National Aerospace and Electronics Conference (NAECON)
Dayton, OH, October 10-12, 2000.

COMPARING PERFORMANCE OF STATIC VERSUS MOBILE MULTIAGENT SYSTEMS

SCOTT A. O'MALLEY, ATHIE L. SELF AND SCOTT A. DELOACH

Air Force Institute of Technology, Department of Electrical and Computer Engineering, WPAFB, OH 45433-7765

Abstract. This paper analyzes the performance differences between static and mobile multiagent systems. To do so, we developed solutions to a distributed text search problem, each using a different approach to multiagent systems (static versus mobile) on an isolated test network. Changes were then made to the agent environment, various constraints applied, and the resulting effect on the systems measured. Each system was evaluated using a number of performance metrics to demonstrate the strengths and weaknesses of the respective approach.

Key Words: Multiagent Systems, Mobile Agents, Static Agents

1. INTRODUCTION

Over the past few years, multiagent systems have grown in popularity as a viable solution to complex, distributed information systems. The largest "open system" is the Internet, with new content being generated every day. The distributed nature of the information on the Internet lends itself to multiagent solutions. Multiagent systems are reported to have advantages such as faster execution time, less communication bandwidth, and greater reliability. The type of agents employed, either static or mobile, also have their own unique set of advantages and disadvantages.

Today, a large body of quantitative data does not exist upon which system designers can base their "agent versus non-agent" system design decisions [3]. Likewise, once an agent approach has been selected, there is no data supporting the decision on the type of agents to use, mobile or static. Both of these design choices are subjects of current research at the Air Force Institute of Technology (AFIT).

The purpose of this paper is to collect quantitative data to help software engineers make the correct design choice when they must select a mobile or static multiagent system design. We begin this process by analyzing the performance differences between static and mobile multiagent systems. To perform this analysis, we developed static and mobile agent solutions to the problem of distributed file content search. We then modified the agent environment, applied various constraints, and measured the resulting effect on the systems. Each system was evaluated using a number of performance metrics to demonstrate the strengths and weaknesses of each approach.

The remainder of the paper is structured as follows. Section 2 discusses the distributed file content search problem the agent systems were designed to solve. Section 3 covers the agent environment used in testing. Section 4 presents the methodology used to develop the systems and explains decisions made during the design of each approach. Next, Section 5 describes the experiments as well as the environment in which the agents worked. Section 6 presents the results of the experiment while Section 7 discusses those results and possible future research.

2. PROBLEM DESCRIPTION

The distributed content search problem was devised with the Internet in mind. The premise of the problem is that information technology users have vast amounts of information at their disposal distributed across their organization's data storage infrastructure. As these technology users perform their jobs, they are required to research certain topics as they generate documents. To assist these users in their day-to-day work, management has decided to have a "content search agent" developed. The goal of the "content search agent" is to search through the files stored within the data stores and report back the location of files that have a high probability of containing information related to the user's research.

To simplify the problem, the following assumptions were made:

- The files to be searched were ASCII text files, including files ending with .txt, .html, .rtf, and .dat
- The search to be used was a simple string matching search

- The number of occurrences of the string in the document was the probability of relevance

Additional requirements included the ability to get interim results on demand and the ability to cancel a search at any time.

3. AGENT ENVIRONMENT

For an agent to operate within an organization's information system infrastructure, they require certain services from the infrastructure. The FIPA Agent Platform is an example of agent enterprise architecture. As described by Odell [4], several enterprise-related issues must be addressed by the agent enterprise architecture:

- *Agent Platform* – The agent platform is an environment in which the agent can be deployed.
- *Agent Management System* – The agent management system is an agent that manages the access and use of the agent platform.
- *Directory Facilitator* – The directory facilitator is a "yellow pages"-like directory service that advertises the services the agents within the system can provide.
- *Agent Platform Security Manager* – The APSM maintains the security policies for the platform. The APSM also is responsible for negotiating access requests for agents with other APSMs.
- *Agent Resource Broker* – The agent resource broker that maintain and broker software services provided by non-agents.
- *Wrapper Agent* – The wrapper agent is an agent that can communicate with non-agent software, allowing agents to interact with the non-agent software.
- *Agent Communication Channel* – The agent communication channel allows agents to exchange information between one another concerning services and communication messages.

Odell suggests that at a minimum, an agent platform should provide at least the first three capabilities. In our study, we used an agent platform called Carolina. Carolina is currently in development at the University of Connecticut as part of the Multi-Agent Distributed Goals Satisfaction project. AFIT, the University of Connecticut, and Wright State University are conducting this research jointly.

Carolina is a mobile agent system written in Java, much like Concordia, Odyssey and Voyager [5]. The current version of Carolina provides the agent platform, agent management system and an agent communication channel. Carolina also provides an abstract agent class called BaseAgent, which provides the basic capabilities for Carolina agents. The BaseAgent class provides the capability to read messages from and write messages to the Carolina's Message Directory, move from one Carolina server to another and perform whatever job is deemed necessary for the system. Agents are identified by agent type and a unique identification number (ID). A Carolina server keeps track of local agents running on it and forwarding addresses for agents that have moved to another host.

Carolina uses TCP/IP socket connections for the passing messages and agents between servers. The Message Manager listens to port 16000 while the Agent Manager listens to port 15000. When messages or agents are sent, the sending system opens a port and sets the destination port to be the appropriate port on the receiving machine.

Communication between agents running on Carolina servers is accomplished by passing serialized Java objects. Messages are posted to the server using the MessageManager Class. The Message Manager decides whether the message is for a local agent or an agent that has moved to a remote host. If the agent is local, the message is put into a message directory where it remains until the agent it is addressed to reads it from the server. Carolina agents must constantly poll the server to see if they have any messages waiting for them.

Messages can be addressed by agent ID or by agent type. The former allows for the interaction between specific agents, while that latter allows an agent to communicate with any other agent of a particular type without requiring knowledge of a specific agent. For messages sent by agent ID, the message is sent to the server where the agent is active and put in the message directory until the agent with the correct ID reads it. However, when messages are sent by agent type, any agent of that type can read the message. Once the message is read it is removed from the message directory and no other agents of that type can read the message. A service broker is a good example of how these two addressing methods work. When a new agent comes into the system, it can announce itself and the service it provides to a broker agent. The agent ID would be one piece of information recorded by the broker. When another agent requests a list of service providers from the broker, it will receive a list of agent IDs and hosts. The requesting agent can then send messages directly to the agent that provides the required service.

In Carolina, mobility is handled by Java's object serialization capability, which takes the identity and state of a class and encapsulates it. Once serialized, the state and identity is passed to another agent platform and reconstructed [5]. In Carolina, the program code for each type of mobile agent resides on each server so the code does not have to be passed when an agent moves.

4. SYSTEM DESIGN

To design the two systems, we used an agent-oriented methodology for developing multiagent systems called Multiagent System Engineering (MaSE). The basic outline of MaSE is shown in Figure 1 [1, 2, 6]. This methodology describes the analysis and design of multiagent systems in seven steps.

To assist software engineers in using MaSE, the AFIT Agent Lab has also developed a computer-based tool, called agentTool. During the design of our systems, agentTool implemented only the design phase of MaSE (it has since been extended to cover both the analysis and design phases). The agentTool environment greatly assists the designer in creating agent classes and defining the interactions between agents.

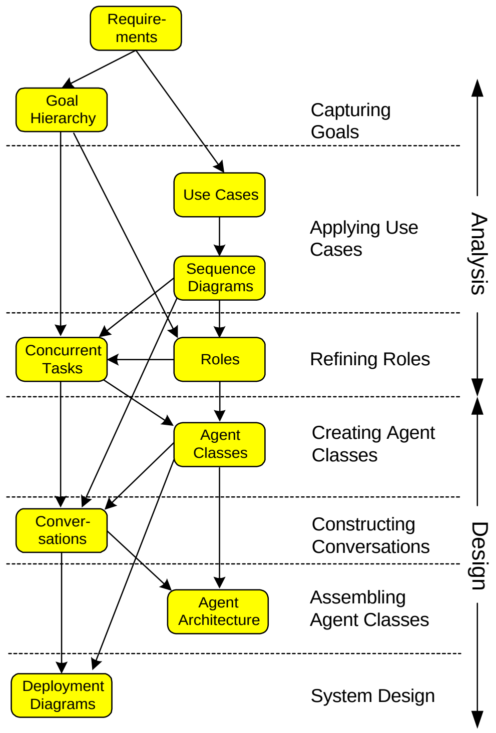

Figure 1: MaSE Methodology

These interactions become agent conversations that are defined using coordination protocols. These protocols describe the possible sequences of messages that may be passed between agents to achieve coordination. Conversations between agents in MaSE are defined by state machine based representation. The implementation of conversations was an integral part of the static agent system.

4.1. Static Agent System Design

For the static agent system, the system was implemented with two extensions of the Carolina's BaseAgent class; these agents were termed auxiliary because they were only required to read messages from and write messages to Carolina's Message Directory. The auxiliary agents created instances of the AgentBody class. AgentBody itself is abstract; the concretized classes that extend the AgentBody class (Search System and Researcher) contained the functionality of the agents, through conversation management and methods. The standard configuration had a SearchSystem Agent and a Researcher Agent located on each machine on the network that had the Carolina platform running. A diagram depicting the overall system architecture for the static system is shown in Figure 2.

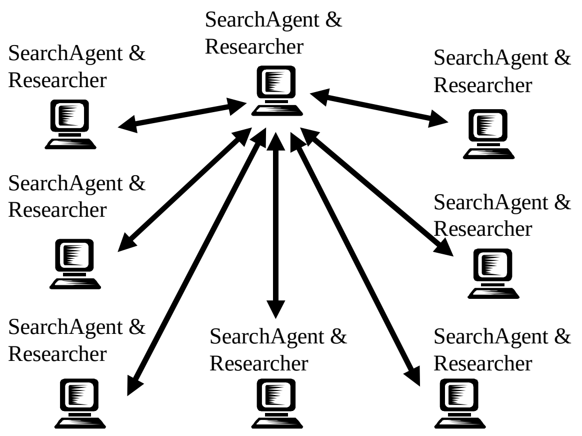

Figure 2: Static Agent System Architecture

For intra-agent communication, the concept of a MaSE conversation was maintained. A *conversation* is the definition of valid sequences of messages that the agent uses to communicate. For example, when it was necessary for a System Agent to execute a search, it started conversations as Java threads and sent messages to each of the Researcher Agents of which it had knowledge. For each agent, a conversation was initiated to allow asynchronous execution of the Researcher Agents. The conversation provided an identifier, the conversation ID, for the messages involved in the conversation, so that the messages went to the correct agents. Each side of the conversation would generate a unique identifier and pass it along with the messages it generated. The conversations interacted with their parent agents through method invocation.

A typical conversation flow is captured below in the sequence diagram of Figure 3. A search begins with an initial request to other SearchSystem agents for information about their local Researcher agents. This exchange is necessary because the only SearchSystem agent that a Researcher registers, or deregisters, with is the SearchSystem on the same local host. After the SearchSystem agent knows about the "foreign" Researcher agents, the SearchSystem sends "request search" messages to each of the Researchers.

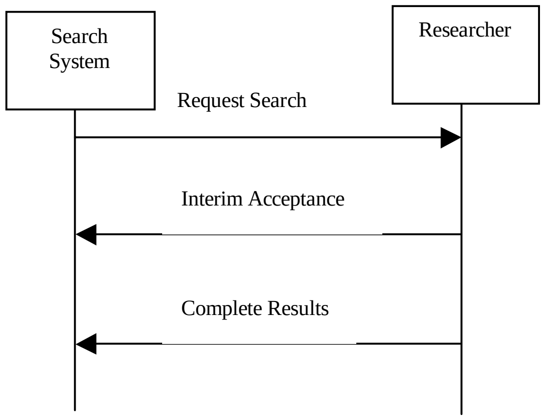

Figure 3: Search Request Sequence Diagram

The return of the "Interim Acceptance" message is necessary so that the Researcher knows with whom it is conversing. This information is stored within the messages that are passed, but the conversation ID cannot be generated until the first message is received. The responder's conversation ID is required in the two systems operation described in Section 2. In either case, the user only interacts with the local SearchSystem agent. The SearchSystem agent then forwards the user's request to the Researcher agents. Because these messages are only appropriate within conversations, and not as conversation initiators, the SearchSystem must know the conversation identifier so the message can be routed to the appropriate conversation.

4.2 Mobile Agent System Design

Mobile agents were also implemented using an extension of the BaseAgent class in Carolina. The mobile system consisted of two types of agents: MobileAgentManagers and MobileSearchAgents, which were the agents that actually moved and searched throughout the system. There was a single MobileAgentManager, which ran one of the network's Carolina servers.

Basic operation of the mobile agent system is as follows. Once a search request is received, the MobileAgentManager instantiates one or more MobileSearchAgents to accomplish the search task. The search string and a list of machines are passed to each MobileSearchAgent. The MobileSearchAgents then move to the machines on their list and search them. Once their list is empty, the MobileSearchAgent returns home and sent a final results message to the MobileAgentManager. A diagram showing the setup of the Mobile Agent System is shown in Figure 4.

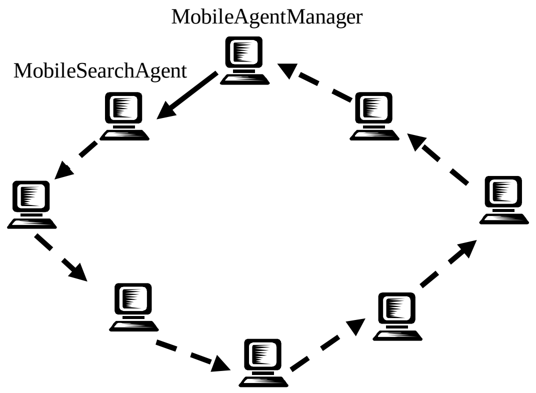

Figure 4: Mobile Agent System Architecture

All messages sent to MobileAgentManagers are sent by type and hostname since the MobileSearchAgents knows the host they came from. Messages sent from the MobileAgentManager to a MobileSearchAgent are sent by agent ID since when a MobileSearchAgent is created it passes its ID to its MobileAgentManager. A typical flow of message traffic for the mobile system is shown in Figure 5.

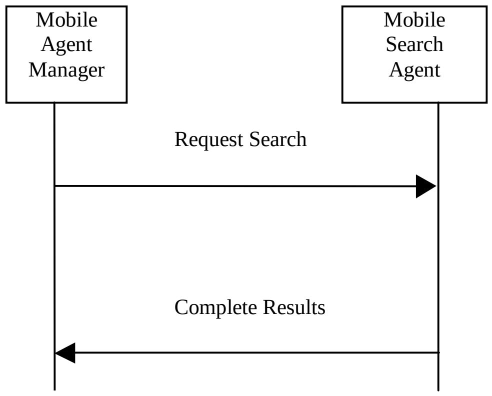

Figure 5: Search Request Sequence Diagram

There is no requirement for a MobileSearchAgent to initiate communications with the manager when it is on a remote host. However, to obtain interim results or to cancel the search, the MobileSearchAgent has to periodically check the server to for these two requests, which appear as messages. When a MobileSearchAgent receives an interim results message, it sends all of its search results, for each host it has searched so far, back in a message. If a cancel search message is received, the MobileSearchAgent stops searching, immediately returns home and sends a final results message to the MobileAgentManager.

Since Carolina keeps track of where agents have moved, the MobileSearchAgents does not need to inform its MobileAgentManager of its current location. If a MobileAgentManager needs to send a message to a MobileSearchAgent, all the MobileAgentManager has to do is post a message to the local Carolina server and the message is forwarded until it reaches the machine where the agent is operating.

5. EXPERIMENT

AFIT's Bimodal Computer Laboratory, or *Pile of PCs*, was selected for the experiment. The Pile is an isolated network of twenty computers running either Linux 6.0 or Windows 2000. It is running a 100 Mbps Ethernet backbone with fiber channels running to the servers. The PCs are Pentium III 600 MHz processors with 128 MB of RAM. To capture the communications between agents, Exdump v0.2 [7] was used. Exdump tracks TCP/IP packets from machine to machine, as well as captures packet size and timing information.

For the execution time experiments, the tests were conducted on the eight machines running Windows 2000. Each machine was loaded with five text files within a specific subdirectory of the local hard drive. The size of the files ranged from 667 kilobytes to 2.31 megabytes.

6. TESTING AND ANALYSIS

During the testing phase, several agent configurations were evaluated. The first sets of tests were designed to acquire quantifiable data on the performance of static and mobile agents. Several scenarios were set up for the static agent system while a second set of tests were designed for the mobile agent approach.

6.1. Static Agent Tests

Two implementations using static agents were developed to test the impact of using conversations. The first approach reported interim results automatically. The intention of this design was to have the results close to the user. With the results close, a request for interim results or a cancel could be posted immediately. Messages that followed from a cancel search just wrapped up the conversation.

The second approach attempted to minimize the number of messages passed, and hence, the amount of processor time lost waiting for messages. This approach made use of an acceptance message to let the SearchSystem agent know about the responder side of the conversation. This acceptance message did not include any of the result data. If the user requested interim results or decided to cancel the search, the SearchSystem agent sent messages to the Researcher agents informing them of the intentions. The Researcher agent responded with the current results it had been accumulating. If the message requested interim results, the agent continued the search process on the current file until completion.

The time required to search the files on a single machine took the first static implementation over fifteen minutes to execute while the second approach took only thirteen seconds. The impact of the extra message handling proved to be quite significant. To post and receive messages in a timely manner, both implementations had to constantly poll the Carolina server. To prevent this polling from dominating the processor, the polling thread is forced to give up control of the processor. The conversations are executed as state machines. When a conversation is waiting for a message it remains in its current state. The state machine is implemented as a while loop with a switch statement based on the state of the conversation. This thread continues cycling through the same state until a new message arrives, which causes significant wasting of processor time when multiple conversations are executing simultaneously. Bringing multiple platforms into the mix only exasperated the situation, as the amount of processing time given to each process varies.

Average search times of the static agents (using the second implementation) on different numbers of machines are shown in Table 1. For every machine added to the search, approximately five seconds was added to the execution time. This overhead can be accounted for in several ways. First, because the Carolina environment does not currently provide a broadcast capability, messages sent to each Researcher agent must be sent consecutively. Since there were multiple agents in the system, with each requiring its own conversation and thus a separate

Table 1: Static Agent Search Execution Time

Machine Config	Avg	Min	Max	St Dev
1 Machine	12.645	10.717	15.195	1.601
2 Machine	17.008	15.244	18.440	1.603
3 Machine	21.315	21.094	21.535	0.157
4 Machine	26.412	24.259	27.585	1.491
5 Machine	32.117	30.329	33.614	1.585
6 Machine	36.678	36.369	38.272	0.606
7 Machine	42.742	42.289	44.041	0.511
8 Machine	49.302	48.619	51.424	1.144

thread, the messages probably were not forwarded without interruption. Additionally, since there were multiple processes running on each machine, the processor was being shared by multiple agents when the message was received. The method the operating system uses to share the processor played a major role in the conversation overhead.

The second way to account for the overhead is based on the time required to process the message. As a message is received by Carolina, it is immediately placed in a container for the conversation to find it. The conversation, however, only looks for messages at certain times in its execution loop. If a message has not arrived for it at that time, the conversation loops through its state machine. Since the state will not have changed, it will come to the same line of code to get a message, and the conversation will check the message container. The conversation thread could repeat this loop many times before control was given to another process. Because the agent—which polls the server for messages—is running in a separate thread, it cannot handle a new message until the conversation releases the processor.

6.2. Mobile Agent Tests

Test runs using mobile agents were performed incrementally. The first test consisted of a single mobile agent searching one machine. Then additional machines were added until all eight machines were searched. Next, two mobile agents were used searching from two to eight machines. We continued this pattern until eight agents were used to search eight machines. As expected, the total search time decreased by up to 50%, along with the network message traffic, as the number of agents increased. Table 2 shows average search times of the mobile agent system in its various configurations.

6.3. Communications Impact

Total network traffic for the static agent system, which passed five messages per search sequence per number of machines, required a total of 400 kilobytes. When all eight machines were being searched, a total of approximately 3.2 megabytes of message traffic was generated. In the mobile system, traffic was generated each time an agent moved from one machine to another. Additionally, the payload of the messages grew with respect to the number of machines that had been searched. The single mobile agent (searched all eight machines by itself) generated roughly 1.1 megabytes of traffic while eight mobile agents (searching the same eight machines) generated approximately 1.2 megabytes of network traffic.

7. CONCLUSIONS

Initially, we expected the static agent system to outperform the mobile agent system. In the first scenario, when one MobileSearchAgent performed a serial search of all the machines, we expected the static implementation, which was running in parallel, to complete it in a fraction of the time proportional to the number of machines being inspected. This in fact was the case with more than two machines.

However, once multiple MobileSearchAgents were employed, the mobile agent system was faster in all cases. As seen in the 8 MobileSearchAgents configuration, the time to execute the search was almost five times faster. We strongly suspect that the reason the static system was slower was due to the amount of overhead (both in communications and wasted processor time) necessary for maintaining the conversations. Using this particular implementation forced the static solution into dividing its processor time among too many threads.

Table 2: MobileSearchAgent Execution Times

	1 MobileSearchAgent				2 MobileSearchAgents				4 MobileSearchAgents				8 MobileSearchAgents			
Mach Config	Avg	Min	Max	St Dev	Avg	Min	Max	St Dev	Avg	Min	Max	St Dev	Avg	Min	Max	St Dev
1 Mach	7.220	7.052	7.412	0.128												
2 Mach	14.985	14.844	15.055	0.067	9.867	9.525	10.016	0.165								
3 Mach	24.744	21.925	26.388	1.663	14.582	14.423	14.702	0.083								
4 Mach	31.767	29.332	33.128	1.294	17.742	17.538	17.878	0.119	10.195	10.025	10.406	0.126				
5 Mach	40.947	40.753	41.394	0.192	23.232	22.896	23.976	0.399	13.546	12.990	13.772	0.276				
6 Mach	46.903	46.903	48.706	0.558	25.757	25.490	26.081	0.211	14.947	14.854	15.083	0.074				
7 Mach	53.814	53.814	55.617	0.576	30.017	29.867	30.217	0.105	16.101	15.975	16.205	0.085				
8 Mach	61.446	61.446	62.587	0.365	33.830	33.542	34.413	0.258	17.986	17.828	18.118	0.109	9.991	9.615	10.607	0.364

In contrast, the mobile agent system had a more finely tuned communication framework. A MobileSearchAgent only needed to check for messages at certain times within its execution. On the other hand, the static agent system had to implement a resource intensive communication framework to maintain their conversation structure.

Further, the amount of TCP/IP traffic placed on the wire by the 8-MobileSearchAgent parallel tests was more than 50% smaller than the traffic generated by the static implementation. Having mobile agents pass results along with its current stack state, proved not to be as costly in terms of network bandwidth as expected. This was the case even though the mobile agent implementation actually created extra overhead by having the MobileSearchAgent return home after searching its list of machines instead of just sending a final results message back to the MobileAgentManager. These results were quite opposite of what we expected at the onset of the experiment. The packets that both implementations sent contained serialized Java objects, which contributed to similarity in message size.

Even though our research seems to show that, in this case, the mobile agent system is a better choice (mainly due to an inefficient static agent system design), our research is certainly not all-inclusive. Besides redesigning the static agent conversation mechanism, an interesting experiment would be to expand the problem to involve service negotiation. In the static agent implementation, the Researcher agent could negotiate based on the amount of processes running on the system at the time of the search. Perhaps high priority jobs could get processor time before lower priority processes. On the mobile side, the agent could determine whether or not there is enough processor time to complete a search. If not, the agent could move on to the next machine and return to the current machine later.

Even though we look at only one problem and a limited number of cases, we believe experiments like ours are an important step towards the objective evaluation of multiagent systems. Having quantifiable data on which to base a design decision makes the justification of such decisions stronger.

8. ACKNOWLEDGEMENTS

This research was supported by the Air Force Office of Scientific Research. The views expressed in this paper are those of the authors and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the US Government.

9. REFERENCES

[1] S. A. DeLoach, “Multiagent Systems Engineering: a Methodology and Language for Designing Agent Systems,” *Proceedings of Agent Oriented Information Systems* (AOIS99), Seattle Washington, May 1999.

[2] S. A. DeLoach and M. Wood, “Developing Multiagent Systems with agentTool,” in *Proceedings of The Seventh International Workshop on Agent Theories, Architectures, and Languages,* Boston, Massachusetts, July 2000.

[3] N. R. Jennings, “On Agent-based Software Engineering,” *Artificial Intelligence*: Vol 117, pp. 277-296, 2000.

[4] J. Odell, “Considerations for Agent-Based Technology,” *Distributed Computing,* August 1999.

[5] D. Wong, N. Paciorek, D. Moore, “Java-based Mobile Agents,” Communications of the ACM, March 1999.

[6] M.F. Wood and S.A. DeLoach, “An Overview of the Multiagent Systems Engineering Methodology.” in *Proceedings of the First International Workshop on Agent-Oriented Engineering (MOSE).* Limerick Ireland, June 2000.

[7] Exdump – v0.2 By PolarRoot, http//exscan.netpedia.net/exdump.html

10. AUTHOR BIOGRAPHIES

First Lieutenant Scott A. O’Malley is a graduate student in the Computer Science and Engineering department at the Air Force Institute of Technology (AFIT). His thesis research involves multiagent system specifications and agent-oriented software engineering. Before coming to AFIT, Lieutenant O’Malley was stationed at Hickam Air Force Base from 1997 to 1999. Lieutenant O’Malley received his BS in Computer Science from University of Iowa in 1997.

Captain Athie L. Self is a graduate student in the Computer Science and Engineering department at the Air Force Institute of Technology (AFIT). His thesis research is on how to design and specify mobile agents using the Multiagent Systems Engineering Methodology (MaSE). Before coming to AFIT, Captain Self was working for the Defense Information Systems Agency (DISA) at the Pentagon from 1996 to 1999. He has also been stationed at the

Headquarters Standard Systems Group. Captain Self received his BS in Computer Science from the State University of New York College of Technology at Utica/Rome in 1993.

Dr. DeLoach is an Assistant Professor of Computer Science and Engineering at the Air Force Institute of Technology (AFIT). His research interests include design and synthesis of multiagent systems, knowledge-based software engineering, and formal specification acquisition. Prior to AFIT, Dr. DeLoach was at the Air Force Research Laboratory from 1996 to 1998. He has also been stationed at Headquarters Strategic Air Command and the Aeronautical Systems Center. Dr. DeLoach received his BS in Computer Engineering from Iowa State University in 1982 and his MS and PhD in Computer Engineering from the AFIT in 1987 and 1996.