EMAA: An Extendable Mobile Agent Architecture

Russell P. Lentini, Goutham P. Rao, Jon N. Thies, and Jennifer Kay

Lockheed Martin Advanced Technology Laboratories
1 Federal Street, A&E, 3W
Camden, NJ 08102
{rlentini, grao, jthies, jkay}@atl.lmco.com

Abstract

The Extendable Mobile Agent Architecture (EMAA) is a new agent architecture specification that aids in the development of an agent system. The architecture's component design that has layers of abstraction, providing a generic system. EMAA provides a framework for autonomous asynchronous mobile software agents to migrate among computing nodes in a network and exploit the resources at those nodes. A single implementation of EMAA can provide a foundation for several different agent applications.

Introduction

The Extendable Mobile Agent Architecture (EMAA) is a new agent architecture specification that aids in the development of an agent system. EMAA uses an object-oriented design, but deviates from the traditional message-passing paradigm among objects that are distributed throughout a network [1]. It provides a simple way for a mobile agent to migrate from one computing node to another and to use the resources at that node. EMAA does not impose any restriction on the behavior of the agents, thus allowing autonomous behavior. This paper explains this new mobile agent architecture, and the extendable nature of its components.

One of the major benefits of a distributed computing environment is the ability to break problems up into smaller sub-problems and solve those sub-problems in parallel [1]. EMAA exploits this characteristic by breaking an agent's high-level goal into tasks that can be executed in parallel. Another characteristic of distributed computing environments is that resources may be distributed among different nodes in the network. The concept of breaking a program up into sub-programs fits nicely with this structure because a task that needs to exploit a resource can be packaged into a separate, independent component.

The resources available at a particular node are determined by the application requirements. In some situations it is convenient to package the logic needed to access these resources into a distinct component. In the EMAA methodology, these service-providing components are called servers.

EMAA is an architecture specification; its implementation is totally reusable and extendable. A single implementation of EMAA may serve as a foundation for several different agent systems.

In this paper we introduce the concept of taskable agents and explain its robustness. This warrants an explanation of the components needed to support such an agent. We first give an overview of the architecture. We then concentrate on the various components that make up the system. We conclude with a section that studies the interaction among components in this distributed architecture.

The Extendable Mobile Agent Architecture

The architecture has three major components: the agents, servers and the dock. At the most basic level, the agents perform the specialized, user-defined work. Agents travel through the system via the docks. Although the dock has many other functions, its primary purpose is to serve as a daemon process used for sending and receiving agents between docks at other nodes. Furthermore, nodes that offer specialized services to agents must do so via a server. An overall picture of a distributed computing environment using the EMAA design is shown in Figure 1.

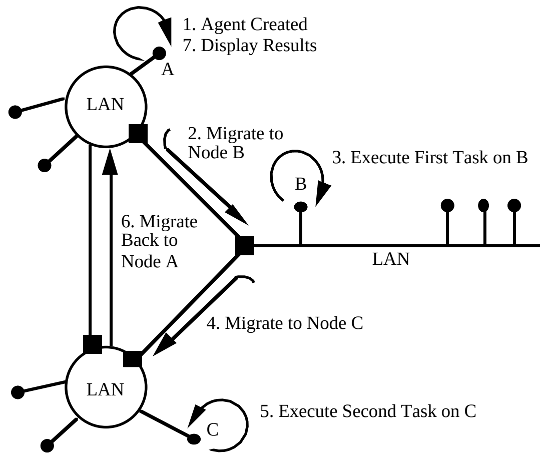


Figure 1: A Wide-Area Computing Environment

Agents

The key functionality of a mobile-agent computing paradigm is the ability to have a program execute at one node, and then migrate to another node with its state preserved. This program is called the agent. It is not sufficient to only allow support for the creation and migration of agents. The EMAA design provides for a more structured paradigm. We will first discuss the structuring of an EMAA agent and how it will help the writing of agent applications.

As any program, an agent is created to achieve certain goals. However, there are some basic differences in the way EMAA views an agent and a standard program. In many agent applications, agents are created to perform a number of tasks to achieve various goals. These tasks may be performed several times by many agents. The key difference is that a standard program is viewed as achieving one goal, while EMAA views an agent as a vehicle to carry and execute a number of these programs, perhaps at different nodes. Programs that achieve a goal are called tasks. The goal of the agent is to complete all of its tasks. This defines the concept of taskable agents. Since the agent is a mobile program, one must consider both the task as well as where that task is to be performed. EMAA supports the mapping of a task to a node, allocating resources for the task to be executed. It is important to note that EMAA does not restrict an agent to only be programmed in this manner, but EMAA suggests that this view may simplify application development.

The standard EMAA agent has an itinerary. The basic concept of an itinerary has been used in other systems such as IBM's Aglets [8]. An EMAA itinerary is a list of all the tasks an agent has been created to achieve. It contains a mapping of which computing node these tasks need to be performed on, and the program to be executed in an event that this task failed to be performed for whatever reason (the event handler).

Figure 2 shows an itinerary where an agent visits three different nodes. The agent is in a state where it has executed the first two tasks, T1 and T2. Task T1 yielded result R1, which the agent can recall if needed. This itinerary shows that task T2 resulted in an error. In this case, the standard agent would have executed the event handler E2. Furthermore, the agent is in a state where it has yet to execute task T3.

Machine	Task	Event Handler	Result	Status
Host A	T1	E1	R1	√
Host B	T2	**E2**	—	✗
Host C	T3	E3		—

Figure 2: A Mobile Agent Itinerary

In practice, there may be inter-task constraints. For example, the tasks may need to be executed in the same order that they appear in the itinerary. Another such example is that a task may not be executed if any task prior to it failed to execute. EMAA is flexible in programming such constraints. For example, the event handler can handle tasks that fail by removing dependent tasks from the itinerary.

Every time an agent is created or migrates to a new node, it starts executing from a predefined entry point. The logic contained within this entry point is responsible for executing the tasks of an agent. Let us consider an example of how an agent may make use of its itinerary at its entry point.

The program segment at the predefined entry point of the agent may look as follows. The structuring of agents into tasks yields a model to program configurable agents not tied to any one specific goal. The program needed to accomplish a task is packaged into a separate component. These components may be carried and executed by agents. Without a model it would not be possible to separate an agent's goal into smaller goals. There are a number of advantages from splitting a goal into smaller sub-goals

Parallelism: Multiple tasks may be executed in parallel by one agent, or by an agent delegating some of its tasks to other agents. Furthermore, these tasks may execute in parallel at different nodes; a single agent can start a task component on one node, and while that task is in progress, migrate to another node and start another task.

Configurable agents: Tasks may be assigned to agents in various ways, as long as they do not violate any existing constraints. This allows for agent configuration at run-time, without having to re-program a new agent for every new goal. As long as the new goal can be accomplished by some combination of existing tasks, an agent can accomplish the goal. It is important to note that an agent may configure itself dynamically choosing to add new tasks to itself or discarding used tasks, thus reducing its size.

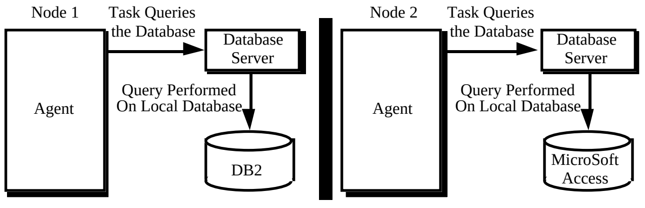
Reusable components: If two agents have slightly different goals, it may be possible to identify some common subtasks. The programs needed to accomplish these tasks may be re-used if they are properly created.

Servers

In many agent applications, one of the compelling reasons that an agent will visit a computing node is to utilize the resources at this node. There are three important points to be noted here. First, to conserve bandwidth we want to migrate as little code with an agent as possible. Second, the code or logic needed to exploit the resources at the node will usually be the same for all agents. Finally, it is desirable to separate the implementation of these resources from the implementation of the agent application.

Consider an agent-based database application where the data resources are distributed among several computing nodes. In a heterogeneous database environment, the code needed to query a database will not be the same at all the nodes. In fact, the actual implementation of the databases at these nodes may be changing. In such situations, it is better to package the code needed to access these resources into a separate component, that remains at the node, known as *servers.* It is beneficial to have a common interface between the agents and the servers that offer similar services at different nodes, even though the servers may differ in their implementation. This keeps the agent machine-independent.

Figure 3 illustrates an agent performing a database query at one node hosting a DB-2 database and then migrating to another node hosting a Microsoft Access database. The agent uses the same interface on each server to execute a database query even though the server's implementation is different at the two nodes.


Node 1
Task Queries the Database
Database Server
Query Performed On Local Database
Agent
DB2
Node 2
Task Queries the Database
Database Server
Query Performed On Local Database
Agent
MicroSoft Access
Figure 3: An Agent Performing a Database Query

We can now piece the agents, tasks and servers together. An agent executes a task at a node. The task may access servers to exploit resources at that node to achieve a certain goal. If the interface to the servers is the same across various nodes, the same tasks may be used with different resources.

Dock

Servers and agents (and their tasks) are sufficient to develop an agent application. However there are some necessary components required to support servers and agents. First, a daemon process is needed at a node to receive an agent. Second, some well-known component that we know exists at a node that can access any other component is needed. This well-known component is called the *dock*. The EMAA model specifies the dock structure in detail and that will be the essence of the following discussion.

The most important features of the dock are to serve as a daemon process and as a placeholder for other components. However, the dock is more structured than just this. The dock consists of the following components:

1. Communications façade
2. Agent manager
3. Server manager
4. Event Manager

Communications Façade: The communications facade is the daemon process itself and is meant to handle all connections to any other computing node. It manages the transmission of agents to and from the local node.

The communications façade offers services to send and receive an agent from a node and begin execution of the received agent at the predefined entry point of the agent program.

Clearly a well-known daemon process or component at a node is needed to send and receive agents. It is a good idea to add functionality to this same component to also send agents, instead of the agents interacting directly with the communications façade at the node the agent wants to migrate to. There are two compelling reasons for this. First, the communication software is packaged into one component, separating it from the rest of the agent system. If a change in the protocol for the transmission of the agents were needed, only the communications façade will need to be modified. This minimizes communication protocol dependencies between components at different nodes [4]. Second, there is a more controlled mechanism for sending and receiving agents. Such a mechanism allows for optimization of network use in the presence of autonomous components.

Figure 4 illustrates a situation that may occur if the agents were independently communicating with the remote node, without a communication façade. At time T1, component C1 attempts to send an agent A1 to the remote node. This process may require making a network connection. For example, if TCP/IP were the protocol being used, this would require setting up a socket connection. Soon after the agent is sent, the connection is released. At the same time, or maybe shortly after, a second component C2 attempts to send another agent A2 to the same node. All the work of setting up the network connection to the node will have to be redone. Setting up such a connection is not a trivial process and continually re-establishing connections on low-bandwidth networks is undesirable [11].

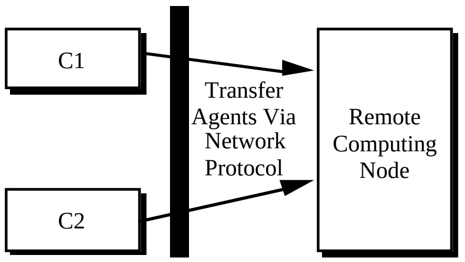


Figure 4: Illustrating the Agent Migration

Now consider Figure 5 where the components request the façade to transmit the agents for them. The connection can be established once, and both agents can be sent over the same network connection.

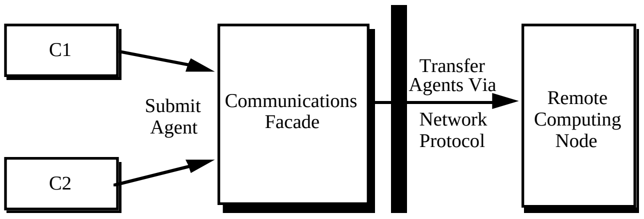


Figure 5: Agent Migration through Façade

In fact, there are more optimizations that may be applied. Piggybacking is a good example. If two agents are traveling in the same direction to the same node, and the agents only differ in the tasks that they are carrying, EMAA can combine their itineraries and send a single agent. These optimizations are best done in the communications façade.

Agent Manager: When an agent is received at a node by the communications façade, it is handed over to a component known as the *agent manager.* The agent manager is responsible for registering the agent and initializing it for execution. In an actual implementation, the initialization of an agent will play a key role. If an agent is to make use of a server at a node, it will need to obtain handles to the servers from somewhere. As we mentioned, the dock serves as a placeholder for all components, including servers, at a node. Therefore an agent may need to obtain a handle to the dock before it can start executing. These handles can be setup during initialization. Other practical uses of an agent manager may be agent collaboration [9] and agent validation for security reasons [12].

Server Manager: The *server manager* manages the server components at a node. The server manager provides a controlled mechanism to start and stop a service. The architecture specifies that an agent must obtain a handle to any server via a request to the server manager before it can be used.

A server manager is meant as a placeholder for any application dependent constraints. For example, the server manager is a good place to incorporate an economy model [6].

Event Manager: An event is defined as an announcement that some component in the system has reached an important state that may be of interest to other components. Many times it is unknown if and when a component will generate an event. For this reason, agents and servers dependent on events must have some way of “listening” for an event that may or may not happen. To support this, an event management scheme where a component can generate an event or wait for one is needed.

Furthermore, a mobile agent system by definition is dynamic in the number and types of components resident at the dock at any given time. This characteristic makes it difficult to pre-program a component that will raise an event to communicate directly with any other components which may need to be notified when that event occurs. Even if this could be done, it is more efficient to handle events through a centralized event manager.

EMAA specifies a fairly simple event model. There is an event manager component contained within the dock at every computing node. Any agent or server can generate an event and register it with the event manager. Agents and servers can also wait for an event or be notified on an event. There are two mechanisms whereby a component is notified of an event: the *Blocking Method* and the *Callback Method*.

In the blocking method, once a component requests the event manager for notification of a particular event, the component itself will block execution until the event occurs. In the callback method, a component registers a callback with the event manager. When the event occurs, the event manager executes the callback. In both cases, the component can handle the event in a suitable manner.

Figure 6 depicts the logical view of EMAA using UML (Unified Modeling Language).

Dock Interaction

We have discussed the major components of EMAA. We conclude the presentation with some higher level discussions and considerations.

Component Loading: When an agent migrates to a node, there may be some components of the agent already present at the destination node. It is more efficient for an agent to migrate with only the references to its components and use those present at the destination.

When the agent manager is ready to execute an agent, it makes sure that all components required by the agent are present. If any component is not found at the node, then the agent manager can request the *component loader* to contact the last node the agent came from that is known to have that missing component. The loader contacts the *component server* at the remote node and requests the component to be transferred. Once these components are locally available, the agent can begin execution. EMAA ensures that all components of an agent are available at the originating node.

Resource Servers: It is useful to have some mechanism to advertise a node’s resources to all other nodes. This allows every agent to be aware of the resources available on the network. An agent can determine what resources it can use to achieve a goal at runtime. Therefore there is the concept of inter-node resource sharing. EMAA specifies a *resource server* for this purpose.

The algorithm for a resource server is split into two phases. At dock start up, every resource server is responsible for announcing the node’s resources to every other node. In response to receiving the resources of another node, the receiving server must do two things. First, the foreign resources must be locally registered. Second, the local resources must be advertised to the new node, since the new node was not running when the local resources were announced. This guarantees that every node and every agent will be aware of all resources at all reachable nodes. Resources no longer available must be removed via another announcement.

An Application to Information Discovery

Using the EMAA design, Advanced Technology Laboratories has developed a mobile-agent-based information retrieval and dissemination application. The Domain Adaptive Integration System (DAIS) is a DARPA funded project with the general purpose of pushing and pulling military intelligence information across low-bandwidth, unreliable networks [17, 7]. These networks consist of heterogeneous sets of nodes containing numerous

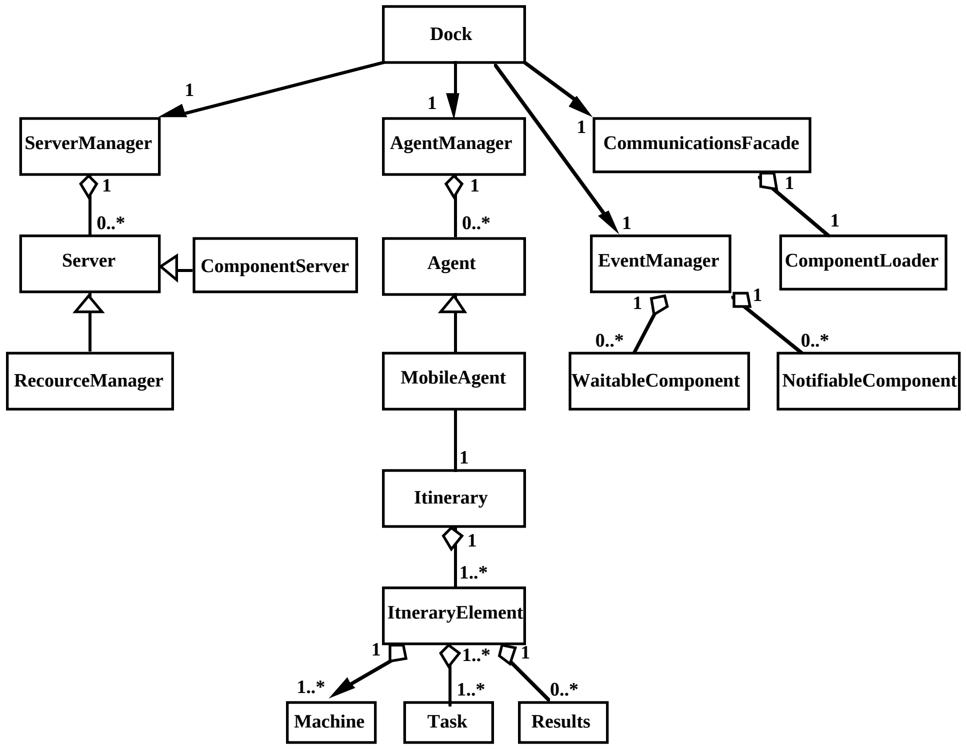


Figure 6: Logical View of EMAA Using UML

distributed heterogeneous databases. The system needed to be reliable enough to be used in live field exercises, and extensible enough to be used in domains other than military intelligence, e.g. logistics.

Development

Using EMAA we were able to quickly layout and develop a prototype system using Sun Microsystems' Java language by defining three servers, six tasks, and a controller to serve as the user-interface to the dock. Each component of EMAA is represented as a Java class or interface. Since many operations in DAIS involve information being inserted into and extracted from databases, the servers and tasks at the very least needed to collectively provide database-querying capabilities. With that in mind we defined the *MetaData Server* and the *Database Server.*

A MetaData Server and a Database Server must be present at every node hosting a database. The role of the MetaData Server is to collect schema information from all databases at the node and distribute it throughout the local network. This is implemented by the resource server model defined in the EMAA specification. In this fashion, all nodes in the sub-net[1] have information about all available databases in that sub-net and can construct queries locally, which will be performed at remote nodes. The role of the Database Server is to provide uniform access to the heterogeneous databases at the associated node. A Database Server distributes its database query tasks to all other nodes according to the resource server model.

1 A sub-net is defined with respect to a dock as the set of nodes that the dock is aware of.

Probably the most fundamental functionality of DAIS is contained in the *Database Query Task* and the *Database Update Task*. These tasks simply execute a SQL statement on a database.

At a higher level of abstraction is the *Abstract Query Task*. This allows a user to query on a "virtual" field that is mapped to actual fields within any of the databases. This leads directly to the third server defined in the DAIS system, the *Domain Server*. This server distributes specialized *Domain Objects*, which contains "abstract mappings" to the nodes on the sub-net. When an abstract query is created, the appropriate Domain Object initially translates it into an itinerary of query tasks. This creates a unique Domain Object containing its own unique domain knowledge for each domain defined in the system.

In many situations a user might not be able to attain sufficient information from the local sub-net. In practice, there may be useful information or resources outside the local sub-net. To accommodate this we defined *the Sub-net Discovery Task* to find such information or resources. An agent containing a Sub-net Discovery Task on its itinerary will attempt to discover adjoining sub-nets at each node it encounters. The agent will with its results only after visiting each node in each sub-net reachable from the local sub-net.

Other tasks defined in the core DAIS system are:

1. The *World Wide Web Task*, which allows an agent to query the Internet for information using any of the well-known search engines.
2. The *Natural Language Query Task*, which uses a template-based parsing algorithm to create an abstract query from natural language input.
3. The *Return Task*, which allows results to be returned to not only the originating node but also any other node on the local sub-net.

A preliminary OMG IDL (Object Management Group Interface Definition Language) interface to DAIS has been developed using the Java JDK 1.2 Beta 2 Java IDL [14], adding CORBA capabilities to the system. This was easily accomplished using the EMAA design by creating a single server that provides access to DAIS services via a standard IDL interface. Foreign agents, applets, etc. can communicate and request services from DAIS via this interface.

Performance

The system has proven to be both easy to develop and extremely robust, running on even 4.2Kbs wireless networks which lose connection frequently. EMAA allowed the authors to easily configure agents that would be efficient and fault tolerant in their operation. For instance, DAIS incorporated a tool to gather network statistics into the Communications Façade. Using this information, the agents could travel the higher bandwidth links whenever possible. When agents do need to traverse low-bandwidth links, the situation may arise that the link deteriorates to a point that makes it impossible or too inefficient for an agent to migrate via that link. In such cases, the agent need not abandon its attempt to migrate to that node. Instead, the agent can be configured to migrate to a new location then attempt a jump to the node from the new location. This behavior continues until either the node is reached, the agent times out, or another user-defined event occurs.

Related Work

EMAA is a mobile agent system architecture with fully configurable and fully taskable agents. It allows full control of agent behavior through the task concept. EMAA agents can be highly specialized entities or they can simply be generic shells to be populated with specialized tasks. With its component design, EMAA agents are fully removed from the tasks that they execute. In a real-world network, it cannot be assumed that all possible tasks will be defined at agent creation. In fact, resource nodes that emerge at arbitrary times during an agent system's life span will likely have new and unique tasks. In EMAA, these emerging tasks are distributed to the individual docks for use by any agent. Other agent systems such as Mitsubishi Electric's Concordia [10] and IBM's Aglets [8] accomplish an agent's goals by executing conditionals contained within the agent body. However, unique tasks emerging during the system's run-time cannot, in fact, be adopted by these agents without actually modifying the agent itself. EMAA offers a more object-oriented approach.

Collaboration among agents is an extensive area of study in mobile-agent research [15, 9]. The EMAA component model directly lends itself for any collaborative scheme to be incorporated with its implementation. EMAA has been augmented with an agent-collaboration tool known as the ATL Postmaster [9].

It is also worth mentioning that there are languages dedicated to mobile computing which are suitable for cross-platform use, such as AgentTcl [5]. Such cross-platform languages allow for a truly heterogeneous computing environment.

The advantages of component-based mobile-agent paradigms have been stressed in [2]. The paper discusses a concept, called code-on-demand, that is similar in some respects to the component server in the EMAA architecture.

Future Work

Further work is being done in the following areas:

Security: We are currently considering various techniques to augment the architecture with security features. The agent manager is a good place to incorporate the validation and authentication of agents. Other security issues are related to security concerns in any distributed application, such as encryption of transmitted agents. A number of issues in mobile agent security have been addressed in [12, 3].

Control: In a practical implementation of EMAA, there will be issues regarding the permission of resource use by an agent. An agent may be able to access some subset of the resources at a node and these constraints can be programmed into the agent and server managers.

Mobile Nodes (Mobile IP): While mobile programs add a great deal of flexibility and enhancements to distributed applications, mobile computing environments will involve mobile computing nodes and users themselves in addition to mobile programs. Extensive work has been done in the study of network protocols that support node mobility, such as Mobile IP [13, 16]. We are currently experimenting with adding Mobile IP support to the Communications Façade.

Conclusions

The taskable concept of the EMAA agents is a very powerful programming paradigm in the mobile computing environment. The structuring of the system into servers and tasks makes EMAA a truly object-oriented agent system. Furthermore, task components empower an agent to achieve a goal that it has not been specifically programmed to, making them fully configurable.

As shown, the announcement of resources is done through the resource server. Such announcements are crucial when viewing the interaction among nodes. Typically, these capabilities become important when the EMAA implementation is used over a wide area network, involving many nodes, with many independent groups contributing to the resources of the entire network. EMAA makes it possible to program such highly distributed applications by the concept of sharing resources through the distribution of task components to access these resources.

Another major advantage of EMAA is the concept of the communications façade. Without this component, administration of network use becomes increasingly difficult as the number of components in the system rises. This becomes a major factor in limited bandwidth networks. As long as mobile computing nodes have to rely on low bandwidth communication links, agent-based applications will have to deal with network use and optimization in clever ways.

We have successfully implemented EMAA in the Java language and applied the architecture to various agent applications. Our implementation of EMAA demonstrated its suitability as an architecture for rapid prototyping of mobile agent systems.

Acknowledgments

We would like to thank Jody Daniels, Martin Hofmann, Julius Etzl, Lori Pridmore, Ken Whitebread, Jose Rivera, Steve Jameson, and all the members of Lockheed Martin Advanced Technology Laboratories. This work has been supported by DARPA Contract N66001-95-C-8636.

References

1. Carriero, N., and Gelernter, D. 1990. *How To Write Parallel Programs: A First Course.* Cambridge, Mass.: The MIT Press.
2. Carzaniga, A.; Picco, G. P.; and Vigna, G. 1997. Designing Distributed Applications with Mobile Code Paradigms. In Proceedings of the 19th International Conference on Software Engineering.
3. Chess, D. Security Considerations in Agent-Based Systems. May 1996. First IEEE Conference on Emerging Technologies and Applications in Communications (eta-COM).
4. Gamma, E.; Helm, R.; Johnson, R.; and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. 185-193. Reading, Mass.: Addison Wesley Longman, Inc.
5. Gray, R.; Cybenko, G.; Kotz, D.; and Rus, D. May 1996. Agent Tcl.
6. Harker, P., and Ungar, L. A Market-Based Approach to Workflow Automation.
7. Hofmann, M.O.; Whitebread, K.R.; and McGovern, A. 1998. Mobile Agents on the Digital Battlefield. Second International Conference on Autonomous Agents (Agents ’98). Forthcoming.
8. IBM Tokyo Research Lab. Aglet API. 1998.
9. Kay, J.; Etzl, J.; Rao, G.; and Thies, J. May 1998. The ATL Postmaster: A System for Agent Collaboration and Information Dissemination. In Proceedings of the Second International Conference on Autonomous Agents ’98, Minneapolis/St. Paul, MN. Forthcoming.
10. Mitsubishi Electric Information Technology Center America, Horizon Systems Laboratory. Concordia White Paper. 1997.
11. Oracle Corporation. March 1995. Oracle Mobile Agents Technical Product Summary, Oracle White Paper.
12. Ordille, J. When Agents Roam, Who Can You Trust? May 1996. In Proceedings of the First Conference on Emerging Technologies and Applications in Communications. Portland, Oregon.
13. Perkins, C. E. Mobile IP. May 1997. IEEE *Communications Magazine* 35(5): 84-99.
14. Sun Microsystems, Inc. Java IDL Core Platform Guide. 1998.
15. Tambe, M. Agent Architectures for Flexible, Practical Teamwork. 1997. In Proceeding of the Fourteenth National conference on Artificial Intelligence ‘97 (AAAI-97). Providence, RI.
16. Teraoka, F.; Uehara, K.; Sunahara, H.; and Murai, J. VIP: A Protocol Providing Host Mobility. *Communications of the ACM*, 37(8): 67-75, 113, August 1994.
17. Whitebread, K., and Jameson, S. October 1995. Information Discovery in High-Volume, Frequently Changing Data. *IEEE Expert, Intelligent Systems & Their Applications* 10(5):5153.