Concordia: An Infrastructure for Collaborating Mobile Agents

David Wong, Noemi Paciorek, Tom Walsh,
Joe DiCelie, Mike Young, Bill Peet
Mitsubishi Electric ITA
Horizon Systems Laboratory
1432 Main Street
Waltham, MA 02154, USA
email: {wong,noemi,walsh,dicelie,young,billp}@meitca.com

Abstract

Use of the Internet and the World-Wide-Web has become widespread in recent years and mobile agent technology has proliferated at an equally rapid rate. In this paper, we introduce the *Concordia* infrastructure for the development and management of network-efficient mobile agent applications for accessing information anytime, anywhere, and on any device.

Concordia has been implemented in the Java language to ensure platform independence among agent applications. The design goals of *Concordia* have focused on providing complete coverage of flexible agent mobility, support for agent collaboration, agent persistence, reliable agent transmission, and agent security.

Concordia offers a flexible scheme for dynamic invocation of arbitrary method entry points within a common agent application and extends the notion of simple agent interaction with support for agent collaboration, which allows agents to interact, modify external states (e.g., a database), as well as internal agent states. *Concordia* provides support for agent persistence and recovery and guarantees the transmission of agents across a network. *Concordia* has also been designed to provide for fairly complete security coverage from the outset. An alpha release of *Concordia* is available.

1. Introduction

Use of the Internet and the World-Wide-Web has become widespread in recent years and agent technology has proliferated at an equally rapid rate. There are two commonly accepted classes of agents in the literature: intelligent agents and mobile agents [25]. Intelligent agents are typically static entities with much built-in intelligence to perform a specific task. Mobile agents, on the other hand, are dynamic and have the ability to traverse an entire network, performing a number of tasks along the way, but with minimal intelligence. A number of recent efforts have been initiated to address the latter class of agents [1,14,15,21,23].

Concordia is a new framework for developing and executing highly mobile agents. *Concordia* offers a full-featured middleware infrastructure for the development and management of network-efficient mobile agent applications for accessing information anytime, anywhere, and on both wire-based and wireless devices. *Concordia* has been implemented in the Java language to ensure unimpeded interoperability and platform independence among agent applications. The design goals of *Concordia* have focused on providing complete coverage of flexible agent mobility, support for agent collaboration, persistence of agent state, reliable agent transmission, and agent security.

The remainder of the paper will proceed as follows. In Section 2, we discuss the overall system architecture of *Concordia*. In Section 3, we discuss *Concordia* agent mobility and transport. In Section 4, we present *Concordia* support for agent interaction. Finally, in Section 5, we discuss the current status of this work and the future directions that this work may take.

2. System Architecture

The *Concordia* infrastructure toolkit consists of a set of Java class libraries for server execution, agent application development, and agent activation. Each node in a *Concordia* system consists of a number of interacting component servers that could be executing on one or more Java virtual machines as shown in Fig. 1.

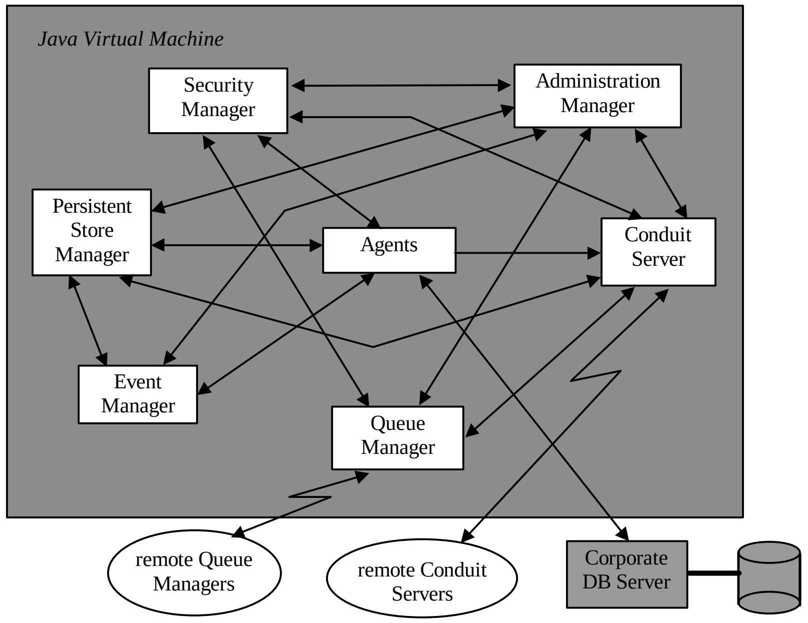

Fig. 1. Concordia System Data Flow Diagram

Concordia is similar to a number of existing agent infrastructures and toolkits with respect to its support for the basic communication plumbing that is required for agent mobility. For instance, both FTP Software's *CyberAgents* [5] and UC Berkeley's *Java-To-Go* [14] require the notion of a agent *propagation* server to propagate an agent. In *CyberAgents*, this server is called a *community*, while in *Java-To-Go*, this server is called the *Hall Server*. In the University of Stuttgart's *Project Mole*, this propagation service is embedded in the *location* server[13]. In *Concordia*, this propagation server is called the *Conduit Server*.

The Conduit Server serves as the communication server for agent transfer. An agent program initiates its transfer by invoking the Conduit Server's methods. The Conduit Server then proceeds to propagate the agent to the Conduit Server at another *Concordia* system. *Concordia's* agent mobility mechanism extends beyond the functionality provided in other Java-based agent systems by also offering a flexible scheme for dynamic invocation of arbitrary method entry points within a common agent application. This flexible scheme for agent mobility is discussed in more detail in Section 3.

While a number of efforts have provided support for agent interaction [1,5], *Concordia* extends this notion of simple agent interaction with support for two forms of inter-agent communication: asynchronous distributed events and agent *collaboration*. Asynchronous distributed events are scheduled and managed by the *Event Manager*, while agent collaboration requires the agent application programmer to specify a collaborating *AgentGroup* object through the utilization of the *Concordia* class libraries. Agent collaboration allows agents to interact, modify external states (e.g., a database), as well as internal agent states. Inter-agent communication is discussed in more detail in Section 4.

Agent persistence is required to ensure that agents can recover successfully from system crashes. Just as a number of other efforts currently (or plan to) offer support for agent persistence [1,7], *Concordia* also offers support for agent persistence. The *Persistent Store Manager* allows the internal state of agent objects to become persistable.

Although the Persistent Store Manager was designed and developed specifically to support persistence and recovery of agents, it is a general-purpose facility which may also be used to store internal state to facilitate recovery after server failure. The Persistent Store Manager implementation is based on the Java object serialization facilities.

Concordia agents are highly mobile and their mobility can extend to a number of local as well as wide area networks. To alleviate potential performance and reliability problems associated with the transmission of agents across networks with different characteristics in the underlying communication medium, the *Concordia* infrastructure also provides support for transactional queuing of agents between Conduit Servers residing on different networks.

The *Queue Manager* manages inbound and outbound queues for reliable transport of agents across a network. The Queue Manager communicates with its local Conduit Server and performs handshaking with other remote Queue Managers for reliable agent transmission. The Queue Manager design goals centered on achieving optimal disk space utilization, fast write operations, and fast recovery from server failure. Its implementation borrowed some ideas from the log-structured file systems research area [19,20] to employ a unique data architecture which ensures better overall performance over traditional message queuing systems [6,8,16]. The preservation of an object's class specification on disk is handled by the Java object serialization facilities while Queue Manager communication relies on the Java RMI package.

Concordia's security model provides support for two types of protection: (1) protection of agents from being tampered with, and (2) protection of server resources from unauthorized access.

Agents are protected from tampering while stored on client systems during transmission and while stored on the persistent store. Storage protection is handled by encryption. The only other system to address agent protection on both the client and server is the Itinerant Agents [4]. *Concordia* uses the SSLv3 protocol to transmit agent information from one system to another. Transmission protection is a de facto requirement for an agent application. In contrast, CyberAgents and Telescript appear to only provide secure internet transmission using encryption and digital signatures [5,23].

Concordia has implemented a highly flexible user-based security mechanism for server resource protection. The *Concordia Security Manager,* a Java object owned by the Java VM rather than a full-fledge process or thread, manages resource protection. Each agent is assigned an identity which allows the agent to access server resources. Resource permissions for agents can also be dynamically adjusted to increase or decrease an agent's security clearance. *Concordia's* resource protection differs from Aglets and CyberAgents in that it is based on the user of the agent rather than the developer of the agent. [1,5].

Concordia system administration is handled by the *Administration Manager*. The Administration Manager starts up and shuts down the other servers in the *Concordia* agent system. It also manages changes in the security profile of both agents and servers in the system and makes requests on behalf of the agent or server to the Security Manager. The Administration Manager also monitors the progress of agents through out the network and maintains agent and system statistics.

Users interact with a *Concordia* system by developing agent application programs. In the *Concordia* infrastructure, agent application programs are implemented as Java objects. Users would first need to write a Java class that specifies some action, such as accessing a database on a remote node. Once this Java class is written and compiled, the user can *launch* the agent program in three ways: (1) via a GUI Agent Launch Wizard, (2) via a command line tool, or (3) using the external API. The first two mechanisms are provided with *Concordia,* while the last one requires the user to write a customized launch class which makes use of the *Concordia* class libraries.

3. Agent Mobility

Since the agent objects are composed of a combination of code and data, object mobility means the network transportation of both code and data. As stated earlier, agent mobility is accomplished by the Conduit Server. Beyond providing for just the mobility of code and data, *Concordia* provides for the transmission of state information detailing where the agent has been and what it has accomplished as well as where it is going and what it still has to do. *Concordia* also provides interfaces allowing agents to create other agents and to clone themselves.

Some of the design goals for the Conduit Server were:

1. Provide for mobility that is transparent to users of the system.
2. Provide a programming model as close as possible to that of "regular" programming.
3. Build on existing infrastructure where available.

Within *Concordia*, an agent's travels are described by its *Itinerary*. The Itinerary is a data structure which is stored and maintained outside of the agent object itself. The Itinerary is composed of multiple *Destinations*. Each Destination describes a location to which an agent is to travel and the work the agent is to accomplish at that location. In the current implementation, location is defined by a hostname of a machine on the network and the work to accomplish is by a particular method of the agent class. Thus, if you had an agent class containing two methods, named *method1* and *method2*, a potential Itinerary for this agent could look like the following;

LOCATION	METHOD
server1.mycompany.com	method1
server2.mycompany.com	method2

When an agent is launched with this Itinerary, the agent would first travel to the machine identified by the TCP/IP hostname *server1.mycompany.com*. At that location the method *method1* would be invoked automatically by the agent system. After *method1* completes execution, the agent is transferred by the system to the node *server2.mycompany.com*. As the agent is transferred to the new location all of its internal state, meaning all of the information stored in its member variables, is transferred with it. This allows the agent to remember any computations it made at prior stops in its travels. Once the agent arrives at *server2* the system invokes its *method2* method which is allowed to run to completion.

There are some important characteristics of this Itinerary model that is worth noting. The Itinerary is a completely separate data structure from the agent itself. Thus where the agent travels is maintained in a separate logical location than what the agent does. This is very different than the Telescript [23] model where an agent's travel is initiated in its code by a call to its *go* method. A design decision was made to separate the agent's Itinerary from its code since this leads to a much more manageable system. Without extensive analysis of the Telescript code composing an agent and some knowledge of what runtime conditions will be like, it can be difficult to predict where a Telescript agent may travel. Further, it can be very difficult to locate where a Telescript agent has traveled after it has been launched. *Concordia*'s Itinerary model provides a simple mechanism for defining and tracking how an agent travels. For flexibility reasons, the system allows agent's to modify their Itineraries at runtime.

This Itinerary model also allows for multiple entry points into the agent to be executed at multiple locations. It appears that some existing agent systems, such as IBM's Aglets [2] or FTP Software's CyberAgents only support a single entry point into the agent. In these systems, at each stop in the agent's Itinerary, this entry point method is invoked when the agent arrives. Within the code of this one method, the agent must determine what work has previously been executed and then proceed to dispatch to the proper code to handle the work left to be completed.

This single entry point model unnecessarily presents the agent programmer with a different programming model than that of the non-mobile programming paradigm. For complex agent applications, this constraint can require the programmer to maintain a large amount of state information which can be better encapsulated within the agent's Itinerary.

Mobility of an agent's data was accomplished using the Java Object Serialization facility [17]. Transfer an agent state is a matter of serializing an agent's data down into a format suitable for network transmission, transmitting the data in this format, and then deserializing the data back into the original agent. This is very similar to the mechanism used by Java Remote Method Invocation (RMI) [18] for passing an object *by value* between distributed objects. RMI itself does not provide for true object mobility as it provides for no mobility of an object's code and in fact requires that the code for any objects passed by value be pre-installed on both sides of the network connection.

Java's Object Serialization features provide an almost transparent mechanism by which Java objects can be serialized into data streams and provided suitable technology for implementing agent mobility.

The problem of transmitting an Agent's code is solved in a manner similar to the way in which a web browser loads a Java applet. A browser will typically implement a special Java class called a called a *ClassLoader* [11]. The Java virtual machine makes a callback into the ClassLoader object whenever the system attempts to load the bytecodes for a Java class. In response to this callback, the ClassLoader implemented within the browser makes an HTTP request to a web browser in order to retrieve the file on the server containing the bytecodes for the class. The Java language provides a mechanism for converting the contents of this file into an actual Java class from which objects can be instantiated.

The *Concordia* infrastructure uses a very similar mechanism to support mobility of code. As an agent travels around a network its bytecodes and the bytecodes of any objects it creates and stores in its member variables are loaded via a special ClassLoader. This ClassLoader packages theses bytecodes into a special data structure which travels with the agent. During the deserialization of the agent, the bytecodes for the agent and its related classes can be retrieved from this data structure and are used to instantiate a new copy of the agent.

4. Agent Communication

Concordia includes two paradigms for inter-agent communication: asynchronous distributed events and collaboration. Its distributed events are, in many ways, similar to those offered by other systems. However, *Concordia*'s implementation of collaboration extends the events mechanisms in unique ways. Hence, this section examines both modes of communication, but explores collaboration in greater detail.

4.1. Events

Concordia provides two forms of asynchronous distributed events: selected events and group-oriented events. The event selection paradigm enables agents to define the types of events they wish to receive. In contrast, group-oriented events are distributed to a collection of agents (known as an event group) without any selection.

Selected Events

Concordia's *EventManager* object is the focal point for selected events. It accepts event registrations, listens for and receives events, and notifies interested parties of each event it receives. The EventManager filters each event it receives by notifying only those objects (e.g., agents) that have registered to receive events of that type.

Before an agent (or any other object) can receive selected events, it must register with the EventManager by sending it a list of event types it is interested in receiving and a reference to a location where it wishes events to be sent. This location is actually a distributed object. Hence, the EventManager can forward events to an agent even after it migrates to another system. Agents may choose to handle events synchronously in their main thread or asynchronously in an event handler thread.

The EventManager saves all registrations it receives in persistent storage via requests to the Persistent Store Manager. No registrations are lost during EventManager failures because it retrieves outstanding registrations from persistent storage whenever it restarts. Furthermore, EventManager failures and restarts are transparent to agents because they communicate via an *EventManagerProxy* object rather than directly with the EventManager. This proxy is responsible for re-establishing failed connections with the EventManager. (The EventManager itself is automatically restarted by an administration server that starts and monitors critical *Concordia* servers.)

The selected events programming paradigm is simple and easy to utilize in agent applications. As an example, an application may launch an agent that will notify it when the airfare between two cities drops below a certain price. The agent may migrate to a remote location and monitor a database containing travel information. When the agent detects the conditions have been met, it can notify the application via an event. Events are also commonly used to inform an application of exceptional conditions encountered by agents it launches. (In the future, this may also be accomplished via e-mail.)

Group-Oriented Events

As mentioned earlier, *Concordia* distributes group-oriented events to groups of objects without filtering them. Agents within an application that need to communicate or coordinate with each other often do so via group-oriented events. As an example, agents may wish to be notified when an agent within the same application encounters exceptional conditions. In particular, collaborating agents communicate via group-oriented events.

Joining an event group is a prerequisite for communicating via group-oriented events. Joining a group is similar to registering with the EventManager: each object (e.g., agent) joining the group sends it a reference to a distributed object where it wishes events to be forwarded. When the event group receives an event from one of its members, it forwards it to other objects in the group. Hence, all group-oriented events are distributed to the event group's entire membership.

Concordia offers two flavors of event groups: basic and persistent. The *EventGroup* object implements the basic functionality described above. The *PersistentEventGroup* object adds persistent group membership and reliable, transparent recovery from failures via proxy objects.

4.2. Collaboration

Distributed events have many applications, but they are not flexible enough to manage complex agent coordination. *Concordia*'s collaboration framework facilitates this type of interaction by enabling multiple agents to work together to solve complex problems.

Consider the following scenario: A user wishes to determine the best package price for a ski trip given the following criteria: a resort in the Alps, for a week in February, with slopeside lodging, and the lowest price for all expenses. To solve this problem, an agent obtains a list of appropriate ski resorts from a database before spawning other agents to query travel databases, possibly in different formats, for package prices at those resorts in February. Agents can perform this task more efficiently when they can correlate their results and adjust their computations based on the outcome of that collaboration.

Suppose the agents visit local travel agencies and then share their intermediate results and collaborate before migrating to travel agency sites in other cities. If an agent determines that a particular resort does not have any available lodging meeting the user's criteria, the agents may determine to drop queries about trips to that destination. As more information is gathered, agents may also make other decisions. As this example demonstrates, agents can perform complex distributed computations more effectively if they correlate their results and alter their behavior based on the combined results. *Concordia*'s collaboration framework facilitates this process.

The class of application described above divides a complex task into smaller pieces and delegates them to agents that migrate throughout the network to accomplish them. These agents perform computations, synchronously share results, and collaboratively determine any changes to future actions.

Concordia employs a simple programming paradigm for this type of collaboration. The goals of the collaboration framework include:

1. A simple programming interface for synchronous collaboration.
2. Asynchronous notification of exceptional conditions via events.
3. Reliable and robust implementation utilizing proxy objects to shield agents from the effects of software failures within the collaboration framework.
4. An infrastructure that enables location transparent inter-agent communication.

Agents within an application may form one or more collaboration units, known as agent groups. *Concordia* provides base classes for collaborating agents and agent groups (i.e., *CollaboratorAgent* and *AgentGroup*, respectively). AgentGroups are implemented as distributed objects which export a simple interface to CollaboratorAgents. These agents hold remote references to AgentGroup distributed objects and access them via Java's Remote Method Invocation (RMI) facility.

AgentGroup collaboration is implemented via a distributed synchronization point, known as a collaboration point, and a software method, *analyzeResults*. The AgentGroup abstraction provides the distributed synchronization. Each application need only supply its own implementation of analyzeResults to analyze the collective results of the agents in the group and to allow each agent to adapt its behavior based on those results. Both the synchronization point and invocation of analyzeResults are encapsulated within the AgentGroup's *collaborate* method.

This distributed synchronization scheme requires that each agent "arrive" at the collaboration point (by invoking the collaborate method on the AgentGroup distributed object) before collaboration may commence. Hence, it is ideally suited to applications that subdivide a complex problem into sub-tasks that correlate their results. When each agent arrives at the collaboration point, it posts the results of its computation to the AgentGroup and blocks until all the agents in the group arrive.

The AgentGroup collects the results of the agents' computations, and when all agents in the group arrive at a collaboration point, its collaborate method invokes analyzeResults on behalf of each agent, passing it the collective result set. The AgentGroup abstraction supports both parallel and serialized execution of the analysis stage of collaboration.

Concordia also provides both strong and weak collaboration models. Applications may specify (via an argument to the AgentGroup's constructor) whether collaboration should be allowed to continue if agents in the group fail to arrive at the collaboration point. The weak collaboration paradigm is useful for information-gathering agents that may wish to coordinate with each other even if some agents have terminated prematurely or encountered network failures. Agents that modify external states (e.g., updating a database) generally employ the strong collaboration paradigm, which aborts collaborations if all agents in the group do not arrive at the collaboration point.

The AgentGroup also utilizes time-outs to detect potential deadlocks. Note that since AgentGroup collaboration is designed for closely coordinated agents, deadlocks are generally caused by programming errors. Hence, the AgentGroup does not need to use a more sophisticated scheme for deadlock detection or avoidance.

An added benefit of AgentGroup collaboration is that it enables location-transparent inter-agent communication. As each agent migrates, it carries a remote reference to an AgentGroup distributed object and utilizes the AgentGroup as a gateway for communicating with the other members of the group.

As mentioned earlier, AgentGroups facilitate both synchronous collaboration and asynchronous notifications. This is possible because the AgentGroup object derives from the PeristentEventGroup object. AgentGroups forward any events they receive from their members (e.g., that an agent caught an exception) to the remainder of the group. Occasionally, they may also initiate events that they deliver to the group.

A benefit of this event management scheme is that AgentGroups temporarily queue events for in-transit agents and flush them after the agents arrive at their new destinations. Hence, no events are lost during agent migration.

The AgentGroup's persistent membership and agent status information also increase reliability. Whenever the group membership or the status of one of its agents changes, the AgentGroup saves the current state to persistent storage. If an AgentGroup terminates prematurely, it is restarted and restores the current state from persistent storage. Events queued for in-transit agents may optionally be saved to persistent storage.

AgentGroup restarts are transparently handled by *AgentGroupProxy* objects. Instead of communicating directly with an AgentGroup object, agents communicate via proxies which shield them from the effects of AgentGroup failures. Each agent creates its own AgentGroupProxy and the proxies coordinate to atomically re-create the AgentGroup, if it terminates or fails to communicate.

As detailed above, *Concordia*'s collaboration paradigm offers several benefits: a simple programming interface for synchronous collaboration; asynchronous distributed event management; support for agent mobility; location-transparent inter-agent communication; reliability, persistence, and transparent recovery from failure; deadlock detection; and a portable implementation. No other agent collaboration implementation offers all these features.

In contrast to *Concordia*'s simple programming interface in a language increasingly used for application development, Telescript is a language designed for writing mobile agents. It supports agent cloning and provides meeting places -- locations where mobile agents may communicate with stationary specialized agents (e.g., a mobile agent may request the lowest airfare between two points). Telescript does not, however, possess any support for agent collaboration. IBM's Itinerant Agents [4] also utilize an agent meeting point abstraction that is very similar to Telescript's meeting places.

The artificial intelligence community provides a broad range of agent collaboration features with an agent communication language (ACL) [10] (which actually consists of two different languages (KQML and KIF) [9]), combined with the development of an application-specific ontology [12]. Hence, it is much more difficult to program agent collaboration with ACL given its added complexity. In addition, ACL does not support mobile agents.

Other agent implementations [3,22,24] provide some of the features of *Concordia*'s collaboration framework, but fall short in several other areas.

5. Conclusion

In this paper, we have described the *Concordia* middleware infrastructure for collaborating mobile agents. *Concordia* offers a complete framework for the development and management of network-efficient mobile agent applications. The design goals of *Concordia* have centered on providing support for flexible agent mobility, agent collaboration, agent persistence, reliable agent transmission, and agent security.

Concordia's agent mobility mechanism extends beyond the functionality found in current Java-based agent systems by offering a flexible scheme for dynamic invocation of arbitrary method entry points within a common agent application. The *Concordia* framework offers support for agent interaction via the notion of agent *collaboration*, which allows agents to interact, modify external states, as well as internal agent states. *Concordia* also provides support for agent persistence and guarantees reliable transmission of agents across a network. *Concordia* has been designed to provide complete security coverage from the outset.

Concordia has been implemented in Java to ensure platform independence among agent applications. A alpha release of Concordia is available at the Mitsubishi Electric ITA Web site (URL=*http://www.meitca.com).* Future extensions to the existing functionality may include support for transactional multi-agent applications and knowledge discovery for collaborating agents.

6. References

[1] *Aglets: Mobile Java Agents*, IBM Tokyo Research Lab, URL=http://www.ibm.co.jp/trl/projects/aglets

[2] D. T. Chang, D. B. Lange, "Programming Mobile Agents in Java" URL=*http://www.trl.ibm.co.jp/aglets/*

[3] D. T. Chang, D. B. Lange, " Mobile Agents: A New Paradigm for Distributed Object Computing on the WWW", In *Proceedings of the OOPSLA96 Workshop: Toward the Integration of WWW and Distributed Object Technology*, October 1996.

[4] D. Chess, B. Grosof, C. Harrison, D. Levine, C. Parris, "Itinerant Agents for Mobile Computing", *IEEE Personal Communications Magazine*, 2(5), October 1995.

[5] *CyberAgents* Documentation, FTP Software Inc., URL=http://www.ftp.com

[6] *DECmessageQ Programmer's Guide*, Digital Equipment Corporation, Maynard, Massachusetts, 1994.

[7] "Distributed and Mobile Object Projects", OSF Research Institute, URL=http://www.osf.org/RI/DMO/DMO.html

[8] *Encina RQS Programmer's Guide*, Transarc Corporation, Pittsburgh, Pennsylvania, 1994.

[9] T. Finin, R. Fritzson, D. McKay, "A Language and Protocol to Support Intelligent Agent Interoperability", In *Proceedings of the CE & CALS Washington '92 Conference*, June 1992.

[10] M. R. Genesereth, S. P. Ketchpel, "Software Agents", *Communications of the ACM,* 37(7):48-53, July 1994.

[11] J. Gosling, F. Yellin, The Java Team, "Java API Documentation Version 1.0.2 - Class ClassLoader", URL=*http://java.sun.com/products/JDK/1.0.2/api/*

[12] T. R. Gruber, "A Translation Approach to Portable Ontologies", *Knowledge Acquisition*, 5(2):199-220, 1993.

[13] F. Hohl, *Mole Alpha 1.0 Documentation*, URL= http://www.informatik.unistuttgart.de/ipvr/vs/projekte/mole.html

[14] W. Li, *Java-To-Go*, Univ. of California, Berkeley, URL= http://ptolemy.eecs.berkeley.edu/~wli/group/java2go/java-to-go.html

[15] D.S. Milojicic, M. Condict, F. Reynolds, D. Bolinger, and P. Date, "Mobile Objects and Agents", In *Proceedings of the Second USENIX Conference on Object Oriented Technologies and Systems (COOTS),* Toronto, Canada, June 1996.

[16] *MQSeries: Message Queuing Interface Technical Reference*, IBM Corporation, Armonk, New York, 1994.

[17] "Object Serialization for Java", Javasoft Corporation, URL=*http://chatsubo.javasoft.com/current/serial/index.html*

[18] "Remote Method Invocation for Java", Javasoft Corporation, URL=*http://chatsubo.javasoft.com/current/rmi/index.html*

[19] M. Seltzer, "Transaction Support in a Log-Structured File System", In *Proceedings of the Ninth International Conference on Data Engineering,* February, 1993.

[20] M. Seltzer, K. Bostic, M. McKusick, C. Staelin, "A Log-Structured File System for UNIX", In *Proceedings of the 1993 Winter Usenix Conference*.

[21] M. Straβer, J. Baumann, F. Hohl, "MOLE: A Java Based Mobile Agent System", In *Proceedings of the European Conference on Object Oriented Programming*, 1996.

[22] K. Sycara, K. Decker, A. Pannu, M. Williamson, D. Zeng, "Distributed Intelligent Agents", The Robotics Institute, Carnegie Mellon University Technical Report, 1996.

[23] J. E. White, "Telescript Technology: Mobile Agents", General Magic White Paper, 1996.

[24] D. Woelk, M. Huhns, C. Tomlinson, "InfoSleuth Agents: The Next Generation of Active Objects", Microelectronics and Computer Technology Corporation White Paper, 1996.

[25] M. Wooldridge, N.R. Jennings, *Intelligent Agents: Theories, Architectures, and Languages*, Lecture Notes in AI, Vol. 890, Springer-Verlag Publisher, Berlin, Germany,1995.