge of server location is not needed by the client. Service discovery is part of the underlying infrastructure.
- It allows a client to continue its operations independent of the availability or unavailability of a service during execution by providing the **leasing** and **notification** mechanisms. With RMI, if a service is unreachable on the network, the client, which was trying to connect to it, fails.

In order for RMI clients to be able to download code from RMI servers, a security manager must be installed. Since all Jini services are clients of at least one other Jini service, the lookup service, all Jini services have a **security manager** installed in them. By default most Java applications do not have a security manager installed in them.

Jini Architecture

Overview

Within the Jini architecture a **lookup service** or **registrar** is placed on the network. There may be multiple lookup services on one network. Any Jini developer that wishes to place his service on the network **discovers** this service and **registers** with it. The lookup service provides each new member with a lease. **Lease renewal** must occur periodically If the service expects to retain its membership with the lookup service. As the name implies, lookup services are registrars that clients seek out and peruse in order to find service objects that can perform certain specific desired tasks on demand.

The Jini lookup service

1. The **Jini lookup service** keeps a list of all the Jini services that are available on the network. It holds the registrations of all other services available in the Jini community. An application that wants to use a Jini service finds the desired service by looking for the service s registration within the lookup service. A Jini service must register itself with the Jini lookup service in order for applications to find and hence be able to use it. Sun s version of a lookup service, is called reggie, Each service that wishes to register with reggie must possess its service interface in order to communicate with and invoke methods on reggie.

2. A service can register with the lookup service to be notified when another desired service has been added to the network. When this **notification** comes the desired service can be used. If the desired service becomes disconnected from the network the original service will also be notified and will be able to adapt itself to its new environment.

3. The lookup **service interface** defines all operations that are possible on the lookup service. Clients use the service by requesting services that implement a particular interface. When an application wishes to use a registered service in the lookjup service, it must know that service s service interface. It then asks the lookup service for all services that implement that particular service interface. The lookup service returns service objects for all registered services that implement the given interface.

4. A client finds a Jini lookup service through one of the fundamental pieces of Jini — a process called **discovery.** For most Jini programming the discovery protocol is hidden by the Jini APIs. There are two ways in which discovery is done.
   - **multicast discovery -** the client sends out a multicast request of a specified format. All Jini lookup services that see the request will respond to it. Clients will discover all lookup services that are within the multicast radius of the network. The multicast discovery packet will be broadcast on the local network. Routers that join two networks may or may not route the multicast packet between networks depending on its TTL (Time To Live) timestamp.
   - **unicast discovery** - The client attempts to connect to a lookup server that is known to exist at a particular location (host and port number).

5. When a Jini service starts, it discovers all the lookup servers on the network. It must then register with each using **Jini s join protocol**. A pure client that wants only to use services of the Jini community does not participate in the join protocol.

Leasing

When a service registers with the lookup service the lookup service issues it a **lease**, which the service must renew periodically. When the service fails to **renew** its lease the lookup service removes it from its internal list of registered services. If the **lease expires** the service is no longer accessible and the lookup service will automatically **unregister the service**. If the service was inaccessible as a result of network failure Jini code keeps track of the network s state and **reregisters services automatically** when the network is fixed.

Notification

This occurs through Jini s distributed event mechanism. Jini extends Java s standard event mechanism in the Jini Distributed Event Specification, which defines a set of interfaces and conventions for distributed events. Distributed events are more complex than local events since they must deal with potential network failures, potential server failures, and so on. The core API handles part of the complexity and part is handled by the Jini application s themselves.

Dynamic implementation of Services At Runtime

The server must provide a **serializable object** that implements the service interface. This object is loaded into the client, and the client executes methods on the object. A vital requirement of a Jini community is the ability to automatically download all classes that it needs through object serialization and dynamic class loading. This is made possible through RMI. In **object serialization** all member data within an object are converted to a stream of bytes, packaged and shipped across the network to the client. This package is tagged with the **codebase** of the code definition of the object i.e. with the directions needed to locate the code definition of the object. This code definition is needed by the client for the reconstitution of the original object.

AMA Architecture in a Jini Environment

Our Agent Monitoring System was implemented in a Jini environment with restricted user access. Since most Jini applications rely heavily on Remote Method Invocation, our Agent and Group Services were modified to each exhibit an interface. **The Agent interface object** possessed the activity method implemented in all Agent servers, whilst the **group interface object** had to be modified to implement its interface. Once this was done the services had to register with reggie.

STARTING OUR JINI SERVICE

<u>An httpserver is started In its own Java Virtual Machine (VM),</u>

In order for RMI to function correctly, there has to be a mechanism to deliver the class files to the client, which may not have these files in its classpath. The standard method is to use a standard web server and to set the java.rmi.server.codebase property to inform the client where to obtain the class files. The http server is needed to download classes that represent the core set of services that come with Jini.

<u>The Remote Method Invocation Daemon (rmid) is Activated</u>

Rmid creates a directory called log in its current directory and restarts any services that are registered with it. Services that use rmid are **activatable services**. The command that runs the activatable service registers itself with rmid and exits. Rmid then spawns a new VM that runs the service. Activatable services are persistent. If an activatable service crashes, rmid will restart it. Rmid will run until it is killed. Since Sun s implementation of the lookup service, reggie, is registered with rmid. **Reggi**e is an activatable service, and hence is activated by rmid.

Reggie is invoked by rmid

The lookup service provided by Jini is reggie. The command line for reggie is used to register the service with rmid and to provide rmid with the options it should use to invoke the VM in which reggie will run. Like all activatable services 2 VMs are involved.

(i) The setup VM which is used to register the service with rmid.
(ii) The server VM that rmid will start to run the service.

- The policy file which, has sufficient permissions to register the reggie service with rmid, is the first argument of the command.
- The next command line argument locates the jar file that contains the code for the reggie service. The class files of the reggie service are bundled as part of this jar file.
- The third argument locates the client side jar file, which is placed on the HTTP server running on a particular host. This URL parameter is used as the java.rmi.server.codebase property to tell the client where the class files for reggie can be obtained. When clients contact reggie they must obtain class files that provide the implementation of the service.
- To set the security of the Java VM that will eventually run reggie, the 4th command line argument is given. This security file must have sufficient permissions for reggie to run.
- The last parameter is the directory that is used by reggie to store its log files, which manage the persistence of the lookup service. Reggie reads these log files to recreate its previous state from the log.

The Jini Lookup Service Browser

The purpose of this Sun Jini service is to seek out and provide information on which services have registered with a particular lookup service.
- When the lookup service browser is started, if no lookup service is currently running on the network the browser displays **no registrars to select .**
- Once reggie is present on the network, the browser displays **1 registrar, not selected.**
- The ***File menu*** on the browser is used to specify or restrict which lookup services we re interested in discovering. A user can select a particular lookup service by hostname (and optionally by network port).
- The ***Registrar menu*** is used to pick the lookup service or registrar to examine. Selecting a registrar in this menu allows us to examine the services registered with that registrar.
- The ***Services menu*** and Attributes menu are used to refine which services within a lookup service we want to examine. Selecting an item in this menu implies that we want to see only services that implement the corresponding interface. Selecting >=1 attribute narrows our view to services that have corresponding attributes.
- The ***Options Menu*** is used to specify the display of the Services and Attributes menu. With this menu, we can specify all the types of Java classes that the Services and

Attributes menus support. In the default state services are selected only by their interface.

<u>Remote Events</u>

The concept of an event is the ability of one object to notify another object when something has happened. The Jini listener is a Remote interface, which allows events to be passed between different VMs. A Remote Event Listener object is an RMI stub that the source calls to deliver objects to the receiver. Listeners must implement the net.jini.core.event.remoteEventListener interface and the events themselves must inherit from the net.jinni.core.event.RemoteEvent class.

Future Work

- To treat each method as an atomic transaction process and to put mechanisms in place that would allow rollback to take place if a method cannot be completed in its entirety.
- To identify and solve portability issues arising with the use of the application.
- To provide mechanisms that would permit group creator proprietary rights to certain public instance group methods such as destroy etc.
- To place viewing restrictions on the user interface of the AMA system.