

Evaluating Agent Architectures: Cougar, Aglets and AAA

Ian Gorton, Jereme Haack, David McGee,
Andrew Cowell, Olga Kuchar, Judi Thomson

Information Sciences and Engineering,
Pacific Northwest National Laboratory,
Richland, WA 99352, USA

Abstract. Research and development organizations are constantly evaluating new technologies in order to implement the next generation of advanced applications. At Pacific Northwest National Laboratory, agent technologies are perceived as an approach that can provide a competitive advantage in the construction of highly sophisticated software systems in a range of application areas. To determine the sophistication, utility, performance, and other critical aspects of such systems, a project was instigated to evaluate three candidate agent toolkits. This paper reports on the outcomes of this evaluation, the knowledge accumulated from carrying out this project, and provides insights into the capabilities of the agent technologies evaluated.

1 Introduction

Agent technologies are a potentially promising approach for building systems that require intelligent behavior from a collection of collaborating, autonomous software components. With their roots in the field of distributed artificial intelligence [1], agent technologies promote a variety of abstractions, protocols and mechanisms that aim to make it easier to construct distributed, collaborating systems [2]. While some of these technologies have been productized, many remain the outputs from various leading research laboratories and universities. This creates a complex landscape for organizations wishing to exploit agents in their applications.

Researchers in the Information Sciences and Engineering division and elsewhere at the Pacific Northwest National Laboratory (PNNL), have considerable interest in using agent technologies across a number of application

areas. These application areas include next generation energy management systems, mining information from massive data sets, intelligent multimodal interactive systems, and cyber security. While each application area has a different set of requirements for the agent technologies, the common thread is the need for distributed, collaborating software components that collectively behave in a goal-driven manner.

Consequently, we evaluated a number of agent architectures that were likely to be of relevance to us. The three architectures were chosen based on the requirements for agent technologies of the PNNL research groups, based solely on their business needs and existing research collaborations. While it would have been desirable to evaluate other technologies such as FIPA-OS, ZEUS and Jack, we could not provide a business justification at the time the project commenced.

Selecting appropriate technologies and architectures for a given application domain is a key activity in the software engineering process [3, 4]. In practice, many projects fail or greatly exceed budget because they base their solutions on technologies that do not perform as anticipated.

The aims of this evaluation project were as follows:

- Qualitatively investigate various features of each technology to assess capabilities against the requirements of the various application areas.

- Quantitatively assess each technology in terms of its performance and scalability.

- Assess the potential for code reuse and interoperability between applications written using different agent infrastructures.

The three technologies chosen for evaluation were AAA [5], Aglets [6] and Cougar [7]. This represents a mix of a production-ready technology (Cougar), an established agent technology research vehicle (Aglets) and a relatively new technology from a leading research university (AAA). This provides a further dimension of interest in the comparison. The evaluation consisted of constructing an identical example application with each technology, and then comparing those applications both qualitatively and quantitatively. The results of the evaluation reveal several important differences in the capabilities of the three agent systems. The results also highlight some of the inherent complexity of the technologies, the variable levels of implementation maturity, and the different performance characteristics.

The following sections briefly describe each of the technologies evaluated, the test application architecture, and the architecture and design used to create the test application with each technology. The comparative results are then presented, including the performance achieved on a consistent test-bed, and an assessment of some of the important features of each agent platform.

The paper concludes with a discussion of related work and an evaluation of the effectiveness of our approach.

2 Technology Overviews

2.1 Aglets

The Aglets software development kit and framework were originally devised at IBM's Tokyo Research Laboratory. The source code was subsequently released as open source available under the IBM Public License. The fundamental differentiating technology in the Aglet architecture is agent mobility.

The Aglet object model (AOM) is designed to exploit the strengths of Java, including platform independence, secure execution, dynamic class loading, multithreaded programming and object serialization. At the same time, Aglets provide additional capabilities for features such as resource control, protection and object ownership of references, and support for preservation and resumption of execution state.

The basic elements of the AOM comprise three key abstractions.

Aglet: a mobile java object that can move around aglet-enabled hosts within an environment.

Proxy: a representative of an aglet that serves as a shield, protecting direct access to the public methods and providing location transparency for the aglet.

Context: the aglets workplace, a stationary object that provides a uniform execution environment. Many aglets can exist in a single context, and a single computer may run multiple contexts.

The aglet package consists of a set of Java API's and a run time environment. Each aglet is a separate class - a master class is written and loaded into the Aglet Tahiti server where the aglet is created and begins execution. For mobility, each machine must be running a Tahiti server.

An aglet has a number of fundamental operations. There are only two ways in which an aglet can be created – it may be instantiated (created) or it may be copied from another aglet (cloned). Aglets may also be removed when they are of no further use (disposal). Aglets can also be temporarily halted (deactivation) and restored (activation) if required. Finally, an agent can be pushed (dispatched) to a new destination context (local or remote) and retrieved from that context (retraction).

A messaging framework supports inter-aglet communication. The principle method for aglet communication is through message passing. A simple event scheme means that it is only necessary to implement message handlers in an aglet for those kinds of messages that are expected. A message is a type of object, characterized by its 'kind', denoted by a string. In addition, atomic arguments may be attached or multiple arguments wrapped as an object. The object-based messaging framework provided is location-independent, so that two aglets need not exist within the same context or in the same location to communicate. Both synchronous and asynchronous messaging is provided; blocking operators may halt execution of an aglet until its companion replies to the message, or execution can continue through the use of a *dummy* object. Aglets also support multicasting in a publish-subscribe manner. Aglets can subscribe to one or more multicast messages and listeners for those messages are then implemented to handle occurrences.

2.2 Cougaar

Cougaar is a Java based agent architecture developed for DARPA under the Advanced Logistics Program. It is designed to solve logistics planning problems using a cognitive model based on human reasoning. Cougaar is an open source technology implemented as a set of Java APIs.

A Cougaar agent consists of a blackboard that facilitates communications (known as the *plan*), and operations modules called *plugins* that communicate with one another through the blackboard and contain the logic for the agent's operations.

Cougaar agents may be created statically using configuration files, or dynamically using an *Agent Server*. The server publishes an interface that allows other agents and components to create, destroy, or reconfigure agents within its boundaries. Regardless of the creation method, all agents have a set of *plugins* and a defined place in the agent community that declares the agent's name, roles, and relationship to other agents.

In addition to a blackboard that is internal to each Cougaar agent, the architecture supports one-way, asynchronous, and one-to-one communications through a *MessageTransportServer*. This server, along with the *NameServer*, allows a message to be sent to an arbitrary agent based on its identifier. Finally, although agents may not directly address the blackboard of another agent, two agents that have a defined relationship may share information directly by acting as *Organizational Assets* for one another.

Cougaar supports inter-agent communications through a mechanism known as *allocation*. An agent will create a task and then allocate this task to an asset. If the asset is another agent, then the task will appear on its blackboard

where a *plugin* subscribing to the task will be notified. The receiving agent then performs the behavior required and sends the results back by changing the status of the task allocation's results, which causes a notification to be sent to the publisher.

The *plugin* execution model was designed so *plugins* can run in parallel. In practice however they run using a shared thread. A *PluginScheduler* decides when an individual *plugin* should run. *Plugins* must explicitly spawn separate threads for functionality that may occupy the shared thread for extended periods of time.

2.3 Adaptive Agent Architecture(AAA)

The Adaptive Agent Architecture (AAA) is a multi-broker, multi-agent system architecture. AAA was developed in the Center for Human Computer Communication at the OGI School of Science and Engineering of the Oregon Health & Science University to promote active research in multi-agent systems semantics and communication. It extends and improves upon version 1.0 of the Open Agent architecture developed at SRI.

Major goals for the AAA are:

- Support for a large variety of hardware platforms, operating systems, and programming languages via a lightweight API.
- The use of Speech Acts semantics driven communication primitives that are passed as messages in the AAA Agent Communication Language (ACL), which is backwards compatible with OAA version 1.0 and provides a more thorough semantics for middle-agent and group communication [8].
- An easy-to-implement Java Interface that automates the process of converting Java data types and collections into an ACL-supported content language such as Prolog Horn clauses or XML.
- Fault-tolerance achieved through the execution of the joint commitments established via the Speech-Acts based communication, resulting in the ability of *Facilitators* (or other agents) to re-establish connectivity with agents that were registered with another *Facilitator*, when communication issues arise [8]
- A blackboard and accompanying algorithms for unification of terms on them, which can be used to create agents with a persistent data store. The *Facilitator* uses a blackboard internally in order to implement its message passing architecture, and agents can use their own blackboards for any purpose.

On initialization, AAA agents register their capabilities with the Facilitator. Messages are brokered by the Facilitator based on the registered capabilities. After an agent has connected to the Facilitator and registered its capabilities, it can then communicate with other agents in any of four ways, as follows:

- **Requests:** These can be either synchronous (the agent thread blocks until the reply returns) or asynchronous (a callback is registered to listen for the reply and is called when the reply arrives). Requests are usually brokered by the Facilitator and routed to capable agents. The Facilitator can accumulate replies, manage distribution of complex action requests to capable agents (agent A do X or agent B do Y), and insure deadlines are met.
- **Informs:** Declarative statements can be made by any agent. If these are sent to a Facilitator then the middle agent will transmit the *informs* to any agent that has registered similar interests.
- **Direct:** Agents may request a list of *capable* agents from the *Facilitator*, who replies with a list of agent addresses. The agent can then communicate directly with any agent in the list without *Facilitator* intervention. Such communication, once brokered, can remain in AAA ACL (i.e., Requests and Inform) or the agents can negotiate a higher performance protocol (e.g., sending Java Objects or other binary directly).

3. Test Scenario and Process

In order to compare and contrast these three technologies, a test scenario was devised, which could be implemented with each. This scenario would provide a basis for direct comparison. The main objectives of the test scenario were to facilitate the:

1. Comparison of the architectures that agent technologies support/promote
2. Comparison of the performance of the agent technologies under test

The scenario is based upon a synthetic stock purchasing application, in which a multi-agent implementation intelligently decides which stock items to purchase from a set of proposed purchases and a budget. An overview is given in Figure 1.

The input is a series of discrete stock purchase requests, where each request comprises a monetary amount, the budget, and a collection of up to 10 individual stock items that the client wishes to purchase. Stock items have a

key, ranging from 1 to 1000, and a current price. A pool of 20 expert agents is dedicated to maintain expert knowledge about a portion of the range stocks. When asked by another agent, expert agents will reply with either a 'buy' or a 'don't buy' message.

Each expert agent is configured to provide advice on 1/10th of the total population of stock, and hence the test is designed with some overlap between the areas of expertise of the experts. For example, one expert agent can intelligently answer questions about items 0-99, another about items, 50-149, another about 100-199, and so on.

A purchase agent inputs a new purchase request, looks at the budget and calculates the total cost of the purchasing all the stock requested. It then optimizes the purchase in terms of the absolute number of requested stock that can be bought for the budget. Once it has decided which stock it will attempt to purchase, it sends a request to an expert agent that has knowledge about the quality of that stock (i.e. whether it's a good buy at this particular moment). The specific expert agent is selected from the ones available using some form of directory service, in which agents advertise their abilities.

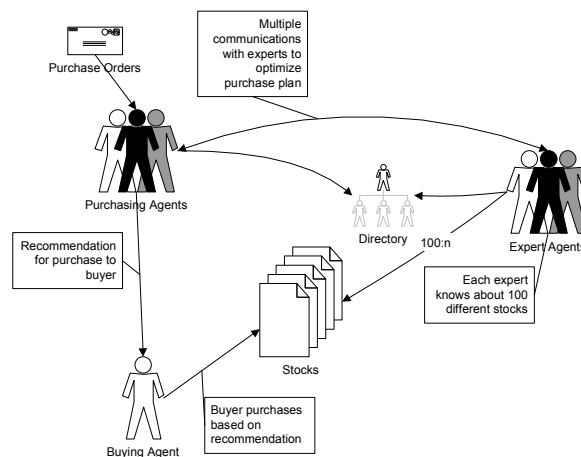


Figure 1 Test Scenario

The *expert* agent receives the request, and returns a message that either confirms a buy action, or indicates that this item should not be purchased. Internally, the expert agent randomly computes 80% 'buy' responses, and 20% 'don't buy' responses.

Depending on the responses of the *expert* agents, the *purchase* agent again attempts to optimize the purchase request, eliminating the latest 'don't-buy' opinions. This may require further communications with other *expert* agents to confirm that another stock item purchase is sensible. At some stage, poten-

tially after several iterations, the *purchase* agent will have a recommended set of stock to purchase. It then passes this result set to the *buying* agent. In this test scenario, the *buying* agent maintains statistics on the results for benchmarking purposes.

In order to provide a solid basis for performance comparison, a collection of 10,000 randomly created purchase requests were created. A solution to the test scenario was built with each agent architecture and installed on the same high performance Windows 2000 machine. The multi-agent system solution for each architecture ran on that system in one Java VM.

The same Java class was used in each solution to read the 10,000 purchase requests from a file and make these available to the *purchase* agent. The same code was also used to implement the *expert* agent behavior (80/20 random Boolean response) across solutions.

To measure performance, the *purchase* agent records the number of iterations needed to finalize a purchase. The *buying* agent also measures and records the time taken to produce each purchase. These measures are stored in memory and output to a log file at the end of the test, along with the total elapsed time to process the 10,000 inputs.

4 Solution Implementations

The following describes the three implementations. Each implementation was written by the same software engineer, and much code was reused across implementations to ensure as much consistency as possible.

4.1 Cougar Architecture

The Cougar solution comprises 2 agents, as depicted in Figure 2. The *Purchase* agent exploits Cougar's blackboard mechanism to load all the purchase orders (POs) on to the blackboard, which is shared by 4 agent plug-ins. When a new PO appears on the blackboard, the *Plan* plug-in is notified. It selects which stock to attempt to buy given the budget, and places the result on the blackboard. This causes a notification to be sent to the *Ask* plug-in. It selects an appropriate *Expert* agent for each stock in the requested list, and allocates a task to the *Expert*, which causes it to appear on that agent's blackboard.

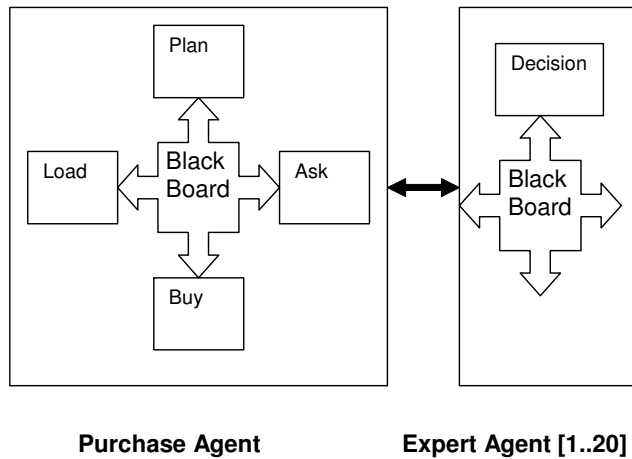


Figure 2 Cougaar Solution

When the *Expert* agent sees the task, it responds by updating the *allocation results* on the *Purchase* agent's blackboard. This causes a notification to be sent to the *Ask* plug-in. Once all the *Experts* respond, the *Ask* plug-in notifies *Plan*, and it decides whether the purchase is complete, or needs to be altered due to 'don't buy' responses from *Experts*. In the former case, *Plan* sends the completed order to the *Buy* plug-in. In the latter, *Plan* alters the selected stock list, and the whole process of asking *Expert* agents occurs again.

The evaluation of all the POs occurs concurrently. Each agent has its own thread, and all the agents run in the same JVM. An agent's *Plug-in Scheduler* coordinates the execution order of plug-ins in an agent. This order is non-deterministic, depending on the order of messages and notifications that are passed between the *Purchase* and *Expert* agents, and the plug-ins.

4.2 AAA Architecture

In the AAA solution, the agents reside in the same JVM and after the *Facilitator* brokers connections between the *Plan* and *Expert* agents, these agents are configured to communicate directly, which is potentially more efficient than using the AAA *Facilitator* to broker all communications.

The *Load* agent sends a direct *inform* to the *Plan* agent for each PO. *Plan* selects stock items to buy, and send an *inform* message to the *Ask* agent. On start-up, *Ask* sends a series of *request* messages to the AAA *Facilitator* to get references to each of the *Expert* agents. It then stores these locally in a data

structure, and uses this to select an expert to evaluate each stock item. This solution is depicted in Figure 3.

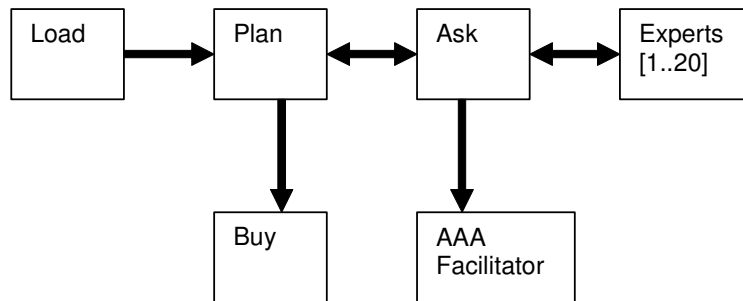


Figure 3 AAA Solution

Ask creates a thread for each stock item in the request, and these threads send *request* messages to their allocated *Expert* agent. Each *Expert*'s reply is handled by a AAA callback (*NotificationListener*), and when all replies are received, *Ask* sends an inform message to *Plan* that contains the results. If the order is complete, *Plan* sends an inform message to *Buy*. If the order needs work, *Plan* selects different stock items and cycles again through the stock evaluation process.

Since it is not possible to publish Java Objects on the AAA's shared blackboard, Java objects, such as purchase requests, must be converted into a supported content-language format, such as XML. Although a framework and helper classes exist to facilitate this conversion, due to time constraints we decided to implement the AAA solution as described above.

4.3 Aglets Architecture

The architecture for the Aglet's solution is depicted in Figure 4. As Aglet's does not support a shared blackboard, the application is structured as a set of communicating aglets, each of which has its own thread of execution. The *Load* aglet sends the input stream of POs to the *Plan* aglet. This is an asynchronous one-way communication. *Plan* inputs each PO and creates a list of selected stock items, which it asynchronously sends to *Ask*. *Ask* sends an asynchronous request to the *Expert Coordinator*, which replies with references to the appropriate *Expert* aglets. *Ask* then spawns a set of threads, one for each stock item, and these threads each communicate asynchronously with

an *Expert*. When all the threads receive a response, the *Ask* aglet asynchronously notifies *Plan* about the outcomes. *Plan* responds accordingly by either asynchronously sending the completed PO to *Buy*, or selecting new stock and repeating the *Expert* evaluation process.

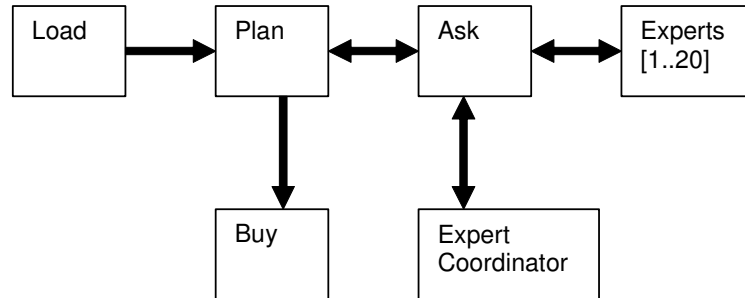


Figure 4 Aglets Solution

The solution co-locates all these aglets in the same JVM. In order to support asynchronous communications, aglets have request queues in which pending requests are placed. Hence the *Load* aglet effectively places all the POs in the request queue for *Plan*, which processes them serially and places the results in the *Ask* aglet’s request queue. From there, the processing hence proceeds non-deterministically due to the threading internally in *Ask*, contention for *Experts*, and the asynchronous notification mechanism used between *Ask* and *Plan*.

5 Evaluation

5.1 Technology Evaluation

As expected, each of the three technologies proved to be rather different in nature.

Table 1 briefly evaluates each against a core set of review criteria that we devised. These criteria were based on extensive technology and architecture evaluation experience, as reported in [3, 4], and have considerable overlap with the review criteria in [14]. We focused on a set that covers key architectural features and flexibility, as well as robustness and quality of supporting documentation. These were deemed the most important criteria for the research groups that were examining agent-based approaches.

- **Documentation:** How easy was it to learn the technology using the documentation? Was the documentation comprehensive, current and easy to navigate?
- **Scalability:** Can the technology exploit multiple processors on the same machine (scale-up), and a fully distributed deployment over multiple machines (scale-out)?
- **Ease of modification/distribution:** How easy is it to modify the solution/code to re-factor the functionality across different agent collections, or to physically locate agents in different locations? Does the technology support location transparency?
- **Architecture flexibility:** Does the technology promote a particular architectural style, such as shared blackboard, distributed memory, or publish-subscribe? Or does the technology provide flexibility in terms of matching the architecture to the problem domain?
- **Mobility:** Does the technology support mobile agents?
- **Inter-agent communication mechanisms:** Does the technology provide a flexible and comprehensive set of inter-agent communication mechanisms, including synchronous, asynchronous and multicast?
- **Robustness of implementation:** Did the technology fail during development or testing? Were certain mechanisms prone to failure?

Table 1 Qualitative Technology Comparison

Criteria	Cougaar	Aglets	AAA
Documentation	Very extensive documentation that is current and well maintained	Extensive documentation, adequate for most purposes.	Documentation is incomplete and out-of-date
Scalability	Solutions should scale across multiple machines. 1 thread per agent restriction inhibits internal agent concurrency.	Designed to scale seamlessly across multiple nodes that support Aglet container.	Solutions inherently support distribution. Replicated facilitator mechanism should make a large-scale deployment possible
Ease of modification and distribution of solutions	Not simple to re-factor solutions by modifying structure of plug-ins and agents. Code changes required.	Not simple to modify aglet structure, as sender must know explicit network location of destination aglets as there is	There should be no code changes necessary as all inter-agent communications must exploit distributed communications

Architecture flexibility	Plug-in architecture suggests an architecture style based on extensive use of shared blackboards	no name service. Aglet architecture forces fully distributed architectural style.	mechanisms. Flexible in terms of ability to easily mix shared blackboard and distributed inter-agent communications architectures.
Mobility	Supports agent mobility. A Plugin within an agent must decide when to issue the move, which agent should be moved (potentially itself) and which node it should move to.	Inherently supported	Moderately supported. ACL messages carry code and data (i.e. state).
Inter-agent communication mechanisms	Excellent support for flexible publish-subscribe based messaging based on changes on blackboard state.	Excellent support for synchronous, asynchronous and multicast inter-agents communications.	Supports synchronous and asynchronous inform and request speech acts, which can be direct or indirectly managed through the facilitator.
Robustness of technology implementation	Robust and reliable	Robust and reliable	A few advanced mechanisms are 'buggy'

In terms of raw performance, formal performance testing was carried out with each solution. The final tests executed the solutions 5 times on the same hardware platform. Results were averaged and analyzed to ensure that each solution experienced similar amounts of iterations in optimizing a purchase request. Table 2 presents the average performance measured in seconds for each solution to process 10,000 purchase orders, and the throughput obtained in messages per second.

Table 2 Performance Comparison

Agent Technology	Overall test time (secs)	Throughput (POs/sec)
Cougaar	35	286
Aglets	76	132
AAA	500	20

Cougaar's performance clearly benefits by being able to provide a solution to this problem using a blackboard-based architecture. Aglets provides just under 50% of Cougaar's performance. This is still a relatively impressive result given the non-shared memory solution. AAA suffers from the fact that its inter-agent communications mechanisms are not optimized for the case when agents are co-located in the same JVM. Passing and decoding messages as Prolog Horn clauses also incurs additional performance penalties.

It would be interesting to experiment and observe how these architectures could scale out to execute on multiple machines. Although we have not had time to fully test a distributed solution, preliminary tests indicate that the results would be somewhat closer in terms of throughput. Also, as indicated in

Table 1, the Cougaar solution would require considerable code changes to spread the processing of the *Purchase* agent across multiple nodes. In contrast, both the Aglet and AAA solutions can be distributed with relative ease.

Diversity: The lack of standard approaches in the agent arena is starkly exposed in these three toolkits. Cougaar has a somewhat eclectic architecture, in which generic agent classes are endowed with specific behaviors through *plugins*. This architecture has been realized due to the evolution of Cougaar over a number of years. The AAA and Aglets take a more traditional approach to extending Java to incorporate agent features. However the precise semantics and mechanisms used for agents and communications are very different. This makes it a non-trivial task for a developer to understand and move between different technologies.

Maturity: Cougaar has a considerable infrastructure that must be understood before applications can be successfully built. However the documentation is excellent, and the technology seems to be well implemented. AAA suffers from inadequate documentation, which makes it difficult to build applications whose behavior deviates too far from the examples provided. The solution to this problem was to use Internet mailing lists and to communicate with the developers of the toolkits. Aglets was especially difficult to configure and get running, but it proved reliable once 'tricks' had been learned. However part of the reason for this was its ability to support mobile agents, a feature that was not exploited in this project.

Cougaar's extensive infrastructure also provides more features that are useful in application deployments. Its ability to recover from failures at the local agent level is particularly useful. AAA has support for replicated *Facilitators*,

which eradicates a single point of failure in an application. This however comes at the cost of state replication across *Facilitators*, which may not scale to large applications. At the cost of increased communications overhead, AAA also provides a rich ACL-based communications mechanism, which aims to simplify the implementation of complex communication schemes.

Aglets provide a lightweight support infrastructure, but add the ability for agents to move from node to node in an Aglets application. This is a potentially useful feature in a range of applications, and one that is not as well supported by Cougaar and the AAA.

5.2 The Evaluation Process

The following summarizes the strengths and weaknesses of the evaluation process that was used:

Cost: The project consumed 700 hours of effort in total. 130 hours were spent on project design, management and writing up the results. The remaining 570 hours were divided almost evenly between the engineers working on each technology to design, build and test the solution. Hence it took approximately 190 hours of effort to perform each investigation.

Effectiveness: The cost of learning the different technologies inevitably consumes a significant part of the budgets of evaluation projects such as this. With limited budgets, this makes it extremely difficult to carry out as thorough an investigation as pure research curiosity would dictate. However, taking a practical approach and gaining development experience across all three technologies creates a basis for rational comparison of the architectures and features. The development effort forces engineers to obtain a deep understanding of the features, and hence a higher quality evaluation is possible.

Measuring performance also provides insights into the quality of a technology. Performance is a key requirement in many modern distributed agent applications, and the test results give some concrete indications on the relative performance that can be obtained with each technology. Performance measures can also focus analysis on how a technology is likely to scale. Potential bottlenecks, poorly designed or implemented infrastructure components and non-scalable architectures quickly become apparent when a test application is available for performance tuning.

In summary, evaluation projects such as this one are typically a balancing act between what is desirable versus what is achievable with the budget available. In this project, the difficulty of learning some of the agent toolkits meant we slightly exceeded budget, mainly because of exploring implementation alternatives.

6 Related Work

As agent technology matures, it is being applied to produce solutions to real problems in many application domains. However, agent based systems have been typically implemented with ad-hoc solutions (communication languages, protocols, etc.) to meet the problem domain requirements. In response to this situation, standards are being created and implemented, such as Foundation of Intelligent Physical Agents (FIPA) [9]. As noted in [10], only recently have the FIPA standards been applied to practical applications. This cited work presents an analysis and evaluation of certain aspects of the current FIPA standards on the basis of the experience gained in developing a video entertainment broadcasting system. The authors analyzed the pros and cons of using FIPA standards for building MAS for such systems.

Performance and evaluation papers have recently started to appear in the literature. Many papers have been written on evaluating certain techniques, learning paradigms, scheduling mechanisms, and other software components that are important to the development of a MAS. NASA Goddard Space Flight Center has started to develop an Agent Concepts Testbed (ACT). ACT [11] is an environment in which rich agent and agent-community concepts can be evaluated through detailed prototypes and scenarios.

In several papers, authors have started to evaluate overall agent architectures, not just sub-components. Yamamoto et al. [12] report on a performance evaluation of a MAS based on large numbers of agents that process jobs by interacting with each other in an electronic commerce environment. The authors focus on evaluation of controlling the memory and CPU usage in a MAS containing thousands of agents. Performance was evaluated based on measuring throughput for two tests (single-server system and two-server system). They noted that by scheduling the activities of agents in an appropriate way, the throughput of agent interactions can be kept to a constant value in response to an increase in the number of consumer agents.

A recent study [13] reports on a direct comparison between the performances of platforms in mobile agent systems. The authors present results of a benchmarking study comparing eight Java-based mobile agent systems (Aglets, Concordia, Voyager, Odyssey, Jumping Beans, Grasshopper, Swarm, and JAMES). The study provides some information about the performance and robustness of each platform. The experiment involved a cluster of six machines connected through a 10Mb/sec switched Ethernet. A simple benchmark application was used that was composed of a migratory agent that roamed the network to obtain a report about the current memory usage of each machine. The authors were investigating three main metrics, namely application performance; robustness; and network traffic. A summary of their findings is presented in [13].

[14] compares a FIPA-compliant agent technology with a non-agent J2EE component based architecture. It describes a comparison framework that reveals considerable differences in terms of performance, deployment support and reliability between the two technologies, and makes recommendations about when to select either technology.

7 Conclusions and Further Work

This paper has presented the results of evaluating three different agent technologies. The evaluation process has been effective in highlighting some of the strengths and weaknesses of each technology. As in any technology evaluation and selection process, there are few, if any, absolutes. The most important outcome is that the capabilities of a particular technology match the foreseeable application requirements.

We are now working on exploiting the outcomes of this project and extending its findings. This includes providing an effective means for disseminating the findings of this study and creating an on-going vehicle for evaluating additional agent technologies that researchers and developers may be interested in. Like [15], we believe that more widespread use of rigorous quantitative evaluation will help more rapidly mature the spectrum of agent technologies. To this end, further evaluation studies incorporating different agent technologies, larger scale test implementations and heterogeneous distributed test-beds are desirable.

References

- [1] N.R.Jennings, K.Sycara, M.J.Wooldridge, A roadmap of agent research and development, *Autonomous Agents and Multi-Agent Systems*, 1, pp 275-306, Kluwer, 1998
- [2] M.L.Griss, G.Pour, Accelerating development with agent components, *IEEE Computer*, May 2001, pp 37-43
- [3] I.Gorton, A.Liu, P.Brebner, Rigorous Evaluation of COTS Middleware Technology, *in IEEE Computer*, vol. 36, no. 3, pages 50-55, March 2003
- [4] A.Liu, I. Gorton, Accelerating COTS Middleware Technology Acquisition: the i-MATE Process, *in IEEE Software*, pages 72-79, volume 20, no. 2, March/April 2003
- [5] Kumar, et al, The Adaptive Agent Architecture: Achieving Fault-Tolerance Using Persistent Broker Teams. International Conference on Multi-Agent Systems, Boston, MA. (2000).
- [6] D.Lange and M.Oshima, Programming and Deploying Java Mobile Agents with Aglets, Addison Wesley, 1998
- [7] <http://www.cougaar.org/introduction/overview.html>
- [8] Kumar, et al, Semantics of agent communication languages for group interaction. National Conference on Artificial Intelligence, Austin, Texas, AAAI Press. (2000).
- [9] FIPA (2002) <http://www.fipa.org>

- [10] Charlton, P., Cattoni, R., Potrick, A., and Mamdani, E. Evaluating the FIPA Standards and Its Role in Achieving Cooperation in Multi-agent Systems, Proc. 33rd International Conference on System Sciences, (2000)
- [11] Truszkowski, W., and Rouff, C. (2002) New Agent Architecture for Evaluation in Goddard's Agent Concepts Testbed, <http://agents.gsfc.nasa.gov/papers/pdf/aap41.pdf>
- [12] Yamamoto, G., and Nakamura, Y. (1999) Architecture and Performance Evaluation of a Massive Multi-Agent System, Conf. Autonomous Agents, pp. 319 - 325.
- [13] Silva, L.M., Soares, G., Martins, P., Batista, V., and Santos, L. The Performance of Mobile Agent Platforms, First International Symposium on Agent Systems and Applications Third International Symposium on Mobile Agents October 3-6, 1999, Palm Springs, California
- [14] M. Casagni, M. Lyell., Comparison of Two Component Frameworks: The FIPA-Compliant Multi-Agent System and the Web-Centric J2EE Platform, in Proc. 25th International Conference on Software Engineering, Portland, USA, pages 341-350, May 2003
- [15] S.E. Sim, S. Easterbrook, R.C. Holt, Using Benchmarks to Advance Research: A Challenge to Software Engineering, in Proc. 25th International Conference on Software Engineering, Portland, USA, pages 74-83, May 2003