AFRL-IF-RS-TR-2003-306
Final Technical Report
December 2003

BATTLE MANAGEMENT ALGORITHMS FOR AUTONOMOUS UNMANNED SYSTEMS (BMAAUS)

CACI Technologies, Inc.

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK

STINFO FINAL REPORT

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2003-306 has been reviewed and is approved for publication.

APPROVED: /s/
KENNETH LITTLEJOHN
Project Engineer

FOR THE DIRECTOR: /s/
JAMES A. COLLINS, Acting Chief
Information Technology Division
Information Directorate

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 074-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	DECEMBER 2003	**FINAL** **Aug 01 – Dec 02**

4. TITLE AND SUBTITLE
BATTLE MANAGEMENT ALGORITHMS FOR AUTONOMOUS UNMANNED SYSTEMS (BMAAUS)

5. FUNDING NUMBERS
C - F30602-00-D-0221 Task 0008
PE - 62702F
PR - 558T
TA - QF
WU - 09

6. AUTHOR(S)

Jennifer Seitzer

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

CACI Technologies, Inc.
1300 Floyd Avenue
Rome NY 13440

University of Dayton
Department of Computer Science
300 College Park
Dayton OH 45469-2160

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL/IFT
525 Brooks Road
Rome, NY 13441-4505

10. SPONSORING / MONITORING AGENCY REPORT NUMBER

AFRL-IF-RS-TR-2003-306

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Kenneth Littlejohn/IFTA/(937) 255-6548, X3587 Kenneth.Littlejohn@rl.af.mil

12a. DISTRIBUTION / AVAILABILITY STATEMENT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

12b. DISTRIBUTION CODE

13. ABSTRACT ***(Maximum 200 Words)***

Autonomous agents are self-directed, independent entities that interact with an environment by in-taking percepts through sensing devices and by acting on the environment through effectors. This work centers on autonomous entities in an adversarial environment that operate with conflicting goals, process noisy data, adapt in real-time to a dynamic environment, and collaborate to achieve one or more collective goals. In this proposed work, the domain of application is robotic soccer. Ultimately, we expect the research to apply to work in Unmanned Air Vehicles (UAV). The work performed in this project relates to the implementation of autonomous agents.

14. SUBJECT TERMS
Autonomous Agents, Reinforcement Learning, Layered Learning, Communication and Collaboration Protocols, Multiagent Planning, Territoriality, Real-Time

15. NUMBER OF PAGES
27

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

TABLE OF CONTENTS

1. Introduction

The work we performed in this project relates to the implementation of autonomous agents. The first part (duration of 9 months) implemented agents in the domain of robotic soccer. The second part augmented an existing Beowulf cluster at the Wright Patterson Air Force Base. Ultimately, we expect both aspects of this research and implementations to apply to work in Unmanned Air Vehicles (UAV).

In this twelve month period of investigation, we have performed an ongoing literature search, downloaded and experimented with several versions of the Robocup Soccer Server along with several existing clients on several platforms, successfully performed superficial modifications to several clients, written two original soccer playing clients in both C++ and Java, and augmented a Beowulf cluster with four additional Pentium machines.

2. Background

Autonomous agents are self-directed, independent entities that interact with an environment by in-taking *percepts* through sensing devices and by acting on the environment through *effectors*. This work centers on autonomous entities in an adversarial environment that operate with conflicting goals, process noisy data, adapt in real-time to a dynamic environment, and collaborate to achieve one or more collective goals. The agent work accomplished in this funding period resided in the domain of robotic soccer. In future work, we expect the research to apply to work in Unmanned Air Vehicles (UAV). The scope of the overall project encompasses four phases: Phase I:

Beowulf Construction, Phase II: Autonomous Agent Implementation, Phase III: Endowing Intelligence, and Phase IV: Application to UAV's. The work accomplished here focused on Phase I and Phase II where we augmented a Beowulf cluster and implemented autonomous entities using a multi-agent computing system to run on that cluster.

2.1. Robotic Soccer

In Peter Stone's *Layered Learning in Multi-Agent Systems*, the author delineates many techniques and principles for the construction of computing systems housing multiple agents. Using the problem domain of robotic soccer, he provides mechanisms for the coordination of independent agents' behaviors where "behavior" is defined as a mapping from perceptions to actions (over time). Robotic soccer is the programming and building of robots equipped to participate in competitive soccer tournaments. This global endeavor is embodied in the pursuit known as RoboCup. RoboCup research is quite apposite to many significant problems in both military and industrial applications. The work of robotic soccer embodies the formalization and implementation of a system of multiple collaborating agents operating in a real-time, noisy, and adversarial environment. The RoboCup organization provides a software platform for research on the software aspects of RoboCup.

2.2. Agent Programs and Agent Architectures

In [Russell and Norvig 1995], agents are defined as an agent program plus its architecture. The agent program is the "brains" of the agent housing decision-making

and reasoning capabilities. It is here that, given a sequence of agent percepts, the *next agent action* is determined. The architecture is responsible for receiving and transforming percepts into a form recognizable by the agent program, and for transferring the agent program's determined action to the agent's effectors. Thus, the main impetus of software research in autonomous agent work lies in agent program development.

Agent program development entails development of planning, collaboration, and navigation algorithms. The agent architecture is manifested either by a physical robot or a software simulator. Usually, development and experimentation takes place on a simulator and then successful programs are transferred to the physical agent (i.e., the robot). The simulators are typically implemented using a client-server architecture by housing the agent program in the client, and the agent architecture in the server.

In this work, we used the server provided by the RoboCup organization (see http://sserver.sourceforge.net) to serve as our agent architecture and thus, to test our algorithms. Both client/agent programs we wrote housed a different original algorithm.

3. The Original Agents and the Robocup Environment

The Robocup environment is one in which the soccer server is standardized and provided for all participants. Any agent is composed of an agent architecture (those parts that react and act on the environment) and the agent program (the decision making module that chooses the agent's actions to take). In the Robocup paradigm, the agent architecture is

housed in the soccer server and the agent program constitutes the client. The clients, therefore, embody the agent's intelligence and skill.

In the funded period, we composed two original soccer-playing agent clients: the JAG client and the Biter-derived client. The JAG client was written in C++ and performed far better than the Biter-derived client which was written in Java.

3.1. The JAG Client (C++ Client)

The JAG client is an operational agent program which successfully plays soccer in the Robocup environment. After the initial, primitive implementation, several strategic issues needed to be addressed in order to increase the ability of the JAG client. Many of these have been implemented and are currently working. Others are still being studied and refined.

> Two major goals that we wanted to accomplish to refine the JAG agent involved developing communication between clients and developing the agent to behave as a goalie.

3.2. The Communication Model

Our main goal with the communication model was to develop a local positioning model between the agents. This position model ensures that if two (or more) teammates are going after the ball, they won't bump into each other. The following procedure is employed:

1) The agents commence by continuously sending messages to the server whenever they get within twenty units (or less) of the ball. These messages contain the agent's team

name, the agent's player number, and the current distance the agent is from the ball. Sending a message does not count as an action, so multiple messages may be sent per time cycle.

2) Once a message is sent to the server, the other agents can receive the message to get relevant information. Received messages look like this: (hear 59 2 "JAG 2 6.7"), where 59 is the time, 2 is the player who sent the message, and the quoted string is the actual message that was sent. As stated above, our messages contain the team name, the player who sent the message, and the player's distance from the ball.
3) Once an agent receives a message from another player, the agent compares its distance from the ball with the other player's distance from the ball. If the agents are both within a radius of 10 units from the ball, the agent closest to the ball will go for the ball, while the other agent's desire to go towards the ball will be suppressed. In the manner, we never have two or more agents from the same team crowded around the ball.

There is one problem with this model, which the global positioning model will take care of once it is fully developed. The problem is that the local positioning model only makes sure that agents are not crowded around the ball. It does not take care of the fact that other agents who are away from the ball may be crowded around each other. However, once the global positioning model is developed, the agents will be spread out into overlapping zones, where they will be restricted to cover their zone area. The local positioning model will then come into play when the ball falls into an area where zones overlap.

3.3. Development of the GOALIE

In order to get the server to recognize a player as being the goalie, a special initialization command must be sent to the server. To do so, the user starts up our client with a –g tag to tell us that they want the first player to be a goalie. The command line appears thusly:

$ ./soccer –g

Once the client is initialized as a goalie, it simply runs the goalie code and suppresses the regular agent code. It is important realize that the goalie is part of the same program as the other players. In keeping with the rules posed by the Robocup organization, a separate program was not developed for the goalie. Rather, the goalie runs a separate piece of code (the goalie code) within the agent. Moreover, the goalie also uses code that is relevant to all agents, such as the communication model.

Our future work in refinement of the goalie, is to force it to play positionally. That is, to let the goalie only play in the goalie area. In the next section, we discuss our current attempts to developing a positional model for all players.

3.4. Development of the Positional Model

Our first attempt at a positional model included using a series of flags to form a positional boundary to which the player was confined. In order to implement this, many flags were needed for comparison. This, in addition to unreliable flag distance and direction values sent from the server, caused the functions governing the boundaries to be very lengthy and the boundaries to be inaccurate and unreliable. This forced us to find an alternate method to achieve our positional model.

Our next attempt at attaining a positional model was to use the absolute position of a player on the field. The absolute position of a player is a set of x and y coordinates that defines exactly where the player is located on the soccer field. The absolute position is achieved by comparing the player's distance to a line to find an x or a y coordinate using the equation below:

$$abs(LINEDISTANCE * \sin(LINEDIRECTION * \frac{\pi}{180}))$$

This equation will find either the x or the y coordinate depending on which lines we are looking at, vertical or horizontal. To find the other coordinate, we used the closest known flag to the player. Based on this we developed three equations to handle the three different locations of the flags: outside the field, inside the field, and on the boundary line. If the flag is on the outside of the field we use this equation:

$$\sqrt{FLAGDISTANCE^2 - (LINESTANCE + 5)^2}$$

If the flag is on the field line, we use the following equation:

$$\sqrt{FLAGDISTANCE^2 - (LINESTANCE)^2}$$

We are still working on implementing the equation for the condition that the closest flag to the player is inside the field. This, along with defining the exact x and y player boundaries, is part of our future work. As soon as we get the absolute x and y

coordinates of the player finalized, we can use upper and lower boundaries limit to restrain the player from moving any further. Once we have this positional model finalized, we will work on a team dynamic positional model in which the team moves as a whole up or down, left or right, of the field and still maintains their positions.

4. Beowulf Cluster Work

The goal of Phase I of the overall research project was for a Beowulf cluster executing parallel models, demonstrating and assessing military effectiveness, to be built. The parallel system will ultimately be endowed with intelligence in the form of *dynamic intelligent modules* that are periodically exchanged between autonomous entities in a peer-to-peer fashion. This mechanism will, among other things, help realize the goal of effective autonomous operation. Additionally, in order to provide an information-centric platform and interface, a *publish and subscribe* information exchange facility will be designed and implemented. In the final three months of the funding period, we augmented the Beowulf cluster with four Intel architecture machines.

4.1. Cluster Computing

Cluster computing enables us to build a scalable multiprocessing computing system using a network of possibly heterogeneous computers. A Beowulf cluster is a collection of possibly heterogeneous COTS processors interconnected by a local area network using a high speed switch and running coordinating software to emulate the operation of a large high performance parallel machine. The main objective of using a Beowulf is to provide a large amount of CPU processing power with minimal expense. Moreover, the

construction of a Beowulf often allows us to absorb hardware into a functional capacity by creating a larger, more powerful computational machine. Some examples of coordinating software that perform the parallel machine emulation over the network are *Parallel Virtual Machine (PVM)* and *Message Passing Interface (MPI*).

4.2. Augmentation of the Cluster

The augmentation of the cluster entailed loading operating systems, cluster software, and benchmark test software. We delineate the steps we took to load the cluster software and testing software as follows.

4.3. Software Installation

To upgrade and install MPI software, we selected the LAM-MPI package because it was initially used on the cluster and because its distribution as source code makes it easier to install on multiple architectures. The first step in the upgrade of the established cluster was the upgrade of the MPI software from version 6.5.4 to 6.5.6. While installing that software, we also added a link to shared install path on Blackbox so all machines have the same LAMHOME path.

The original MPI software is still on Blackbox under /usr/export/debian/usr/local/lam-mpi-old. The new software has two different versions for the two architectures in the cluster and only the appropriate version is mounted on each of the worker nodes via NFS. The actual install paths are all on Blackbox, but each computer mounts the platform specific MPI package in /usr/lam-mpi. On Blackbox these packages are in /usr/export/debian/usr/lam-mpi for the Sun architecture and /usr/export/debian-

x86/usr/lam-mpi for the x86's. Blackbox has a soft link to the x86 version in its local /usr/lam-mpi.

Once the software was installed, the systems needed to be configured to be aware of the LAM-MPI's binary files. This was done in both the .profile and .bashrc files in the /home/sserver directory. This replication was necessary due the bash shell's nonstandard remote login procedure; when bash does a non-interactive remote shell, it only loads the .bashrc file. All that was necessary in the .bashrc and the .profile files was the inclusion of /usr/lam-mpi/bin in the PATH variable's list. This list is then exported and the MPI binaries are visible without needing absolute paths.

At this point the installation was tested with the original group of Sun machines. It performed a lamboot successfully, which creates the daemon on the remote machines to allow MPI processes to run. This procedure works even though Blackbox's PATH variable is going through a link and is not an absolute path.

Once the systems would initialize the MPI daemon, we attempted to run the LAM test suite. This attempt failed and it caused a cascade failure that took down much of the network. This was our first indication of the Ethernet adaptor problem that is discussed in-depth elsewhere.

Because of the previous failure, the next step was to make sure any program would run successfully on the cluster. This was done by compiling the example programs that are included in the LAM-MPI package. The first program that was attempted was a simple program called ring. In order for this, or any MPI program, to work on the cluster, two versions must be compiled and copied into a directory that is part of the PATH variable. For these test programs, we standardized on the /usr/local/bin directory on both the Sun

machines and the x86's. Once a Sun version was compiled and added to the /usr/export/debian/usr/local/bin directory on Blackbox, the Sun machine's /usr/local/bin directory, the test was performed and ran successfully.

Further testing showed that the basic test programs work with no problems, but programs that transfer large amounts of data will kill the network. This was first discovered using the Mandelbrot test program that failed in the same manner as the LAM test suite.

At this point the new x86 nodes were added into the cluster and the successful tests were completed without complication, but the unsuccessful tests would still disconnect the Sun machines from the network while causing no harm to the x86 nodes. The unsuccessful tests were then performed on only the x86 machines without the Suns; in this configuration the tests completed with no problems including the Mandelbrot program and the LAM test suite.

Location of Important files

/usr/export/debian/usr/lam-mpi	Sun MPI Install Directory on Blackbox
/usr/export/debian-x86/usr/lam-mpi	x86 MPI Install Directory on Blackbox
/usr/lam-mpi	Local MPI Install Directory on all Nodes
/home/sserver/.bashrc	Location of PATH export command

Adding a New User

Copy .bashrc and .profile from /home/sserver to /home/new_user

Copy lam-bhost.def, lamSun-bhost.def, and lamX86-bhost.def from /home/sserver to /home/new_user

Configuration Files

lam-bhost.def	Standard LAM-MPI boot definition, includes all nodes
lamSun-bhost.def	Original LAM-MPI boot definition, only uses Suns
lamX86-bhost.def	New LAM-MPI boot definition, only uses X86s

Running a MPI Program

In order to successfully run a MPI program on this cluster, the program must be compiled twice and put into a directory that is part of the PATH variable. Before any compilation or execution can occur, the cluster must be invoked. This can be done in a variety of methods. Depending on the complexity of the data sent during the execution of the program, it may or may not run on the Sun machines. To run the cluster excluding the Suns, invoke MPI using the lamX86-bhost.def file in /home/sserver. To boot normally use the default lam-bhost.def file. To use either file as the boot definition, simply type lamboot –d /home/sserver/lam-bhost.def at the command prompt. This will create a lamd daemon on the node machines ready to accept remote access.

Once the system has been initialized, the program can be compiled. It must be compiled for both the Sun and the x86 architecture. To compile the program for the Suns, rsh into one of the nodes b1-b9 and run the Makefile or mpicc *.c as appropriate. Copy the resulting binary file into /usr/local/bin while still logged into the Sun machine. Exit from this remote shell and repeat the compile on Blackbox. The result of this compilation needs to be copied into /usr/export/debian-x86/usr/local/bin in order for the x86 nodes to be able to run successfully. Once it is copied successfully the original executable can be invoked with the command mpirun N appname. This will cause the program to run across

all available nodes. If execution is to be limited to a subset of nodes, mpirun n0-n4 appname can be used instead. This command will execute the binary file on nodes 0-4 and ignore any additional nodes.

Once we were sure that the Sun machines were the problem and that the problem was likely a hardware limitation, we proceeded to begin benchmarking the system. Our primary benchmarking tool was the PovRay program used to test the initial cluster. The other MPI aware test program HPC Games, was limited in its testing capability and focused primarily on the performance of Blackbox. Although its focus did not help judge the system, there was one interesting result from one test it performed that may explain results gathered from PovRay.

The first test that was performed was a simple comparison between the render speeds of different configurations of nodes. The results of this test are roughly what where expected: overall the system performed at its best with all nodes as part of the system. In individual tests, the 9 Sun nodes and the four x86 nodes came out to be roughly similar in processing power. We also attempted a weighted configuration; this attempted to force MPI into considering the x86 nodes to be roughly twice as powerful as the Sun nodes. This produced almost identical results to the basic configuration. We suspect this is because MPI is already performing load balancing and the additional configuration is not necessary. Figure 1 shows the four configurations and how long each configuration took to render the image. Figure 2 shows the average processing power of the overall cluster in the four configurations.

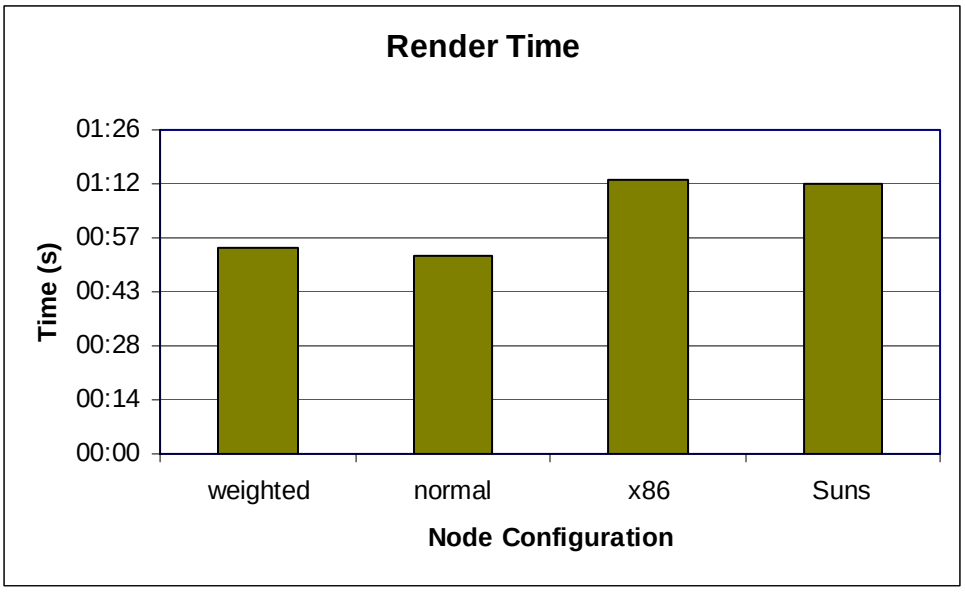

Figure 1 (shorter bars are better)

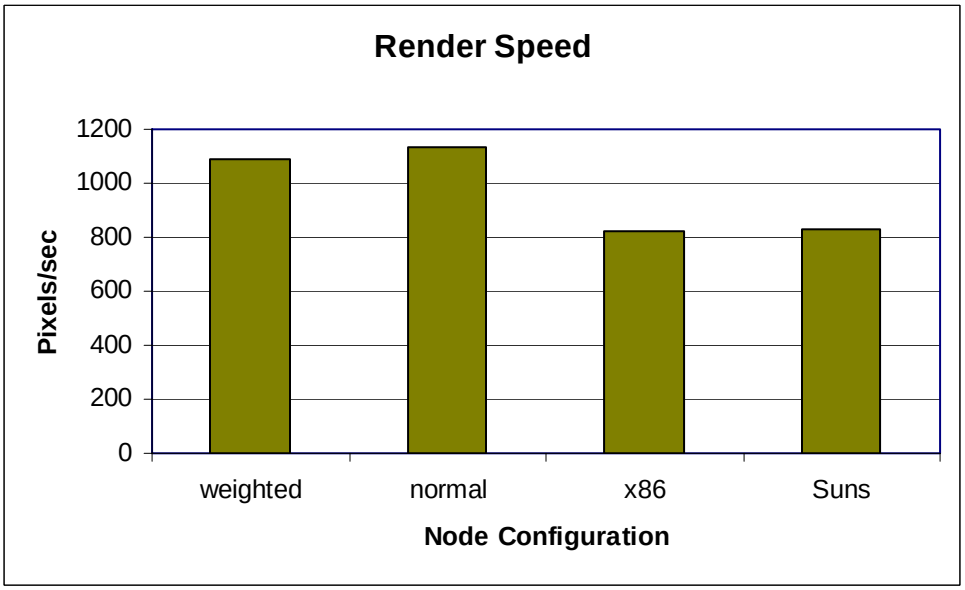

Figure 2 (longer bars are better)

The next set of tests that was performed on the cluster was designed to test how different sized processing problems affected the cluster's performance. The results were a little surprising and later tests using HPC Games showed a potential culprit. Each test performed was on the same render image displayed in Figure 3 at four different

resolutions. Each resolution was run on the three typical boot schemas, the weighted schema was removed due to minimal difference between it and the standard setup. The results showing overall processing time are shown in Figure 4, the pixels per second results are shown if Figure 5.

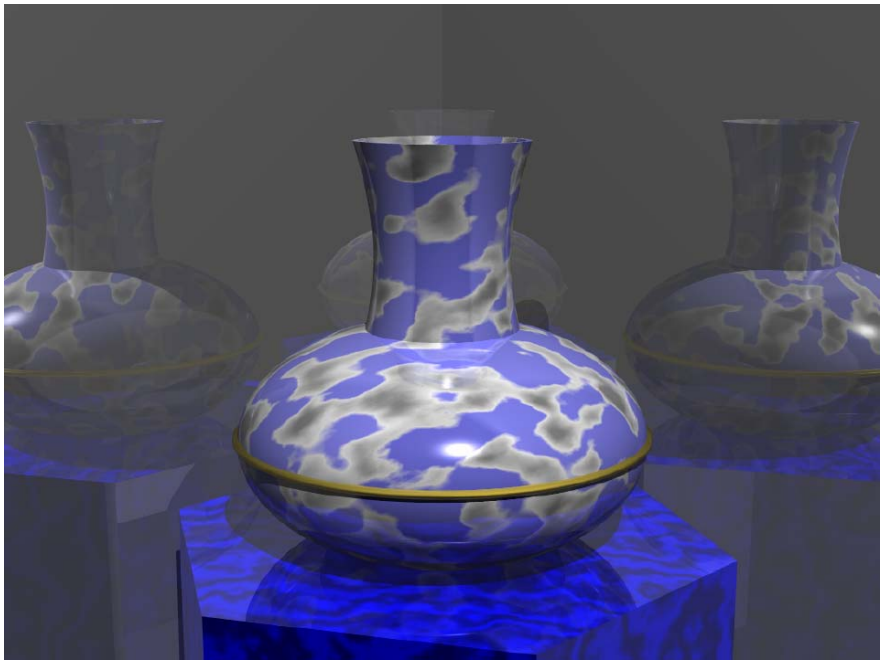

Figure 3 Test Image

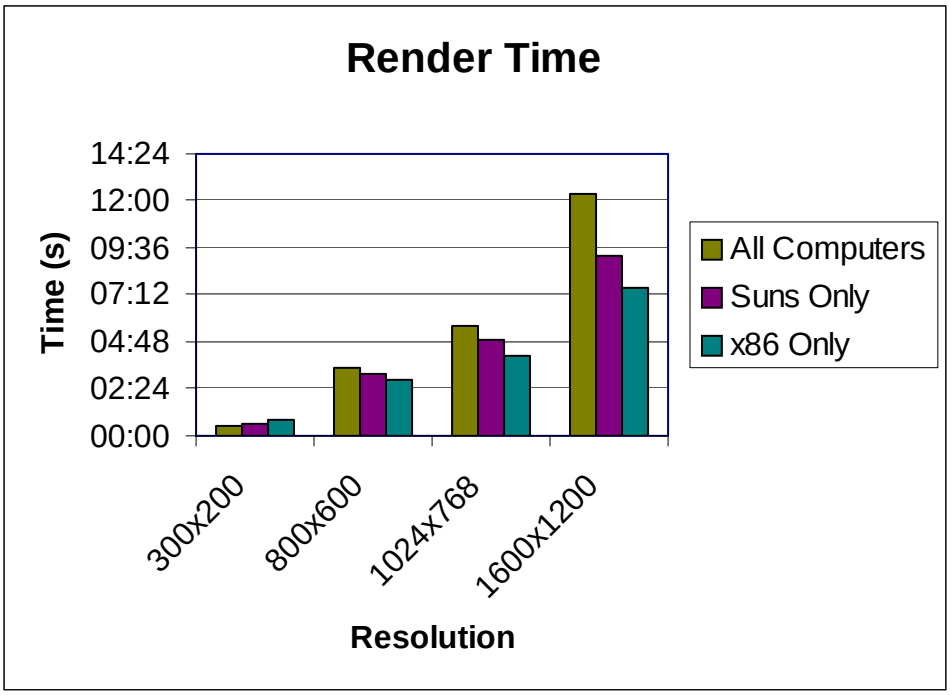

Figure 4 (Shorter Bars are Better)

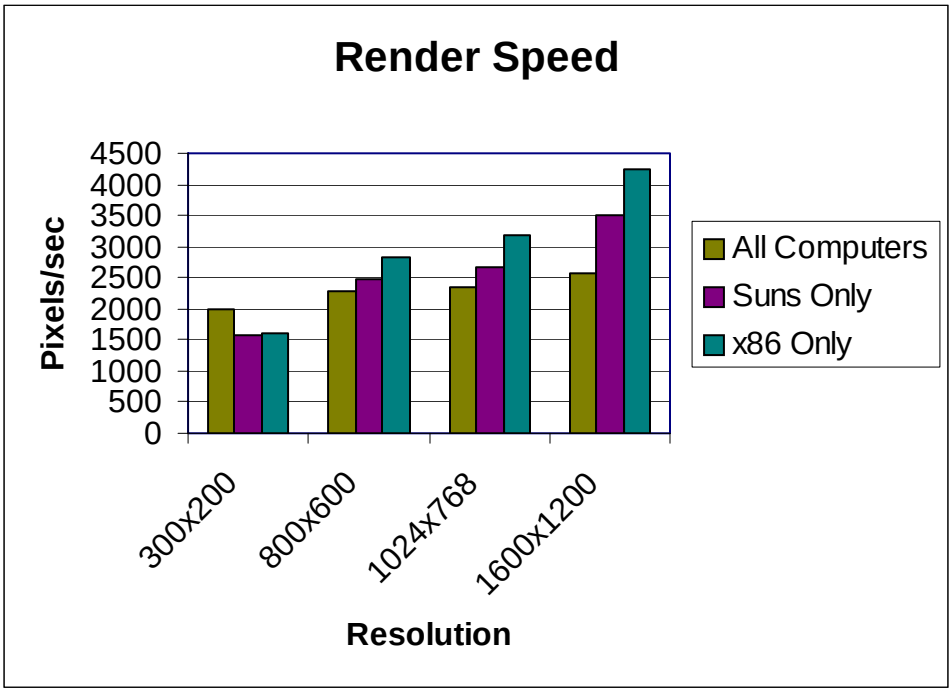

Figure 5 (Longer Bars Are Better)

The unusual part of these results is that as the render gets more complicated, the combination of all of the machines runs slower then either just the x86's or just the Sun's. We theorized that this is another manifestation of the network problem discussed earlier. This is because of the results of one of the HPC Games benchmarks, which performed a stress test of the network and determined the maximum bandwidth at varying sized blocks of data. If this test is performed with just the Suns or just the x86's, the typical maximum network transmission speed is around 5MB/sec. If all of the nodes participate in the test, the transmitting speed plummets to between .5MB/sec and .1MB/sec. This would greatly influence any test that extensively uses the network. This is clearly visible in Figure 6. This graph shows the percent difference between the test performed on all machines and the test performed on just the x86's. As the size of the render increases, the performance gap also widens. This is also visible in Figure 5, after the 800x600 render, the pixels/sec measurement stays roughly the same while at the same time the homogenous boot schema's continue to increase.

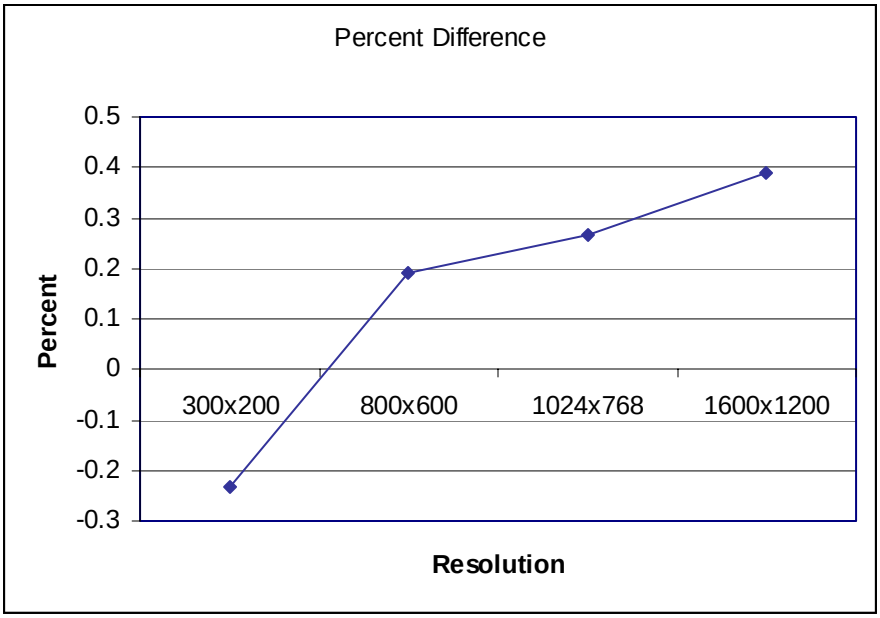

Figure 6

When we measured the time spent on the render on each of the 13 nodes, we discovered that the system was balancing load reasonably well. The x86 nodes were performing roughly twice the calculations compared to the Sun machines and the Sun machines listed at the end of the boot schema were receiving less work then the others. This data is plotted in Figure 7. This indicates that the load balancing system is working correctly and is not the cause of the poor performance of the overall network. This conclusion is further collaborated by the fact that as the render gets more complex, the percentages are not affected and should produce a linear increase in speed. Since it does not, the problem is probably in a different part of the system.

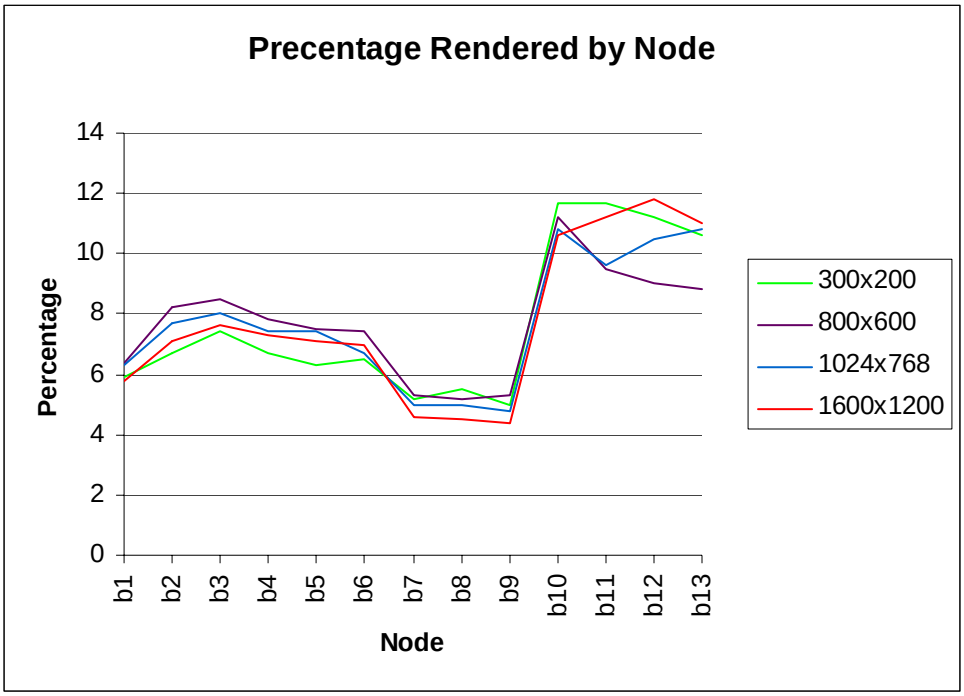

Figure 7

The final set of tests performed on the cluster was mapping how the addition of nodes affects the total render time. To do this we tested clusters with 2 x86 nodes, 3 x86 nodes, and finally all four x86 nodes. The results of these calculations are in Figure 8. As each

node was added, the time it took to render the same set of images dropped, but not quite linearly. Each node added a percent of its processing power but as more nodes were added, that percentage dropped. As more nodes are added, the effect of each additional node will shrink until such time as its addition will have a negligible impact on performance.

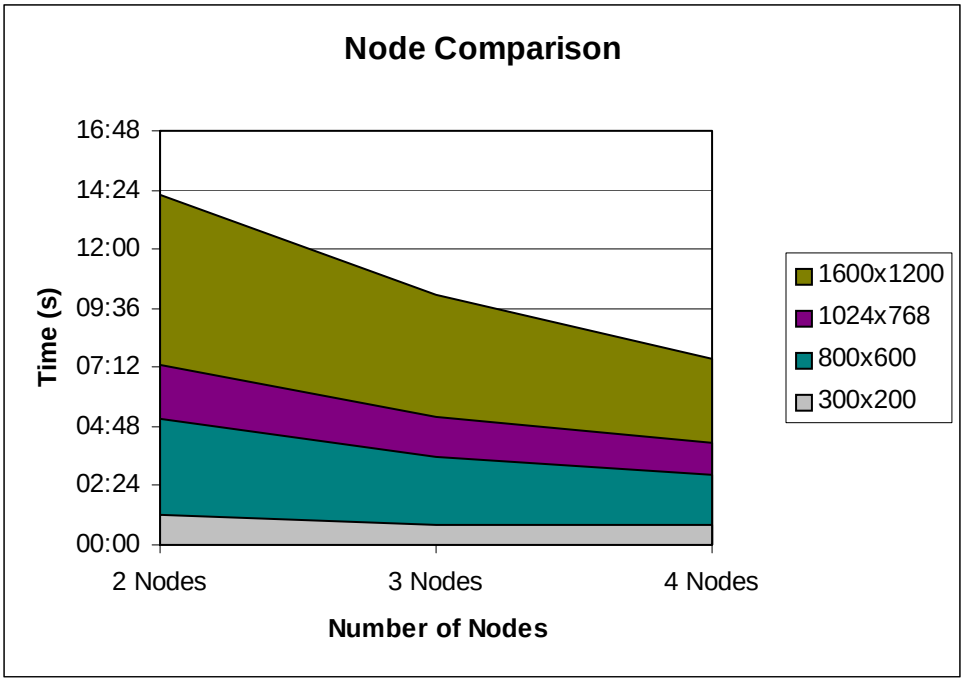

Figure 8

4.4. Recommendations on Cluster Upgrades

A less positive note about the operational status of the Beowulf can be found in a hardware limitation imposed by the Sun workstations. The current network card utilized in each machine, known as the SunLance NIC, has a hardware limitation in the size of its internal buffer. Once there is an incoming or outgoing datagram that is too large, or too much data in either direction, the buffer on the network card fills with error data. This

error data causes the network card to cycle through the buffer until all the error data has been removed.

The process of cycling through the buffer requires the network card to be reset each time an error data is read from the buffer. This constant cycling causes the system to be removed from the network until the network card becomes stabilized again.

This problem only occurs on large data sizes. Small data sizes can fit through the network card buffer with no problems. The actual size that causes a problem to manifest itself is unknown at this time due to inadequate testing of the network card.

Also, this problem has been found to exist on x86 hardware in network cards built from the “Tulip” chipset. Any network card that uses the “Tulip” driver for Linux will have similar problems under the same circumstances for the same reasons.

Recommendations:

If the current hardware configuration is to be maintained, a proper testing of the network card is in order. Otherwise full utilization of the Beowulf cluster will fail to exist. The point of break should be determined and documented so that future programming for the cluster can be written with less of a hassle.

Another method of overcoming the hardware limitation of the device would be to replace the network cards in the Sun workstations. The current SunLance NIC is a half-duplex 10Mbit network card, which can be defined as well below slow in comparison to some of yesterday’s network technology. Replacing the network card with a full-duplex 100Mbit

network card would improve the overall bandwidth of the Beowulf cluster as well as overcome the hardware limitation of the current network card.

The easiest and best way to overcome the x86 problem is to replace the network card with another that does not use the "Tulip" driver. Network cards for the x86 architecture are relatively cheap and easy to obtain.

5. Conclusion

This year-long project was to study, develop, and implement autonomous entities on a distributed cluster of workstations. We hope this work will eventually be applied to the area of unmanned air vehicles. The work involves a four-phase endeavor spanning five years of effort, work, and support.

In year one, described in this report, the Beowulf cluster was successfully augmented and two original autonomous agent clients were implemented.

6. Bibliography

Official Robocupp website: http://www.robocup.org.

RoboCup Simulation page: http://sserver.sourceforge.net

Russel, Stuart and Norwig, Peter. *Artificial Intelligence: A Modern Approach,* Prentice-Hall, 1995.

Stone, Peter. *Layered Learning in Multiagent Systems A Winning Approach to Robotic Soccer,* MIT Press, 2000.

7. Appendix

User Instructions for Running the JAG System

RoboCup C++ Client Documentation for Team JAG
Developers: Greg Buzzard, Jeff Wassil, Anne Niehaus

1) Starting the server and monitor together on BlackBox

-- Open a "root" shell, by clicking on the shell icon on the toolbar
at the bottom of the screen, and having someone log in as root

-- Type in the following commands at the prompt ($), omitting the $

```
$ cd /usr
$ ./StartSoccerAll
```

-- The server & monitor will now be running on BlackBox. Leave the window open and proceed.

2) Starting the C++ JAG team clients

The following directions will distribute clients on nodes b1 through b4 of the Beowulf

FOR NODE b1:

-- Open a "root" shell, by clicking on the shell icon on the toolbar
at the bottom of the screen, and having someone log in as root

-- Type in the following commands at the prompt ($), omitting the $

```
$ rlogin b1
$ cd /usr/local/sserver/client_soccer/scripts
$ ./StartUpG1 blackbox
```

-- Leave window open and proceed

FOR NODES b2 - b4:

-- Open a "root" shell, by clicking on the shell icon on the toolbar
at the bottom of the screen, and having someone log in as root

-- Type in the following commands at the prompt ($), omitting the $

```
$ rlogin b2 (where b2 is the current node you are working with)
```

$ cd /usr/local/sserver/client_soccer/scripts
$ ./StartUp3 blackbox

-- Leave window open and proceed

3) Starting Opponents

-- Open a "root" shell, by clicking on the shell icon on the toolbar at the bottom of the screen, and having someone log in as root

-- Type in the following commands at the prompt ($), omitting the $

$ cd /usr/export/debian/usr/local/sserver/client_soccer/Respina2001Bin
$ ./start3

-- We think this is the correct directory where the Respina team is located, but we are not entirely sure. You may have to search for this directory.

4) Killing monitor, server, and clients (Must complete this step to re-run clients)

KILL MONITOR:

-- Kill the monitor (GUI of soccer field) by clicking the "Quit" button on the actual GUI

KILL SERVER:

-- Go to the command prompt window that you used in Step 1.
-- Hit the "CTRL + C" combination
-- At the prompt, type:

$ ./StopServer

TO KILL JAG C++ CLIENTS:

FOR EACH CLIENT COMMAND PROMPT WINDOW (i.e. Command prompt windows for b1 - b4):
-- Hit the "CTRL + C" combination
-- At the prompt, type:

$ ./StopAllSoccer

When you are finally done running clients, close down all windows (you can leave them open if you want to run them again.)