October 2004

Autonomic Systems

Christopher Lee Kopec
Worcester Polytechnic Institute

Eric S. Leshay
Worcester Polytechnic Institute

James Dominic Baldassari
Worcester Polytechnic Institute

Project Number: DXF-GS04

Autonomic Systems

A Major Qualifying Project Report:

submitted to the Faculty of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

____

James D. Baldassari

____

Christopher L. Kopec

____

Eric S. Leshay

October 15, 2004

Approval:

____

David Finkel, Advisor

This page intentionally left blank.

Abstract

An autonomic system is defined as self-configuring, self-optimizing, self-healing, and self-protecting. We implemented the Autonomic Cluster Management System (ACMS), a low overhead Java application designed to manage and load balance a cluster, while working at NASA GSFC. The ACMS is a mobile multi-agent system in which each agent is designed to fulfill a specific role. The agents collaborate and coordinate their activities in order to achieve system management goals. The ACMS is scalable and extensible to facilitate future development.

Acknowledgements

We wish to express our utmost appreciation to the following individuals and agencies:

Our NASA mentor, Walter F. Truszkowski, for his constant support, guidance, and encouragement throughout the project.

Our advisor, Professor David Finkel, for his consistently valuable suggestions and sound advice.

Professors Fred Looft, Steven Bitar, and John Blandino for their constructive feedback.

NASA for extending the opportunity to complete our MQP at GSFC.

The GSFC information technology staff for allowing us to convert our workstations to Linux and for providing us with networking equipment for our system tests.

Patricia Rago and Dragana Kostic from IBM Autonomic Systems for their helpful comments about our project.

Table of Tables

Table of Figures

Table of Acronyms

ACMS	Autonomic Cluster Management System
API	Application Programming Interface
DMZ	Demilitarized Zone
GSFC	Goddard Space Flight Center
GUI	Graphical User Interface
HPC	High Performance Computing
IBM	International Business Machines
IP	Internet Protocol
IT	Information Technology
Jar	Java Archive
JVM	Java Virtual Machine
MPI	Message Passing Interface
MPP	Massively Parallel Processing
NASA	National Aeronautics and Space Administration
OO	Object Oriented
RoD	Resources on Demand
SDK	Software Development Kit
SMP	Symmetric Multiprocessing
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UML	Unified Modeling Language
VO	Virtual Organization
W3C	World Wide Web Consortium

Executive Summary

The NASA Goddard Space Flight Center (GSFC) in Greenbelt, Maryland supports the gathering and dissemination of knowledge about the Earth, solar system, and Universe. GSFC's primary responsibility is the development and operation of unmanned scientific spacecraft. The successful completion of GSFC's objectives often entails the solution of large computational problems, such as the modeling and simulation of complex systems. Many of these problems are so computationally demanding that some form of *High Performance Computing* (HPC) is required to solve them.

Traditionally, *Massively Parallel Processing* (MPP) computer systems have been used to meet HPC requirements. These systems may contain hundreds or thousands of processors within a single computer system. MPP computers are usually very expensive and difficult to upgrade, but they perform extremely well and are relatively simple to manage.

A recent trend in HPC has been to overcome the cost and scalability issues associated with MPP systems by using a different type of HPC system called a *cluster*. A cluster is a collection of inexpensive individual computers, referred to as *nodes*, that are connected via a network and configured to appear as a single computer to its users. Increasing the computational capability of a cluster is as simple as adding nodes to the system, resulting in a highly scalable HPC solution. The largest disadvantage of using a cluster is the complexity of its management and configuration. Instead of administering a single computer, as with an MPP system, management and configuration tasks on a cluster must be performed on every node. In a cluster comprised of hundreds or thousands of nodes management becomes a daunting task. Manually configuring thousands of nodes is inefficient, if not impossible. While an operating system may be able to optimize its own processes, it is not aware of the cluster as a whole and cannot coordinate its activities with the other nodes. A severed network connection or an otherwise unresponsive node could cripple the cluster if it is not able to recover from failures. Finally, unauthorized access to the cluster is a constant concern for system administrators.

Autonomic computing is a relatively new approach to managing complex systems that can potentially solve many of the problems inherent in cluster management. The definition of an *autonomic system* is one that is self-configuring, self-optimizing, self-healing, and self-protecting. Using these autonomic properties as a guide, we designed and implemented an Autonomic Cluster Management System (ACMS).

The ACMS is a mobile agent system composed of a number of agent processes communicating across a network of nodes. Each agent is written to perform a particular task, and together the community of agents collaborates to achieve a common goal. The common goal in the ACMS is to manage a cluster.

The ACMS is middleware that was written in Java and executes within the Java Virtual Machine (JVM). The JVM is platform independent and runs on top of an existing operating system. The ACMS can therefore be used on heterogeneous clusters, regardless of the operating system or underlying system architecture of each node. The ACMS maintains a record of all nodes in the cluster, as well as comprehensive system information about each node. The ability to maintain this record and handle the addition and deletion of nodes is what satisfies the self-configuring autonomic.

Distributed applications running on the cluster are optimized by the ACMS. It accomplishes this by examining each node's record. When the ACMS finds a node it deems available, the ACMS assigns new distributed applications to that node first. The ACMS uses the load balancing algorithm to satisfy the self-optimizing property of autonomic systems.

The myriad of agents that compose the ACMS require a method for communication. The communication system used by ACMS is TCP/IP encrypted by Secure Sockets Layer (SSL). Encrypted communication prevents rogue agents from joining the system and satisfies the self-protecting property of autonomic systems.

The ACMS was built to be robust and handle the occasional fault or failure. Each agent runs in its own JVM, so the failure of one agent will not affect other agents on the same node. The ACMS contains redundant agents; these agents assume the role of their partner if it should fail. The ability to handle failure and create new agents satisfies the self-healing property of an autonomic system.

Object Oriented (OO) architecture with encapsulation and inheritance was used in the design of the ACMS. For example, code for the four agents is stored in the *agent* package. The four agents are the *ConfigurationAgent*, *OptimizationAgent*, *GeneralAgent*, and *UserAgent*. The *ConfigurationAgent* is responsible for maintaining the location of all agents and acts as a central contact point for the other agents in the ACMS. The *OptimizationAgent*'s role is to decide how best to utilize the resources of the cluster. The *GeneralAgent* is present on every node in the system and acts as an interface between the agents and the nodes. The *UserAgent* is the user interface for the ACMS. While the ACMS is running there are two *GeneralAgents* on each node, two *ConfigurationAgents* per cluster, one *OptimizationAgent* per cluster, and any number of *UserAgents*.

To evaluate the performance of the ACMS we devised a series of four tests. The results of these tests reveal the overhead associated with using the ACMS to manage the execution of a distributed application. The distributed application we used during the performance evaluation was a simple prime number calculator designed to find all prime numbers less than one million. The first two tests were run with a single node, while the last two tests were run with five nodes. The first trial in each set was with the distributed application alone. During the second trial in each set we used the ACMS to manage the distributed application. The results show that using the ACMS incurs only a small amount of overhead. In the worst case scenario, which was when the ACMS and the distributed application were both running on one node, the ACMS caused less than 5% overhead. When additional nodes are introduced into the cluster, the ACMS distributes itself among the nodes so that the performance impact on each node is reduced. When the size of the cluster was increased from one to five nodes, the overhead associated with the ACMS decreased to less than 0.75%. The results also show that when the size of the cluster increased from one to five nodes the performance of the distributed application increased by 458%, which is significant because of its proximity to the maximum theoretical increase of 500%.

During our time at Goddard we accomplished all of our project goals. We designed and implemented an Autonomic Cluster Management System that incurs less than 5% overhead. We were able to use the ACMS to demonstrate each of the four autonomic system properties. GSFC can use the ACMS to educate NASA scientists and engineers about autonomic systems and their many potential applications, or use our code as a foundation for other advanced management systems.

1. Introduction

The National Aeronautics and Space Administration (NASA) has many sites throughout the United States. Each site has different objectives with respect to research, space, and flight. The Goddard Space Flight Center (GSFC) in Greenbelt, Maryland supports scientists and engineers learning and sharing their knowledge of the Earth, solar system, and Universe. The primary objectives at GSFC are developing and operating unmanned scientific spacecraft. Managed here are many of NASA's Earth Observation, Astronomy, and Space Physics missions [1].

Scientists and engineers at GSFC often require significant computational power in the research and development of advanced technologies. In the past, this capability was typically provided to researchers through the use of a single, powerful computer called a supercomputer. Supercomputers are usually highly customized, have hundreds or thousands of processors, and are extremely expensive.

A more recent trend in high performance computing has been to utilize a variation of a supercomputer called a cluster. Clusters consist of tens, hundreds, or even thousands of individual computers that are interconnected to provide the functionality and computational capabilities equivalent to a traditional supercomputer. Clusters are often less expensive than supercomputers because they can be constructed with widely available components and require comparatively little customization. System scalability is another advantage clusters have over supercomputers. While it is possible to upgrade a supercomputer, it is often a complex process involving the replacement of many internal components. In a cluster, however, simply adding computers to the system can increase computational power.

With the considerable power and scalability of a cluster comes an increase in the complexity of deploying and maintaining the system. Instead of configuring and performing administration tasks on a single computer, this process must be repeated on hundreds or thousands of computers. Special tools and software packages are required on large clusters to effectively maintain their operation. However, existing cluster management solutions are often not dynamic enough to react to changes in large clusters. For example, if a single computer in the cluster is responsible for delegating tasks to the rest of the cluster, and that computer becomes unresponsive, the entire cluster would be rendered ineffective. Considerations such as this must be made when designing cluster management systems.

A possible solution to the management problem has been proposed by International Business Machines (IBM). IBM has begun an initiative to develop self-managing systems so that the need for human intervention in complex systems will be reduced. Autonomic computing is defined by IBM as “An approach to self-managed computing systems with a minimum of human interference. The term derives from the body’s autonomic nervous system which controls key functions without conscious awareness or involvement" [2]. Specifically, autonomic systems must be self-configuring, self-optimizing, self-healing, and self-protecting. These properties of autonomic systems allow them to adapt to changing operating environments and system demands, and to be fault-tolerant. All are essential abilities for an effective management system.

GSFC has recently begun joint collaborative work with IBM, researching and designing systems with autonomic capabilities, and has determined that an autonomic approach to cluster management would be a practical application of autonomic computing. To demonstrate the advantages of autonomic computing we designed and implemented an Autonomic Cluster Management System (ACMS). Conforming to the principles of autonomic computing, the system configures itself, optimizes its resource usage, repairs components of the system that have stopped responding, and protects itself from unauthorized access. The cluster management system we developed is able to demonstrate the capabilities of an autonomic system using realistic operational scenarios. As the system became more functional, we were able to speak with several employees of IBM that were interested in the research being done on Autonomic Systems. We submitted a whitepaper to Patricia Rago and Dragana Kostic of IBM, which can be viewed in Appendix I.

We conducted performance evaluations on the ACMS after its completion to determine its functionality, effectiveness, overhead. The maximum overhead caused by the use of the ACMS remained under 5% throughout all tests, and we found that it seemed to scale well and was quite robust. We created a user guide for the ACMS (See Appendix E), as well as documentation for implementing distributed applications that can be managed by the ACMS (See Appendix D). Throughout the development of the ACMS we documented all of our code to facilitate future development (See Appendix J). The ACMS could be used in the future by GSFC as the foundation for a more advanced system to manage its large clusters or to further explore the role of autonomic systems in distributed computing environments.

2. Background

This chapter presents fundamental information about our project. We begin the chapter with an overview of the properties and topologies of distributed systems. Next, we provide an introduction to cluster and grid systems. Following the discussion of distributed computing, we provide information on the various network protocols that are commonly used in distributed systems. We then discuss agent-based systems and their role in distributed systems. The chapter concludes with a definition of autonomic computing and the properties that all autonomic systems must possess.

2.1. Distributed Systems

A distributed system is defined as any system whose components can be executed concurrently on discrete computer systems [22]. The discrete computer systems that comprise the distributed system are referred to as nodes. There are several requirements to consider when designing an application for a distributed system. Basic design decisions include the division and distribution of data and code, the communication protocols, multithreading requirements, and system security.

An objective in designing a distributed system is to decide how to partition the system's components into modules that can be executed independently on different nodes. The division of components into modules is usually done in one of two ways. The first partitioning method is to maximize the use of data that is local to each node, thereby minimizing transfers over the network. This method is most often used when there is a restriction imposed on the network traffic or when the time required transferring all necessary data is a significant percentage of the total time required to complete the task. The alternate approach is to encapsulate any necessary data and code into a module, which is then transferred over the network to the node. The second approach is traditionally used in compute-intensive tasks, in which the node spends the majority of its time executing code.

The design of the communication protocols depends largely on the method used to partition the system's components. If all of the data is available locally to each node, we can limit communication to only messages between the nodes, and not data. However, if data and code is to be transferred between nodes it is necessary to have robust protocols that are capable of exchanging abstract data types or even objects themselves. These protocols must be able to guarantee the integrity of all transferred data. Regardless of how these protocols are designed, it is important for them to be extensible and flexible to accommodate new functionality or modifications in the future.

Multithreading is often implemented in order for the system to achieve its maximum efficiency, reliability, and availability. Using multiple threads of control simultaneously allows the system to optimize its resource usage and leverage the full potential of each node. Multithreading also allows the system to have asynchronous properties. Asynchronous communication in particular is important because delays in response time caused by a heavy load on one node will not deteriorate the system as a whole. Data can be transferred and responses handled at the maximum rate for each node, even if these rates are different among nodes.

Security is an important consideration when a distributed system is accessible from an untrusted network or has a wide user base. Any sensitive data that needs to be transmitted over the network should be protected. Encryption is a common method for securing sensitive data as it is sent between nodes. An additional security measure is to authenticate each transaction so that the system knows a particular node or application is trusted prior to transmitting sensitive data or accepting commands [13].

2.2. Decentralized Systems

The distributed computing environment consists of different topology designs. The Internet in the past has consisted of many centralized systems; however, the decentralization of the Internet has begun through peer-to-peer programs. There are four common architectures used in the design of a distributed system; the four topologies are: centralized, decentralized, hierarchical and ring. Each of these topologies can be displayed on its own or combined with others to form hybrid patterns. Distributed systems can be described with seven properties says Nelson Minar; these are listed as "manageability, information coherence, extensibility, fault tolerance, security, resistance to lawsuits and politics, and scalability" [4]. With each property he can then rate the system showing how a specific design will perform. Observing each of the patterns, the centralized and decentralized topologies both have attributes closely related to the functionality displayed in our autonomic system design. The major contrast between these two topologies is the orientation that nodes are connected to one another.

2.2.1. Centralized Topology

The centralized topology has been the most common form used in applications. The design is typically seen as a client/server pattern used by applications such as databases, web servers and other simple distributed systems. A centralized system functions by having one server node with many clients each connecting directly to the server. Each client connects to the server to send and receive information. The server would be the location which would dispatch application requests and store any types of databases containing information.

Using the evaluation scheme designed by Nelson Minar [4], it can be seen that the primary advantage of a centralized system is with its simplicity. All data is located in a central node; this allows for the system to be easily managed and the consistency of data correct and coherent. Centralized systems can be easily secured; only one host needs to be protected. The major feature allowing for simple manageability also causes a downfall in the design, which is that all the data is in one location. The system is not fault tolerant; if the main node fails the whole system is no longer functional.

The system cannot be easily extended; information can only be added to the central system. The limitations of the system are placed on the capability of the central server, proving that centralized systems are almost impossible to scale.

2.2.2. Decentralized Topology

A decentralized system topology functions with almost completely opposite characteristics of a centralized topology. In this design all peers communicate symmetrically and have equal roles in the system. This type of system is most commonly seen in peer file sharing programs [4].

Unlike with a centralized system, decentralized systems are normally very difficult to maintain and data within the system is never fully authoritative. Since every node may contain data regarding the system and information being processed, a decentralized system becomes very hard to keep secure. However, decentralized systems are very easy to extend. New nodes can easily join the network and start communication. The fault tolerance of a decentralized system is much greater, there is no central point of failure; the failure or shutdown of one node in the system does not affect any other nodes in the system.

Scalability of a decentralized system in theory becomes more capable as more hosts join the network. Although in practice, algorithms that keep decentralized systems coherent often carry a lot of overhead.

2.2.3. Centralized + Decentralized Topology

Combining these two topologies forms a hybrid centralized and decentralized design. This topology architecture is built of centralized systems embedded in decentralized systems. This type of system can be seen in how mail clients have a centralized relationship with a specific mail server, but the mail servers share email in a decentralized manner [4].

The hybrid system is able to enjoy the advantages of both the centralized and decentralized designs. Decentralized systems contribute to the extensibility, fault-tolerance and possible large scaling of the system. The partial centralization makes the system more coherent than a purely decentralized system. There are fewer hosts that are holding authoritative information. Faults of this system are that manageability is still as difficult as with a decentralized system and the system is no more secure than previously. A large advantage with this system is how easily it scales. The design can easily handle hundreds of millions of users, as seen with Internet email.

2.3. Beowulf Clusters

A Beowulf cluster is a network of computers configured to behave as a single supercomputer [5]. The computers that compose a cluster are referred to as nodes. The power of a cluster is influenced by two factors, the number of nodes and the speed of those nodes. A Beowulf cluster is a specific network topology; a cluster's nodes are isolated from the outside world and work collaboratively to solve large tasks by sharing resources. The operating system used on a cluster is typically Linux and is configured to maximize throughput rather than responsiveness. The first Beowulf cluster was built at GSFC in 1994 [5]. The clusters at GSFC have been upgraded as new technology became available and are still in use today solving computationally complex problems for scientific research.

2.4. Grid Computing

A "grid" is a type of parallel and distributed system that enables the sharing, selection, and aggregation of geographically distributed "autonomous" resources dynamically at runtime depending on their availability, capability, performance, cost, and users' quality-of-service requirements" [11]. Grid Computing focuses toward Resources on-Demand (RoD), which is the practice of allocating resources across a distributed network transparently, with as little delay as possible [10]. Grid Computing solves issues related to latency, security, and packet failure that are not typically addressed in local area network cluster designs. Henri Casanova states that research is still required in Grid Computing, more specifically the "dissemination of and access to data and information" [2]. Research in Grid Computing has brought about the consensus among scientists that a set of standards must be created in the form of an Application Programming Interface (API) and a Software Development Kit (SDK). Using these tools scientists should be able to create Virtual Organizations (VO) that are analogous to the

Beowulf clusters discussed previously, except that its members are located on a WAN instead of a LAN [2].

2.5. Network Protocols

There are several ways to implement peer-to-peer communication across a network. The Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP) are the most common network protocols used by applications [6]. The specific use of these protocols depends on the type of communication that an application needs. TCP is a reliable and robust protocol that is often used in peer-to-peer network connections. When it is necessary to send a message to multiple hosts simultaneously, UDP is frequently used because of its broadcasting and multicasting capabilities.

2.5.1. TCP

TCP is a connection-oriented, reliable protocol for data transmission. TCP guarantees that all packets will arrive at their destination in sequence and error free. A TCP connection is comprised of three phases: connection establishment, data transfer, and connection termination.

In the connection establishment phase, a client will send a request for a connection to a server listening on a known address and port. The server will then reply that the connection request has been acknowledged. The final step in this phase involves the client sending an acknowledgment back to the server that the connection has been established. The three steps in the connection establishment phase are traditionally referred to as the three-way handshake.

TCP uses three primary mechanisms for ensuring reliability during the data transfer phase. A sequence number attached to each packet is used to order the packets correctly and to detect any duplicate data transmission. A checksum is used to detect any errors in the packet that may have occurred during transmission. The checksum is a number that is generated based on the data in the packet; if any bit in the packet changes, the checksum will be different. The checksum is generated just before the packet is sent, then appended to the packet. When the packet is received, the checksum is generated again and compared to the checksum that was sent with the packet. If they do not match, an error has occurred and the packet will be resent. In addition to errors at the bit level, occasionally an entire packet may be lost. Acknowledgments and timers are used to detect data loss during the transfer. An acknowledgment is sent for each packet that is successfully received. If this acknowledgment is not received within a defined time period, the sender assumes it was lost during transmission and retransmits that packet.

The connection termination phase is similar to the connection establishment phase. The client will send a connection termination request to the server. The server acknowledges the request, and then sends its own termination request to the client. The client acknowledges the request, and the connection has been terminated. The result of TCP's many safeguards and checks is a reliable, error-free data exchange. However, this reliability comes at the price of considerable overhead compared to less reliable protocols such as UDP [6].

2.5.2. SSL

Secure Sockets Layer (SSL) is a protocol for encrypting communications over a TCP connection. It utilizes the concept of public key cryptography. This type of authentication uses both a public and a private key, known collectively as a key pair. The public key is made available to anyone who wants to communicate securely. The private key, however, is never distributed. Data that is encrypted with the private key can only be decrypted with the public key. Conversely, data that is encrypted with the public key can only be decrypted with the private key. When a secure transmission needs to occur the sender, which we will call *A*, will first make its public key available to the receiver, *B*. *A* then encrypts the data using its private key and sends the encrypted data to *B*. *B* can then decrypt the data by using *A*'s public key. If *B* needs to send data back to *A*, it can encrypt the data with *A*'s public key, then send it to *A*. *A* will be able to decrypt the data with its own private key [8, 9].

2.5.3. UDP

UDP is a connection-less, best-effort protocol for data transmission. It does not guarantee error-free transmission or that packets will arrive in the correct order, as TCP does. Additionally, there is currently no high level encryption layer for transmissions sent over UDP. Although UDP is not as robust and reliable as TCP, UDP has much less overhead.

Another advantage of using UDP is its ability to send broadcasts and multicasts. A broadcast is a transmission that is sent to all nodes on a network with a single transmission. A multicast is a subset of a broadcast. While a broadcast will be sent to all nodes on the network, a multicast is only transmitted to a smaller group of nodes within the network, called a multicast group [7].

2.6. Multiple Agent Systems

Research into Multi-Agent Systems has been conducted alongside research in the areas of cluster and grid computing. Multi-Agent Systems are distributed applications where a group of entities called agents interact to reach a common goal. An agent is an application running on a computer, programmed to accomplish a single goal. In an ideal Multi-Agent System the agents have different roles that define the agent providing a goal and objectives; therefore the dissemination of information is decentralized so that every agent only requires a sub-section of the total information regarding the system. This forces the agents to work in a collaborative manner and trust the other agents in the system to accomplish a given task. Multi-Agent Systems are flexible and robust because they do not rely on any single structural crutch. If a failure does occur in the system, the unaffected agents can continue to communicate and conduct work uninterrupted, although at a diminished capacity [13].

2.7. Mobile-Agent Systems

Multi-Agent Systems provided the groundwork for the development of Mobile Agent Systems. Mobile Agent Systems are a type of Multi-Agent System where the agents are not fixed to a particular location. The agents have the ability to move from one to location to another, both physically and logically. Mobile Agent Systems such as grasshopper [23] allow an agent to move from one node to another by traveling across TCP/IP in a serialized state. The agent can then be instantiated on the remote machine and its threads resumed. A logical movement in location is where agents are able to move from one logical grouping of agents to another. If agents can move from one multi-agent system to another then communication and collaboration can be gained from multiple systems each containing its own community of agents. A constraint of moving agents around in the system is that all agents must have a common language; the language of consensus in the mobile agent community is that of Java [13].

2.8. Autonomic Computing

The word autonomic is defined as "acting or occurring involuntary; automatic: an autonomic reflex" [14]. Autonomic computing is derived from an aspect of the human body; the nervous system. The human body regulates itself and reacts without conscious effort often; this process often being referred to as a reflex. The concept behind autonomic computing is to build a nervous system for a computer that can recognize everyday Information Technology (IT) events and react with the appropriate reflex response [2].

The four core autonomic properties an autonomic system exhibits are self-configuring, self-healing, self-optimizing, and self-protecting. IBM is currently coordinating a large effort to further research on Autonomic Systems around the world [14]. Goddard is one of many scientific institutions to join with IBM in this endeavor. IBM has defined autonomic systems by outlining eight elements that make up an autonomic system:

1. An autonomic system must be aware of where it is located and what resources are available to it.
2. An autonomic system must be able to adapt to changes in its environment; reconfiguring itself and its resources, as necessary.
3. An autonomic system is never satisfied with its performance; it is constantly searching for optimizations in order to improve its performance.

4. An autonomic system must be able to recover from failure or extraordinary events.
5. An autonomic system must be able to defend from outside attacks.
6. Autonomic systems must be able to adapt to their environment, but also have the ability to change the environment in order to adapt to the system.
7. The implementation of an autonomic system must be open and support a heterogeneous set of platforms.
8. Autonomic systems need to accomplish their tasks while making the details of their operations transparent to the user.

This list of elements acts as a blueprint for the design and implementation of autonomic systems [14].

An autonomic system built along IBM's blueprint must also contain a MAPE loop. Another development concept from IBM, MAPE stands for Monitor, Analyze, Plan and Execute. This is how an autonomic system must behave. First monitor the situation and gather data, second analyze the data for trends, third generate a plan of action, and fourth execute the plan. Central to the four steps is knowledge. Knowledge is collected while the system executes and is used to influence decisions in later MAPE loops. In this manner the system gains knowledge during each loop and is able to learn from its past experience. This allows the autonomic system to adapt and evolve over time [15].

IBM developed several systems incorporating autonomic properties commercially available today. One application is Tivoli. Tivoli is a suite of applications targeted at bringing autonomy to IT management in enterprise business. Each individual application in the suite incorporates one or more autonomic properties. This is where the most current developments in autonomic systems are present today. None of the applications fulfill IBM's definition for an ideal autonomic system; however each application is a step closer to building that ideal system [16].

IBM has broken down the development of autonomic systems into five distinct levels. The levels are in increasing complexity and flexibility: basic, managed, predictive, adaptive and autonomic. The basic level is where the aspects of a system are monitored and managed by an IT team full-time. The team is responsible for finding problems, researching solutions, and fixing the errors. The managed level refers to a system that can summarize its own environment. The system is able to provide the IT team with comprehensive performance information and the location of errors. At the third predicative level the system can recognize patterns informing IT managers a problem could occur, so they can act before the problem can manifest. The system also has the ability to provide the IT managers with a list of possible solutions to correct the problem. At the fourth adaptive level, the system not only suggests a solution but executes it as well. A system at the fifth and final level is autonomic. An autonomic system is capable of performing all the functions of the first four layers as well as being dynamic; the system must be dynamic and be able to evolve with a business and its goals [17].

A survey conducted by IBM in 2003 found that 40% of the market was at the basic level with only 1% reaching the autonomic level in 2002. IBM extrapolated upon this data and predicts that by 2006 only 19% of the market will be in the basic level with 5% using autonomic systems. IBM presents these levels as a five step plan for evolution of future software, with the distribution of true autonomic systems as the goal [18].

2.9. Summary

In order to fully understand our project and the need it addresses it is necessary to be familiar with several topics in computer science: high performance computing, distributed systems and applications, multi-agent systems, and autonomic computing. We first researched autonomic computing to understand how the self-configuring, self-optimizing, self-healing, and self-protecting properties were applied to existing systems. To learn about the application of autonomic principles we researched some of IBM's actual products that exhibit autonomic properties. We then began researching clusters and distributed computing as a potential application of autonomic computing principles. It is clear from our research that distributed systems, such as clusters, can present unique challenges in regard to management and maintenance. It was our opinion that autonomic principles could be an effective response to these challenges. Once we had decided on designing an autonomic management system for clusters we were able to define our project statement and goals.

3. Problem Statement

The goal of our project is to design a system displaying the autonomic properties in a multi-agent environment using distributed computing across the agents. Within the context of our overall goal, we developed several objectives.

3.1. Determine a programming language

Java was originally chosen as the language to use for this project to allow easy integration with two toolkits we considered using, Madkit and the Log and Trace Analyzer plug-in. Both of these applications have since been determined unnecessary for the project. Re-evaluating the language choice we looked into numerous languages narrowing down the spectrum to C++ and Java, which are two languages that well suit our needs. The system has certain requirements; handling and easily building an object oriented environment, the use of a well organized and useful development workspace, and easy integration to the department's CVS which maintains revisions of the code and allow simple integration. The possibility to run the application on multiple platforms with ease is another feature that would provide additional scalability and ease of use for future users; rather than having to recompile for each operating system since both Windows and Linux systems are being used during development.

3.2. Evaluate toolkits and applications

We evaluated a variety of toolkits and applications that dealt with different aspects of autonomic computing. Among the applications were a Log and trace analyzer plug-in, a multi-agent development environment called Madkit, and the Eclipse development workspace. We also looked into finding an application or plug-in for UML design that would analyze the class diagrams and produce code. Research was conducted to find the best method of developing an autonomic system of agents; and which applications would provide the most assistance during the development stages.

3.3. Define each of the four attributes

We researched into others companies that have begun research and development in the field of autonomic systems and how they defined the four attributes. The best examples we found came from IBM; they have begun integrating autonomic components into their servers. Using IBM as a model, we came up with our own set of definitions of how each attribute would act within the system, its functionality and how it would exemplify the specific characteristics of autonomic computing.

3.4. Design of the System

Once each attribute was well defined, as to how it relates within the system, we determined how the agents would act within the system. This included determining which attributes would be actual agents acting autonomously in the multi-agent environment and which attributes would act as properties on the entire system. Along with the agents we determined that a messaging system was needed to communicate across the network between agents.

3.5. Design of the Agents

Unified Modeling Language (UML) class diagrams were created to map out the structure of the agent system. Since each agent will have a particular functionality and relationship with the other agents it is necessary to determine each agent's functionality before beginning to program. The diagram also planned out the actions each agent will perform, helping to organize the system structure and helps to prevent major structural changes later in development.

3.6. Design of the Message system

UML class diagrams were also created to organize the messaging system and the different types of messages that could potentially be sent across the network to each of the agents. The system needed a way to send and receive messages between agents. There are a variety of methods to send messages; the ones taken into consideration were the TCP/IP protocol, multicasting to groups of subscribing agents and broadcasting messages to an entire subnet on the network.

3.7. Program each of the four attributes

We looked into different programming techniques and methods to help with the implementation of the four autonomic attributes. The iterative design process was observed as a process we could follow for creating and implementing subsystems in the packages. We also used the UML diagrams to determine the importance and order each subsystem would be completed and tested for functionality.

3.8. Design the agents to interact with each other

There are multiple means for communication between agents. TCP/IP provides a secure connection with error checking to ensure all information arrives at the destination correctly, also allowing for secure connections to be instantiated. However, with this additional functionality there is a great deal more overhead. Using TCP/IP, the IP address of the destination agent server must be known; however in some cases this cannot be determined. There are also two other protocols for sending to many agents if an IP address is unknown. Both broadcasting and multicasting messages are options. Both are flawed since no encryption can be used and both transmit over UDP. Broadcasting a message would go out to everyone on a particular subnet, whereas multicasting would be sent to a specific known address in which agents would need to subscribe. Agents that have subscribed to a particular multicast address would receive any messages sent to that address. The UDP protocol is used in both broadcasting and multicasting information. The UDP protocol is connectionless and therefore does not provide any assurance that a packet will arrive at the destination. UDP could also be used in place of the TCP/IP protocol when an address is known; this would be the case if faster communication is needed and packet loss is not a major concern.

3.9. Design the agents to act as a distributed system

The agents are programmed so that they can survive and function on their own. However they still communicate and relay information between each other. In this case the agents act similar to a distributed system, in that they perform actions on their own and communicate results to help improve the performance of the system as a whole.

3.10. Design the agents to manage a distributed computing environment

The agents' main goal is to manage the processes on a distributed system. This allows for computing of a program to become more efficient. The agents are designed to utilize the maximum processing power from the cluster of computers. The agents are able to load balance by moving processes running on computers from one system to another trying to improve overall performance.

3.11. Test the system

In order to show that the system works correctly test scenarios were written up to display to a user each of the attributes. The test scenarios are necessary to show communication between agents and allow others to see how the system works but also to find any bugs hidden in the system.

4. Methodology

There were several steps we followed to achieve the goals outlined in our Problem Statement (See Chapter 3). We first made basic design decisions about the system, such as the programming language and development environment that we would use. We then evaluated existing tools to determine if they would be useful in our project. Before design began on the system, we specifically defined each autonomic property and how would be applied to the system. We then began designing the system outlining a high level design of the system with UML diagrams. Once the system was documented, implementation of the design began. We strived to keep the system always at a functional state so that continuous testing could be done finding any unforeseen problems.

4.1. Choosing a Programming Language

The system we developed required a high level language, one that supported multi-threading, networking, and portability. Multi-threading is required in order for the agents to multi-task. Networking support is fundamental to our language choice because the agents must be able to communicate to each other. Portability is important because we want to support heterogeneous clusters. We quickly narrowed the possibilities to the two most popular Object Oriented (OO) languages, Java and C++. We carefully weighed the merits of each language against the goals of our project. C++ provided easy access to system information and a fast runtime. However, it was not platform independent and would require us to write system specific code for each possible architecture and operating system. Java, on the other hand, provided platform independence at the price of speed. Another benefit of Java over C++ was garbage collection and a well-documented Application Programming Interface (API). Additionally, the Java Virtual Machine (JVM) handles fatal errors gracefully. It allows us to restart the process without harm to the underlying system, which might not be possible if we used C++. With this research taken into consideration the language used for our application was Java. Java met more of the requirements set when determining a language for development.

4.2. Evaluating toolkits and applications

With Java chosen as the development language for this project we began looking into different tools and applications that would help in the development of the system. There were a variety of tools and applications that were useful in different aspects of the system. Among the applications were Madkit, a Log and Trace Analyzer, Eclipse, a CVS server, and Linux. Madkit is a development tool for multi-agent environments; the Log and Trace Analyzer is used to discover problems in a distributed web server; Eclipse is an IDE for Java; the CVS server allowed for group collaboration on code development; Linux was used as the operating system for development and deployment of the system.

4.2.1. Linux

Development began using the Windows platform; however problems arose while using Windows that forced a switch to Linux. During development the use of GUI's to display error messages and program information was used. This became very useful but also meant that many more resources were being used which lagged Windows. This problem did not arise when run on Linux and therefore development was moved off Windows and onto Linux platforms. Another issue found was the collection of system statistics. Java can only determine the JVM's available memory, along with other statistics regarding the JVM. Using Linux allowed easy access to the system statistics because this information is stored in files in the /proc directory. Information from the /proc files can be collected and parsed to gather the necessary information.

The other reason for converting to Linux was that Windows was unable to handle the necessary number of instances of Java we required. During testing, we found that after three to four instances of Java were opened the system began lagging and calculations, such as the timers, were lasting much longer than intended. This failure is inadequate for our system because upon startup the program will require five agents, and therefore five instances of the JVM, on one system. Further testing on Linux showed that the program performed as expected when run on Linux. Each instance was opened when expected and no lagging was noticed.

4.2.2. Eclipse

We chose Eclipse [24] as our development environment because it is very robust, and is open source software. Eclipse offers a wide variety of plug-ins that can be used if necessary. The Log Trace Analyzer is an example of an Eclipse plug-in; other types of plug-ins allow for development in other languages, such as C++. The application is also very easily navigable, organizing the packages and classes along with generating source code. Eclipse includes functions that can generate repetitive source code such as constructors and private variable getters and setters. Along with generating the source, it includes headers formatted for Javadoc and the ability to *refactor* the code if variables or names need to be changed; this increases productivity, allowing focus to be on the main system rather than creating and maintaining repetitive code. Eclipse also easily integrates with the CVS server used in our department.

4.2.3. CVS Server

Once programming began we realized that it would be very difficult for each of us to maintain our own code, and then try to integrate all of the code into one set of files. We began looking into different options such, as network shares or version control, which a CVS server provides. The use of the CVS server allows for each of us to code and then easily integrate the code back into the set of files maintained by the server. The server also maintains old versions of code that allow all previous changes to be viewed in the case that errors are inadvertently introduced. Eclipse allowed us to easily determine the code changes that resulted in errors [25].

4.2.4. Madkit

Madkit was originally suggested by Walter Truszkowski as an application to investigate, as it could be useful for the project. It is a "Java multi-agent platform that provides General Agent facilities (lifecycle management, message passing, and distribution) and allows high heterogeneity in agent architectures and communication languages" [26]. From the start it looked like we would use Madkit to develop the system. However after a more in-depth investigation of the software we found that it has a lot of downfalls. It seems that to run the source code, Madkit must be installed on the system. This would not be beneficial to the system we are developing if every node required Madkit; Madkit would need to be installed on every computer wanting to join the system and Madkit may also require the use of a graphical interface. Installing Madkit on every computer would require much more time and also make the development process more complicated. Using a graphical interface would mean that a larger portion of the memory would be used to run the system rather than devoted to the applications run by users of the system. Also we are trying to conserve processing power for the application being maintained by the agents, therefore using large amounts of processing power and memory to maintain an instance of Madkit on each system would produce negative results.

One other advantage we thought Madkit would provide was the ability to handle multiple agents in an environment; controlling their actions and communication between one another. However, it seems that the application does not allow for communication of agents over a network, only within one system.

Since Madkit is incapable of communicating over a network along with the extra baggage it needs to run the code on a system; we concluded that Madkit does not provide any necessary extra features that would save time during development. Coding the agents and messaging system without any application needed to manage them also provides us with much more flexibility in designing the objects to fit the system rather than designing to fit the Madkit application needed to run them.

4.2.5. Log and Trace Analyzer plug-in

The final application we investigated was a plug-in for Eclipse called the Log and Trace Analyzer [27]. This tool seemed to provide useful features toward developing an autonomic system, being that it maintained logs of anything that occurred on a system; then allowing for analysis of the logs to find flaws or problems that were occurring. This would allow for the problems to be fixed before fatal errors occurred which would crash the system.

The downfall with this tool is that it only worked for a web server. Since we are not limiting the agents to maintaining a web server the application was not useful. The features would have been useful, however it seemed the application could not be easily converted to work with any type of system that it was introduced, rather was very limited with what it maintained. Once this was realized, we decided to perform all the analysis on our own rather than depending on this or a similar application.

4.3. Applied Autonomic Attributes

The purpose of our Multi-Agent system was to demonstrate the four autonomic properties; we defined each attribute in the context of our project. The four autonomic attributes are self-healing, self-configuring, self-optimizing and self-healing (See Appendix C).

4.3.1. Self-Protecting

We defined the property of "self-protecting" as security; our system had to be self-protecting by using secure communication and preventing rogue agents from joining or monitoring the system. We decided to realize this goal by using Secure Sockets Layer (SSL) with RSA encryption.

4.3.2. Self-Healing

The property of "self-healing" was defined as resilience to failure. Our system had to incorporate the ability to handle the failure of an agent or node by spawning new agents to replace those that fail. This was designed as a fundamental principle in the agent framework, as every agent has the ability to spawn other agents. Although this is true, we recommend that only General Agents spawn new agents to eliminate any confusion regarding which node a new agent will be located.

4.3.3. Self-Configuring

The property of "self-configuring" was embodied in an object called the Configuration Agent. The Configuration Agent maintains the status of the system and knows where all nodes are located. He is responsible for ensuring the system is configured correctly and in an operational state.

4.3.4. Self-Optimizing

The property of "self-optimizing" was given an entire agent as well. The Optimization Agent uses the information gathered by the Configuration Agent to determine load management of applications and agents. One of its primary roles is to ensure that the two Configuration Agents are always on separate nodes, except in the case of a single node, so that if the node fails the system can recover.

4.4. High Level Design Overview

We designed the ACMS to be a multi-agent system, and as such is composed of a multitude of agents. While the ACMS system is built to support any number of agents, there are three types of agents that make-up its foundation. These agents are the backbone of the ACMS and must be present in the system at all times in fixed numbers. The first type of agent is the Configuration Agent. This is the agent responsible for maintenance of the ACMS and there must be at least two Configuration Agents active at all times. The second agent is the Optimization Agent whose role is to load balance the cluster; there is one and only one Optimization Agent active at any given time. The third and last agent is the General Agent. The General Agent is replicated throughout the system as more and more nodes are added. Its role is to keep track of a node's statistics and start and/or stop processes. There are always two General Agents on every node in the cluster. So the number of General Agents in the cluster is equal to the number of nodes multiplied by two.

The agents will communicate by passing messages back and forth. These messages can be contain text and data objects. It will travel over standard Ethernet from agent to agent. The messages will be encrypted using RSA encryption. The agents handle the messages by parsing the text contained in the message.

Another integral module of the system is the database. The database is where information on every node is stored such as system performance and location. The database was designed to hold approximately one hundred nodes, although it can hold any number with a drop in performance proportional to the number of additional nodes.

The ACMS is designed to run distributive processes on the cluster and allocate resources as necessary. In order to run these processes, which we will refer to as applications, requires the applications to meet specific criteria. These criteria define where the application and ACMS interact and at the implementation level is an interface. All applications run on the ACMS must implement this interface so that the agents can start, stop, pause, resume and move the applications while they are running.

For further development of agents in the ACMS a built-in debug tool is provided. There is a debug console which any agent can create and send output to be displayed. It comes complete with a text area and a button to terminate the agent, killing the virtual machine as well. This feature allows you to add print statements to your code for debugging purposes when a console is normally not available.

4.5. Agent Design

The system consists of three types of agents; each has functionality exemplifying autonomic system properties. The three agent types we designed are called General Agents, Optimization Agents, and Configuration Agents. The ACMS is comprised of two Configuration Agents and one Optimization Agent per system, and two General Agents per node. The agents' goal is to manage a distributed application while maximizing its performance by implementing load-balancing techniques on the system.

4.5.1. Configuration Agent

The purpose of the Configuration Agent is to make the system self-configuring. The functionality of the Configuration Agent consists of maintaining a current list of all the agents in the system and making this information available to other agents. When an agent first comes on-line it broadcasts to the Configuration Agent's multicast address stating that it has joined the system. When this message is received, the Configuration Agent examines the table to ensure that the new agent is needed. For example, if there are already two Configuration Agents in the system and a third comes on-line, the system might become unstable. If the new agent does not belong in the system, a termination message is sent back to the agent. The Configuration Agent cycles through the database of agents asking each if it is still functioning properly. If the Configuration Agent is incapable of establishing a connection with an agent, it can be assumed that the agent is no longer functioning correctly and will therefore be removed from the database. Otherwise, the agent responds with a list of information such as the address and port number of the agent's server, the agent type, and its system statistics (processor speed, number of processors, total memory, free memory, etc.). This list of information can be easily expanded to include requests for other information, if necessary in the future. When the Configuration Agent receives this information it is updated in the table.

The system contains both a primary and a secondary Configuration Agent to support redundancy and the self-healing autonomic property. The primary Configuration Agent will be referred to as the President Configuration Agent while the secondary Configuration Agent will be referred to as the vice President Configuration Agent. Ideally, both of the Configuration Agents would be on different nodes in the system so that if one node stops responding, there would be at least one Configuration Agent in the system. The reason for redundancy is that the database is stored locally by the agent in memory. Therefore, if the agent stopped functioning for any reason all the information within the database would be lost. To prevent this occurrence, the Vice President Configuration Agent synchronizes with the database of the President Configuration Agent. Only the President performs the system configuration tasks. However, if the President Agent were to stop functioning, the Vice President Agent would be able to continue the President's role. The Optimization Agent would detect that there is only one Configuration Agent functioning and recreate a second Configuration Agent.

4.5.2. Optimization Agent

The purpose of the Optimization Agent is to make the system self-optimizing.

The role of the Optimization Agent within the system is first to contact the Configuration Agent for a current copy of the database. Once received, the Optimization Agent begins analysis of the database to ensure that there are the correct number and types of agents in the system. It verifies that there are exactly two Configuration Agents in the system, one Optimization Agent in the system, and two General Agents on each node in the system. If it finds this information to be incorrect, it sends commands to create or kill one or more agents, stabilizing the system. After performing a brief analysis of the system, it then begins observing the loads and statistics of each node, noting the lightly and heavily loaded systems. When the application needs to start a new process the Optimization Agent searches for the first system that is not heavily loaded, it contacts a General Agent on the corresponding node and commands it to start the requested process. The Optimization Agent has the capability to move agents and processes from one node to another; allowing processing power to be utilized over multiple systems for a task, rather than having one system perform all of the processing.

No redundancy is built in to the Optimization Agent because it does not store any important information in memory. If the agent were to stop responding the Configuration Agent could easily recreate it. Once recreated, it would continue functioning properly with no loss of critical data. The only loss that occurs is any analysis of the table that the previous Optimization Agent had completed.

4.5.3. General Agent

The main functionality of each General Agent is to execute the commands of the other agent types. These commands are either to start or stop processes running on its system, to spawn a new agent, or to terminate itself. This method gives configuration and

Optimization Agents the ability to start any type of agent on any node in the system, since all nodes contain at least one General Agent at all times. Redundancy, as with the Configuration Agents, is built into the General Agents. The reason for redundancy in this case is not to preserve data, but rather to ensure that a node will remain part of the system. If there were only one General Agent on a node, and that agent stopped responding, the entire node would be disconnected from the system. However, if there are two General Agents per node, and one fails, the remaining agent can recreate the failed General Agent. Once again this behavior satisfies the self-healing autonomic property. The self-healing property of the General Agents reduces the chance that a node will be removed from the system due to agent failure, thus requiring less maintenance by human intervention.

4.6. System Topology

Our system focuses on the hybrid centralized and decentralized design. The system acts similar to a centralized system, with all information being contained in the President Configuration Agent. The database which is maintained contains information regarding all living agents within the system. Since all information is maintained in one location the system becomes easily maintainable and coherent. However, fault tolerance is handled in a decentralized manner. Data is redundant with both a primary and secondary Configuration Agent. Also there is replication of agents if any fail or are shutdown. Decentralized systems are insecure for the most part since nodes can join at any point and start sending data that may be incorrect. However in our system, all message transfer is encrypted, therefore any node that joins the system would not be able to communicate with other agents unless the correct certificates are used. Scalability resembles the decentralized topology. Any new nodes with the correct certificates can join the system and immediately begin communicating with other agents.

Our system has taken the advantages shown in a centralized and decentralized pattern while improving on the problems this topology encounters. Manageability and security are both problems in this type of topology however our system handles both of these cases to improve the system. Improving the centralized and decentralized topology there are no major downfalls to the system; although management is still more difficult than in a purely centralized system.

4.7. Network Communications

The communications system is central to any distributed or clustered system, but its role in an autonomic system is even greater. In addition to providing a mechanism for transferring data across a network, our system also had to satisfy the self-protecting autonomic property. We originally chose to implement this property by encrypting all system communication, to reduce the possibility of an attacker gaining access to system commands by monitoring unencrypted network traffic. However, in addition to the peer-to-peer communication between nodes, we realized that in certain cases we would need to broadcast a message to a group of agents. One of these cases is when a new General Agent is created and needs to announce its presence to the Configuration Agents. We later discovered that there is currently no way of encrypting broadcast messages, because broadcasts use the User Datagram Protocol (UDP) instead of the connection-oriented Transmission Control Protocol (TCP). We decided that the messages that needed to be broadcast to the entire system did not contain any sensitive information, so they could be transmitted unencrypted.

After some research into encryption methods we found that Java had built-in support for Secure Sockets Layer (SSL), a popular and trusted method for transferring encrypted data across networks. We decided that SSL met the needs for our secure peer-to-peer communication because it is capable of using strong 2048-bit encryption, and implementing it would not be much more difficult than using standard network communications because of the excellent SSL support in Java. We chose to use 2048-bit RSA encryption, and generated the *keystore* and *truststore* files. The keystore holds our private key, and the truststore tells the system to trust this key. These two files must be present on all nodes of the system for SSL communication to function.

Although we needed a method for sending a message to multiple agents simultaneously, broadcasting seemed inefficient. It was not necessary for all agents on every node to receive a broadcast. Each message that is sent is only destined for a finite group of agents, and a broadcast message will never need to be sent to all agents in the system. Broadcasting to the entire system is not necessary, so we decided instead to use multicasting. Java also has built-in support for multicast sockets, so adding this functionality was not difficult. With the use of multicasting, we wanted to be able to send a message to all agents of the same type by assigning each type of agent a different multicast address. For example, all Configuration Agents would be in one multicast group, and all General Agents would be in another multicast group with a different address. When we implemented multicasting in the system, however, we experienced a problem that was caused by our unique system architecture. In most multicast systems there is one process listening on a particular address and port. In our system there were multiple processes (agents) on a single system that were listening on different addresses, but on the same port. Although it is not possible to bind multiple processes to the same port using TCP, it is possible to do so with UDP because it is a connectionless protocol. The unexpected result was that if one process began listening on the multicast address 230.0.0.1 and another began listening on 230.0.0.2 on the same node and port, both processes would receive packets that were sent to either 230.0.0.1 or 230.0.0.2, which was not the desired behavior. We were able to correct this problem by assigning each multicast group a specific port to use in addition to the group's distinct multicast address. For example, the General Agents listen on 230.0.0.1:1200, while the Configuration Agents use 230.0.0.2:1201. Using this method we were able to successfully implement multicasting across the distributed system.

4.8. Messaging System

Once we had made decisions about the methods for agent communication at the network level, we had to design a system to pass information over the network and take the appropriate action when it is received. We had several options when designing the messaging system. We could have used ASCII text or an array of bytes to represent the message, but using one of those methods would require parsing and additional interpretation by the receiver. Instead, we decided to take advantage of a unique feature of Java to make the messaging system design both clean and scalable. We created a *MessageWrapper* class that includes the destination address and port, the origin address and port, and a message object, which stores the message payload. Using the *serializable* Java interface, we are able to send the actual *MessageWrapper* object over an encrypted network socket and receive it as an object, without the need for any parsing.

4.9. Summary

We chose the programming language, researched and chose additional tools to use for the implementation of the project, defined the basic properties of the system, and completing a thorough and detailed design. We decided to use Java as our programming language for its object-oriented and cross-platform properties. We determined that we would not be able to use tools such as Madkit and the Log and Trace Analyzer plug-in for Eclipse in the development of our system. We defined each of the autonomic properties in the context of our project. From the properties we decided to implement a mobile multi-agent system that would be able to satisfy all of the properties.

5. System Design and Implementation

ACMS was designed and implemented using the Java Standard Development Kit (SDK) version 1.4.2. ACMS uses an Object Oriented (OO) architecture, which allowed us to take advantage of useful OO concepts such as inheritance and encapsulation. Inheritance allows an outside developer to reuse our code, while encapsulation provides a modular division of the code. The modular division is created by dividing code into packages. Each package contains thematically related code; for example, ACMS contains separate packages for the agents, the messaging system, the database and the applications.

5.1. Object Oriented Architecture

The ACMS was designed using the Object Oriented principles of the Java programming language. During the design we focused on two questions: "Will it scale?" and "Is it extensible?" Our architecture reflects these questions with many points of extension for further functionality to be incorporated. ACMS is a modular design and each subsystem was developed separately within its own package.

5.1.1. Agent Architecture

The heart of the ACMS lies within the agents, of which there are four. Each agent is a class of its own, but much of the code is the same among all the agents, such as message passing and initialization. In order to organize the code and minimize the amount of redundant code, each agent extends an abstract Agent class (See Appendix B for UML diagrams). This abstract class contains all the code for starting the messaging system, along with other initialization tasks. The abstract class allows easy integration of new agents into the system by providing a robust code base to build upon. When developing new agents for ACMS, a developer may just extend the agent class and receive all the code necessary to interact with other agents; the developer can then focus on his or her agent's particular functionality.

5.1.2. Timer Task Architecture

The agents often perform tasks at a regular interval, such as querying all agents for their system status. A task is a section of code that an agent schedules and runs over a given period of time. The task is run in a separate thread and used to complete goals of the agents. Each agent has at least one task which performs any additional functionality missing from the message passing and implementation of messages. Examples are the sending of messages to agents by the President Configuration Agent and optimizing of the system are both tasks in the respective agents. Our architecture provides a common interface for creating these tasks. We have an interface called *Itask* which every task implements. Each task contains a basic run method that should be implemented by the developer to perform the required action. *Itask* extends *Thread*, which ensures that when the task executes it does not block any message handling or other tasks that must be completed in parallel. The developer instantiating a task does not need to be concerned with the specific timer task to create for a particular agent. The instantiation is handled by another class called *AgentTimerTask*, which contains the logic for allocating the correct task for each agent. We created the *AgentTimerTask* class implementing the Factory Design Pattern [20].

5.1.3. Messaging Architecture

The messaging system is contained in two packages. The first package contains the classes responsible for sending and receiving the messages; there is one server and client for TCP connections and a second server and client for UDP connections. The second package holds the Message class and a wrapper. The Message class contains a string that contains a command and an attachment of type *Object*. The attachment provides flexibility to the messaging system because any *Object*, or a group of nested *Objects*, can be sent across the network. The message attachment provides the agent architecture with a common transport protocol capable of sending any information that can be stored in an *Object*.

5.1.4. Database Architecture

The database was designed with speed of access as the core requirement. In order to meet this requirement we used a hash table to store information on each node, which provides the system with a O(1) search time for finding information on any node. We provide different types of search criteria by maintaining several lists of keys. These include a hash of keys for quickly finding any particular type of agent, such as fetching only Configuration Agents. The second list of hash keys we store is one entry for each node. This provides quick access for finding the list of agents on any particular node. We need to have a list of agents that are on each node in order to verify that there are exactly two General Agents on each node. The database is also *serializable* so that it can be copied and sent across the network. The database's ability to be sent to other agents is used for redundancy and to allow the information in the database to be available to the User Agent or future types of agents.

5.1.5. Job Architecture

We developed a common interface for communication between user applications and the ACMS. This interface is defined in a class called *Job*, which all applications must extend. The *Job* class allows us to start and stop applications, as well as to relocate them from one node to another. An application must be *serializable* in order to be moved across the network. In order to run an application on our system a developer must create a class that extends our *Job* class, in addition to the application that will be run on the system. The *Job* class is an example of the proxy pattern, and provides translation of messages from the ACMS to the underlying application and vice versa [21]. This allows new code to be integrated into the ACMS that was previously unavailable during compile time.

5.2. Agents

Once the design of the agent system was completed, we implemented the agent classes. The four classes that were created were the General Agent, Optimization Agent, Configuration Agent, and User Agent. The first three are the basis for running the system, while the User Agent is not an integral part of the system, but rather a link between the agent processes and the users. All agents inherit from an abstract class called Agent. The basic functionality of each agent is provided in this class, along with a constructor that starts and initializes both the *BroadcastServer* and *SSLServer* threads. All agents are multi-threaded, running processes in parallel to optimize their performance.

5.2.1. Configuration Agent

The Configuration Agent uses the timers to start either a *PresidentConfigurationTask* and *BroadcastEveryoneTask* or a *VicePresidentConfigurationTask*. The decision about which of those classes to start depends on whether the agent is a President or Vice President. The President then contacts all agents to determine their state. If a connection cannot be established with a particular agent, an *Exception* is thrown which informs the system that the agent is not correctly functioning; the malfunctioning agent is therefore removed from the system. Otherwise the agent will respond with information regarding itself, its system and if it is available. A scoring function (See Appendix G) is run on a particular node and returns a value. If the value returned is above a certain threshold the agent node is marked as available; otherwise, the node is marked unavailable. The availability is used in conjunction with load balancing; when a new application is requested to be started, the Optimization Agent will take into account the availability among other information to determine a location (See Appendix F). Both the Vice President and Optimization Agents request a copy of the table periodically. Since the table needs to be sent across the network, all variables contained in the *ClusterDatabase* need to be *serializable* to allow the object as a whole to be serialized.

5.2.2. Optimization Agent

The Optimization Agent starts its thread, which first determines that the correct numbers of agents are alive in the system and on each node. If an agent is missing or needs to be moved because of poor node performance, the Optimization Agent makes this decision. In moving an agent the Optimization Agent uses the *getGoodGeneralAgentNoConfig* algorithm. This will analyze the table, applying a series of rules to determine if there is a better node in the system than the one on which the agent is currently executing. An agent is moved if there is an available node in the system, while the agent's current node is not available. If a better node is found, the Optimization Agent sends a kill message to the correct agent. Once the agent is killed, the system would then realize there is an agent missing and, using the same algorithm, would choose the better system to spawn the new agent.

Once all agents are stabilized and it is determined that there is the correct number of each agent in the system, the Optimization Agent begins analysis on the database. Each node on which an agent is executing is rated by a scoring function. The scoring function is used to determine if a node in the system should be marked available or unavailable. Another algorithm called the *getGoodGeneralAgent* algorithm is used to determine the correct location for application processes to be run. While the *getGoodGeneralAgentNoConfig* algorithm will never return a node that has a Configuration Agent, the *getGoodGeneralAgent* algorithm can return any available node, but the number of applications running on a node is also considered. If there are two available nodes in the system and one of the nodes is running an application, the algorithm will choose the system that does not have any active applications.

5.2.3. General Agent

The General Agent receives commands from the Optimization Agent to start new *Job* processes on its node. The General Agent's goal is to begin those processes. When a General Agent receives a new *Job* to start it begins by instantiating a new *JobDispatcher*. The purpose of the *JobDispatcher* is to add the *Job* to the agent's active job queue, start the Job thread, wait for the thread to complete, and finally to remove the Job from the active queue after it finishes. The *JobDispatcher* itself is a thread, so the General Agent does not block while the *Job* is executing. Therefore, several *Job* processes can be running simultaneously.

5.3. Message System

The ACMS uses two types of communications: SSL and multicast (See Section 4.6). Although these are two fundamentally different methods for transmitting information at the network transport level, we used an intuitive messaging system to abstract the complexity of the lower level network protocols. The messaging system allowed us to send and receive messages in a simple and consistent manner, and to eliminate many repetitive code fragments.

5.3.1. Messages

We created a *Message* class, which stores only the content of the message. The message content consists of a string and an object. All messages use the string to identify what type of message it is and to pass any other information that can be represented as a string. However, there are cases when an abstract data type that cannot be represented by a string needs to be transmitted with the message. Typical examples of these data types are vectors, hash maps, or any type of object. In these cases abstract data types can be stored as objects inside the *Message* class and transmitted along with the rest of the message. The design we chose for messages is similar to the concept of e-mail. A message is comprised of a body (the string) and an optional attachment (the object). The only limitation to this design is that there is currently no inherent Java functionality for sending an object over a multicast socket. Due to this limitation, only strings can be sent in a multicast message because they can be easily converted into small arrays, which can be sent as datagrams.

In order to send the message to an agent, it must be properly addressed. The *MessageWrapper* class stores the Message and all information required to transmit the message over the SSL or multicast protocols. The addressing information includes the IP address and port of the destination agent, as well as the IP address and port of the sending agent so that the receiver will be able to contact the sender should a response message be required. Although messages are sent by the *SSLClient* or the *BroadcastClient*, we implemented methods in the Agent class to facilitate the process, which involves several lines of code. The Agent's message sending methods take a Message and a destination address as parameters, generate the *MessageWrapper*, and use either the *SSLClient* or the *BroadcastClient* classes to transmit the message to its destination.

5.3.2. Message Handling

When a network connection is accepted by an *Agent*, the data it receives is first processed by either the *SSLServer* or the *BroadcastServer*, depending on what port the data was sent to. These servers first construct a Message from the data they receive, and then create a *MessageWrapper* to store the *Message* and its addressing information. The entire *MessageWrapper* is then pushed to the back of the *Agent*'s message queue, and the server notifies the Agent of the presence of a new message. The *Agent* will pop the *MessageWrapper* off the front of the message queue, inspect the *Message* it contains, and then call the appropriate method to handle messages of that type.

5.4. Database

We created a class, *ClusterDatabase*, which holds a table containing information about all agents that exist within the system. The main copy of this table is stored with the President Configuration Agent. However, other agents can request for a copy of the table to perform analysis or for redundancy. The *ClusterDatabase* contains different reference tables pointing to fields in the table which provide quick searching features. The main table in the database consists of a *Hashtable*. The hashtable contains keys constructed from each agent's IP:Port combination, so that information about any agent can be easily found in the database. The reference tables do not store any duplicate information such as agent information. This saves on space because these tables only link to specific fields in the database rather than storing the information on their own.

A *Tuple* is an object created that contains data fields for all information we are collecting from each agent (i.e. IP address, port number, type of agent, and all system statistics). Also in the database are other hashtables that contain different key and object pairs to find information from the database more quickly.

The first hashtable used to reference the database is the *KeyList* hashtable. This table contains keys consisting of a string referring to the type of agent. These keys link to a *Vector* object that contains all agent IP:Port strings that are of the agent type specified by the key. This second set of keys allows us to easily verify that two Configuration Agents and one Optimization Agent are present. Otherwise, the database would have to be searched in its entirety to make these determinations. Navigating and obtaining information is quicker using a hashtable in place of an array or linked-list, especially since these types of checks happen very often within the system. However, when distributing the database the size is larger since there is an extra hashtable of references.

The second reference table used in the *ClusterDatabase* is a *NodeList*. This is also a hashtable which links IP address string keys to a *Vector* containing all IP:Port strings of agents existing on a particular IP address (or node in the system). This allows for specific nodes to be easily searched determining what agents are running on a system. This is very useful in the *GoodGeneralAgent* algorithm used in determining the best nodes to start new agents and schedule applications.

Along with the database and reference tables which are located in the *ClusterDatabase*, there is a *jobQueue Vector*. The *jobQueue* holds a list of all applications that are waiting to be scheduled on nodes in the system. The Configuration Agent receives messages that new application processes need to be started and these messages are added to the *jobQueue*. The Optimization Agent receives the *ClusterDatabase* periodically and determines if any nodes are available to run new applications in the system. The distribution of applications will be discussed in more detail in Section 5.5.

5.5. Jobs

A job is an application written specifically to run on ACMS; typically at GSFC these applications will be long computationally intensive processes. ACMS provides a management system that can run these applications on distributed network architecture with minimal user interaction and intervention. This comes at a small cost to the developer because most of the work has been done for them in the form of an Application Programming Interface (API).

5.5.1. Jobs are developed Independently of ACMS

The API details the interactions required for incorporating a user's application with ACMS. Most distributed applications will not be written specifically for ACMS, however using the Proxy design method, an outside application can be incorporated into ACMS post production. The proxy object acts as an interpreter for the user applications and ACMS. The proxy intercepts all messages destined for the ACMS or user application and using the API of the two programs as a guide converts messages from one program's API to the other programs API. As an example, this is the same as interpreter in a conversation between two world leaders who do not speak each other's native language; the interpreter listens to each leader's words and then repeats them to the other speaker in their native language. The proxy model allows user applications to be developed independently from ACMS.

5.5.2. Job Application Programming Interface

The API for a proxy begins with the definition of the class; the proxy must extend the *Job* class we have provided. Extending *Job* allows the developers to reuse our code, such as networking, and is required in order to incorporate the proxy into ACMS. The proxy must be a *Job* because the ACMS is always running and cannot be stopped, recompiled and restarted anytime a new application is ready. The proxy allows newly compiled code to be introduced into the ACMS without any recompilation of ACMS itself. The proxy must implement a *run* method that is inherited from *Thread*. The run method contains code that is written by the application developer and is executed when the proxy is run. The proxy must implement *Thread* in order to run alongside a General

Agent on a node.

5.5.3. Distributing Jobs

Jobs are distributed and run through a series of messages being passed and execution of code as new threads (See Figure 1). Figure 1 shows the life cycle of an application; the squares represent when a new agent receives the initial message related to a new application while the circles explain the process once agent receives the initial message. *Jobs* are distributed using serialization and a message. The proxy is serialized and attached to a message, which is then sent off to the Configuration Agent. The Configuration Agent then forwards the message to a General Agent. The general agent receives the message, gets the proxy and executes the run method. This starts the proxy on the same node as the General Agent. It is then up to the proxy to start the Application and inform the General Agent when the application has finished. In this version of the ACMS it is up to the application developer to ensure that all necessary class files and application files are available on all nodes in the system. In the future, the ACMS could be extended to automatically distribute all necessary class files to each node.

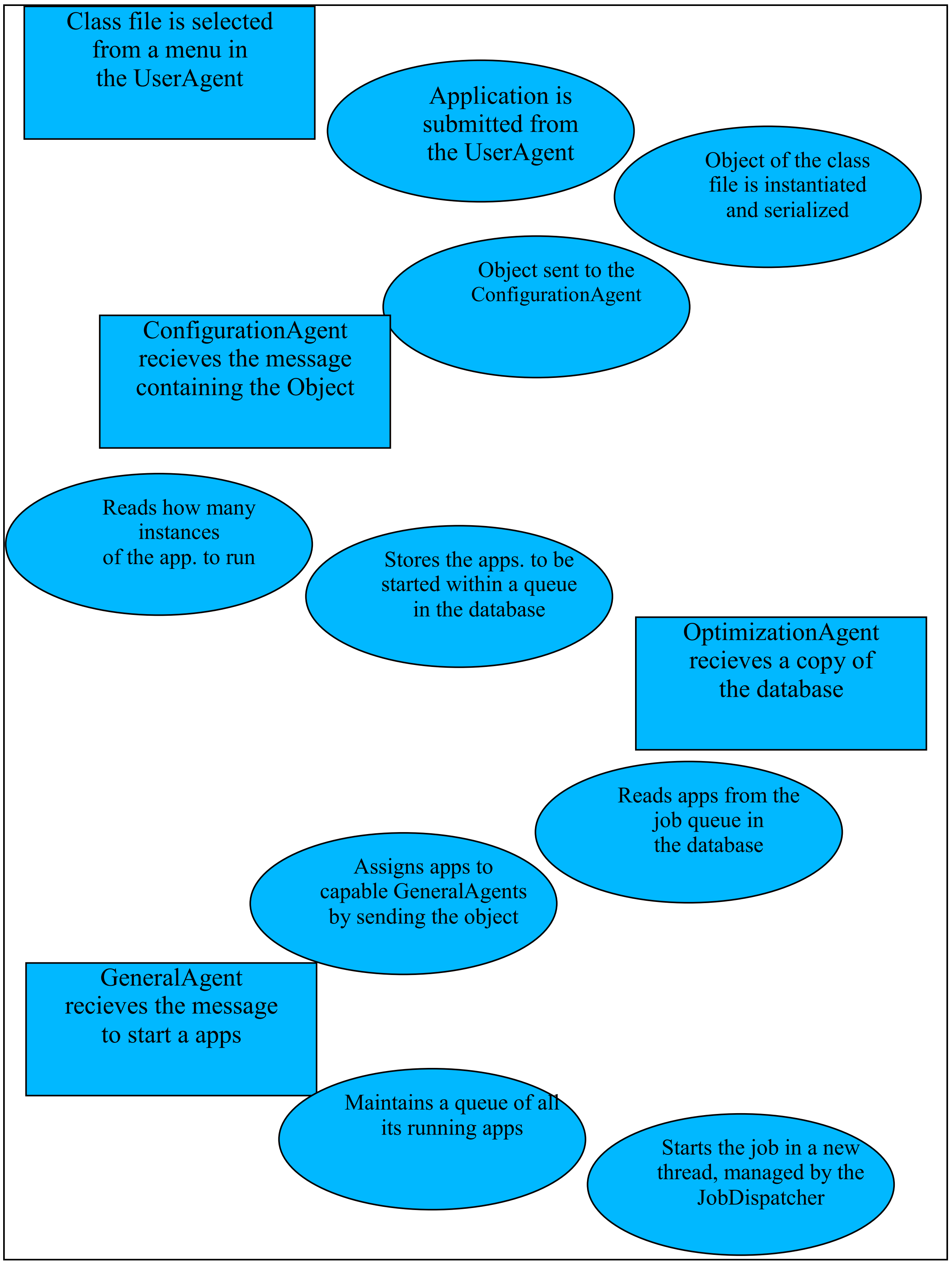

Class file is selected from a menu in the UserAgent
Application is submitted from the UserAgent
Object of the class file is instantiated and serialized
Object sent to the ConfigurationAgent
ConfigurationAgent recieves the message containing the Object
Reads how many instances of the app. to run
Stores the apps. to be started within a queue in the database
OptimizationAgent recieves a copy of the database
Reads apps from the job queue in the database
Assigns apps to capable GeneralAgents by sending the object
GeneralAgent recieves the message to start a apps
Maintains a queue of all its running apps
Starts the job in a new thread, managed by the JobDispatcher

Figure 1: Application Startup flowchart

5.5.4. Scheduling Jobs

The Configuration Agent and Optimization Agent are responsible for *Job* scheduling, but *Job* submission occurs in the User Agent. When a user clicks the button in the User Agent to submit a *Job*, the User Agent first executes the *main* method of the class the user selected. Executing the *main* method causes the *Job* to output a serialized copy of itself into the current directory. The User Agent then reads in this serialized file and encapsulates the *Job* in a *JobWrapper* object, which it then sends to the Configuration Agent.

The Configuration Agent receives the *JobWrapper* and adds it a queue. This queue represents all the applications that were submitted for execution on the ACMS, but which have not yet been started. This queue is stored in the *ClusterDatabase*, so the Configuration Agent periodically passes the queue to the Optimization Agent when it sends the database. The Optimization Agent then uses an algorithm to choose an agent to assign the next application to. This algorithm weighs an agent's current system performance and the number of applications currently running under that agent's supervision. The *JobWrapper* is then sent to the chosen General Agent to be run. The *JobWrapper* is not removed from the Configuration Agent's queue until the General Agent sends back an acknowledgment that it received the *JobWrapper*. If the acknowledgment is not received in a predetermined amount of time, the application is rescheduled and assigned elsewhere.

When the General Agent receives the *JobWrapper* it immediately sends an acknowledgement to the Configuration Agent. The General Agent then instantiates a

JobDispatcher and passes it the *JobWrapper* that was just received. The *JobDispatcher* executes in its own thread, and its purpose is to control the initialization and cleanup of the *Job* and to monitor the execution of the *Job*. When the JobDispatcher starts, it first adds the name of the *Job*, which is the string that is returned by the *Job*'s *toString()* method, to the agent's local job queue. The General Agent maintains its own job queue so that the Optimization Agent and the User Agent can determine the number and name of the user applications that are executing on each node in the cluster. After the name of the application has been enqueued, the *JobDispatcher* extracts the *Job* from the *JobWrapper* and calls the *startJob()* method on the *Job*, which actually begins the execution of the user application. Directly after starting the application, the *JobDispatcher* calls the *join()* method on the *Job*, which causes the *JobDispatcher* to block until the *Job* thread has terminated. The *JobDispatcher* is in a separate thread from the General Agent, so when the *JobDispatcher* blocks the General Agent is not affected. After the *Job* terminates, the *join()* method returns and the *JobDispatcher* removes the name of the *Job* from the General Agent's queue, at which point the *JobDispatcher* terminates.

5.6. Summary

The modular architecture of ACMS gives the system robustness and extensibility that would not be there otherwise. The focus during development was on scalability and extensibility. With the modular architecture the system can be further developed, such as replacing the messaging package with one that uses a non proprietary format or replacing the database package with an interface to a database application such as MySQL. The system can be customized by replacing and rewriting modules depending on the project ACMS is applied to. Every application has unique requirements and when we designed ACMS we built it to adapt and extend to meet all of those requirements.

6. Results

We designed several test procedures to verify that all aspects of the ACMS satisfy its design specifications. The main goal in developing test procedures was to provide methods for validating each of the ACMS' autonomic properties and observe that each element of the system performed as expected. Since we were also able to implement a framework for executing distributed applications on the system, we had the opportunity to evaluate the performance of the system from a cluster management perspective.

6.1. Validation of the Autonomic Properties

Scenarios have been organized and tested on our system to eliminate the possibility of problems occurring. Table 1 shows the seventeen different scenarios that were tested on our system. The table also contains any startup procedure that was needed to obtain the particular scenario and the results that should be expected if duplicated on our system. All seventeen scenarios that appear in Table 1 performed as expected when we tested them on the ACMS.

Table 1: Scenarios and Test cases

Scenario	Test Cases	Startup Configuration	Expected Results
1	Start up normal (one node in the system)	Start a single General Agent.	Five agents should be running on the system, including: two Configuration Agents, one Optimization Agent, and two General Agents.
2	Having one node in the system, have a second node join	Follow Scenario 1. On a second computer within the same subnet, run a General Agent.	The second computer should receive a second General Agent along with at least the Vice President Configuration Agent. Other agents depend on the availability of each computer, which computer would be a better host for the agents.

3	Run over the weekend (long-term)	Follow Scenario 2. Add any number of computers. Leave the system running over the weekend.	After an extended period of time, verify that all agents are still alive. This can be done by testing their functionality. Kill a General Agent and watch that it respawns (testing the Optimization Agent). Kill the Optimization Agent (testing the President). And finally, kill the President (testing the Vice President).
4	Kill a General Agent	Having a fully running system. In the User Agent choose to kill a General Agent.	The General Agent will be respawned on the same node in which it was killed.
5	Kill the President Agent	Having a fully running system. In the User Agent choose to kill the President Agent.	The President Configuration Agent will be respawned on a “good” node in the system (determined by the good node algorithm). See Appendix F.
6	Kill the Vice President Agent	Having a fully running system. In the User Agent choose to kill the Vice President Agent.	The Vice President Configuration Agent will be respawned on a “good” node in the system (determined by the good node algorithm). See Appendix F.
7	Kill the Optimization Agent	Having a fully running system. In the User Agent choose to kill the Optimization Agent.	The Optimization Agent will be respawned on a “good” node in the system (determined by the good node algorithm). See Appendix F.
8	Add a third General Agent to a node	Having a fully running system. In the User Agent choose to spawn a third General Agent on a node in the system.	The system will immediately terminate the extra agent from the system.
9	Add a second Optimization Agent to a system	Having a fully running system. In the User Agent choose to spawn a second Optimization Agent on a node in the system.	The system will immediately terminate the extra agent from the system.
10	Add a third Configuration Agent to the system	Having a fully running system. In the User Agent choose to spawn a third Configuration Agent on a node in the system.	The system will immediately terminate the extra agent from the system.
11	Start up system with two President Agents	Start two General Agents simultaneously.	The two General Agents will each spawn Configuration Agents at the same time. These will both not hear back from a President Agent and therefore promote themselves. When an Optimization Agent is spawned it will organize the number of agents and kill one of the Presidents.
12	Combine two full systems	Startup the system on two nodes. Disconnect one node from the network. This will cause the agents to complete each system. With two fully functioning systems, reconnect the node to the network.	The agents will detect each other and eliminate extra agents until only one complete system is left. On two nodes this would be two Configuration Agents and one Optimization Agent and two General Agents on each node.
13	Configuration agents separate with one node available and the other not available	Start up the system on two nodes. Have one system be available and the other unavailable. This can be done by running other applications to bring down the score value.	The Vice President Configuration Agent should still be moved to the not available node. The President and Optimization Agent should be located on the available node.

14	Migrating President Agent from a unavailable node to an available node	Start up the system on an unavailable node. Once all agents are running on this node, connect a second node to the system having this node be available.	When the second node connects, ensure that the President Agent is moved to the new node. This will be done by moving the Vice President first and then killing the President. Also the Optimization Agent will be moved to the new available node.
15	Three nodes ensure that agents don't bounce between nodes	Have the system fully running with three nodes in the system. Have one node be available and two not available.	The Vice President should remain stable on one of the false nodes. Ensure that it does not bounce between nodes.
16	Start a single application	Submit the application using the User Agent.	The application will follow the flow shown in Appendix A. It will be started on a General Agent.
17	Start a application with the number of Instances greater than one	Submit an application using the User Agent. Make sure the variable *numInstances* is greater than one so multiple applications will be submitted.	The application will follow the flow shown in Appendix A. It will be started on a General Agent.

6.2. Performance Evaluation

We created a simple distributed application to evaluate the performance of the system and its load-balancing abilities. The distributed application's only task is to find prime numbers. We chose this task for several reasons. It approximates the characteristics of many actual distributed applications, it is compute-intensive, and we were able to implement it relatively quickly. It is a task that can be easily partitioned into smaller tasks that can be executed in parallel, and the time required to exchange all necessary data is small compared to the execution time per data set.

6.2.1. Distributed Application Design

The distributed application we created to find prime numbers is separated into two elements: a server and a client. The *PrimesServer* generates discrete ranges of tenthousand numbers for the clients to check for prime numbers. There is a defined maximum for the total amount of numbers to check. Once this number has been reached the server broadcasts a termination command to the clients, then exits. The *PrimesServer* writes output to two files. One file contains timestamps that mark the first client connection and the time the server exits. The two timestamps represent the total run time of the application. The second log file contains a list of all the prime numbers that have been found by the clients.

The *PrimesClient* executes the algorithm that tests each number in a given range to determine if it is a prime. The prime checking algorithm is fairly simple; it divides each number by half of the numbers below it, with the exception of 1. If the remainder of the modulus division is zero, the number is not a prime, so it immediately stops checking that number and continues to the next. This brute-force algorithm is not the most efficient method to determine if a number is a prime. However, it suits our test case well because it requires a considerable amount of computation, and partitioning data sets for use with the algorithm is simple. One property of this algorithm that should be noted is that, as the number that is being tested increases, the length of time necessary to determine if it is prime also increases because more modulus division operations need to be performed. For example, to determine if the number 1000 is prime using this algorithm it is necessary to perform 500 - 1 = 499 modulus division operations, but to test the number 10,000 requires 5,000 - 1 = 4,999 operations.

6.2.2. Performance Evaluation of the ACMS

Once we had implemented and tested the distributed application for finding prime numbers we were able to use it to evaluate the performance of the ACMS. We designed four operational scenarios to measure any changes in the distributed application's performance by executing it on a cluster as opposed to using a single node. Each scenario is explained in the list below.

Execute the distributed application on:

1. A single node without the ACMS.

2. A single node with the ACMS' job infrastructure and load-balancing.

3. 5 nodes without the ACMS.

4. 5 nodes with the ACMS' job infrastructure and load-balancing.

Comparing the execution time of scenarios 1 and 3 allowed us to measure the approximate performance gain that could be achieved by harnessing the parallel processing power of a cluster over that of a single computer. Comparisons of scenarios 1 and 2, and additionally scenarios 3 and 4, allowed us to measure the approximate overhead of the ACMS and how that overhead changed as the size of the cluster increased. Finally, comparing scenarios 2 and 4 showed how well the ACMS scaled as the cluster size increased from one to five nodes. We performed each test three times to obtain a reasonable average. The average times for each test appear in the tables below. Individual results for each trial, as well as the configuration of the cluster and the network, can be found in Appendix H. All times are in hh:mm:ss format.

Table 2: System Evaluation – Result Averages

	Average Run Time	% of Test 1 Time	% Gain	% of Test 2 Time	% Gain
Test 1	0:49:51	100.00%	0.00%	95.53%	-
Test 2	0:52:11	104.68%	-	100.00%	0.00%
Test 3	0:11:02	22.13%	**451%**	21.14%	-
Test 4	0:11:24	22.87%	-	21.85%	**458%**

These results clearly demonstrate the power of distributed computing. Increasing the size of the cluster from one to five nodes resulted in a performance increase of approximately 451% without the ACMS and 458% with the ACMS. The maximum theoretical performance gain would have been 500%; we were very pleased that our results came so close to the ideal gain. It is important to note that when measuring the performance gain it is only reasonable to compare Test 1 with Test 3 and Test 2 with Test 4, which is the reason the other comparisons are not shown in the table.

Comparing the results from Test 1 and Test 2 gives an approximate value for the overhead associated with running the ACMS on one node. It is important to note that this approximate measure of overhead, about 5%, is the worst-case value for the overhead. This value is the highest possible overhead because all five agents, in addition to the user applications, were running on the same node. The ACMS guarantees that in any configuration with more than one node there will be no more than four agents on any single node, and most nodes will only have two agents. Therefore, as the number of nodes in the system increased, we expected the overhead caused by the ACMS to decrease. This prediction was confirmed when we performed the last two tests.

In comparing the results of Test 3 and Test 4 it is clear that the overhead due to the ACMS was significantly reduced. As the number of nodes increased from one to five the overhead decreased from almost 5% to less than 0.75%. This result is significant because increasing the cluster size by a factor of five actually decreased the overhead by a factor greater than six. We were encouraged by this result because, although we were not able to test the ACMS on a large cluster, the data imply that the system scales efficiently. Figure 2 shows the overhead associated with the ACMS in the one node and five node tests.

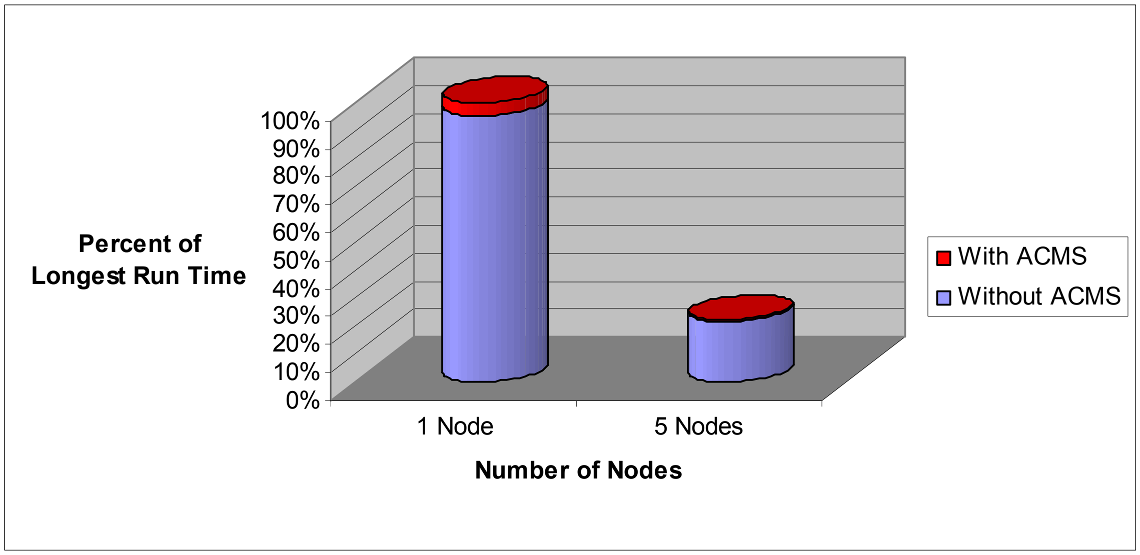


Figure 2: Run time comparisons

6.3. Summary

We were very pleased with the results from the autonomic property tests and the performance evaluation of the system. The autonomic property tests showed that the ACMS satisfied the requirements of an autonomic system. The performance evaluation demonstrated that we had accomplished our goal of developing an autonomic system with less than 5% overhead. The tests also imply that the system scales well as the number of nodes in the system increases. Increasing the number of nodes both decreases the overhead significantly and increases the performance gain for distributed applications at a rate that is close to ideal.

7. Recommendations and Conclusion

We incorporated as much functionality into the ACMS as we were able to do in our time at GSFC, but we were limited by the ten-week duration of the project. Throughout the project we have thought of several ways in which the principles of the ACMS could be applied to other systems. We believe that the ACMS could be extended to become a more robust and flexible management system, and that it could serve as the foundation for a diverse array of applications.

7.1. Suggestions for Future Development

At the conclusion of our project we determined several aspects of the ACMS that could be developed further. We first noted areas of the current system that, if we had more time, we would have continued developing. We also thought of several ways in which the system could be easily extended to perform tasks that it was not originally designed to do.

7.1.1. Outstanding Development Tasks and Evaluations

The tasks that we identify in this section concern the incorporation of additional functionality in the ACMS. The goal of this future work is to make the system more robust, usable, and scalable.

7.1.1.1. Node Statistics and Scoring

Gathering statistics about each node is important so that the ACMS can determine which nodes are best suited to receive new agents and user applications. While the Linux

/proc filesystem enabled us to easily obtain all the information we required making decisions about node availability, we found no convenient way to do so in the Windows operating system. At the very least, gathering information about a Windows computer would require writing an additional program in Visual Basic, C++, or C#. As we did not have access to a Windows IDE, we were unable to write such a program, and the ACMS is currently not capable of obtaining any information about Windows nodes. However, we included support for Windows node statistics in the ACMS, so that it would be simple to make this information available to the system if someone were to implement a program or other method to retrieve it.

Another challenge that was present throughout the project was how to best determine the availability of each node. We designed a simple algorithm to compute a score for each node, but found that it was very difficult to compute a score that was representative of the state of each node (See Appendix G). Additionally, it was equally difficult to define a standard score threshold, which is used to make the final determination of node availability. We recognize that determining the availability of a particular node in a cluster is complex problem and is outside the scope of our project. However, it is an interesting research topic which future groups may wish to consider.

7.1.1.2. Job Infrastructure and Load Balancing

We have identified two areas relating to the execution of distributed applications that would benefit from further development. The first relates to the distribution of application code. In our current system, if a user wishes to execute a distributed application, all of the application's class files must reside on each node in the system. Our original design goal for user applications was to be able to submit an application from any node in the system and to have the code be automatically distributed to each node on which it will execute. After additional research we realized that automatically distributing all necessary files would be a difficult task. We decided that, due to our time constraints, we would instead implement only the basic functionality that would allow an application to be executed on the system. It is our opinion that future development efforts in the job infrastructure should begin with a method for automatically distributing all required binary files. We suggest research into Java archive (Jar) files that can be transmitted as discrete entities attached to the *JobWrapper* object, and then either uncompressed or accessed directly by the General Agent upon receipt.

The other outstanding issue with distributed applications is load-balancing. Although load-balancing is performed initially when a user application is first assigned, no further actions are taken after the application has begun executing. The topic of post-assignment load-balancing is also far from the scope of our project. However, it may be a subject worth researching in the future. It is certainly not a trivial task to move a user application once it has begun executing. The application may use temporary working files, sockets, and several forms of inter-process communication. Given these challenges, the cost incurred by relocating processes may far outweigh any potential gains associated with an optimal load balance. However, we feel that relocating processes is a topic that needs to be investigated if a system like ours is intended to be used in a production environment.

7.1.1.3. Integration of Open Standards for Interoperability

We designed and implemented every aspect the ACMS ourselves, using only the standard Java libraries. In doing so, we developed a system designed with our specific needs in mind. This design allowed us to rapidly implement the system because we did not need to spend time learning external tools or technologies. However, if the system were to be extended in such a way that interoperability among different technologies were required, some parts of the AMCS could be redesigned for greater compatibility. The messaging system should be the primary focus if interoperability is a requirement. The use of open standards and technologies such as XML, CORBA, or RMI could replace the exchange of *Message* objects for inter-agent communication. A similar approach could be taken with the job infrastructure if a more standard, open technology is needed.

7.1.1.4. Scalability Testing

During our testing we did not have access to an actual large-scale cluster. The only resources we had at our disposal were an assortment of workstations and our personal laptops, from which we constructed an ad-hoc system more akin to a grid than a cluster. Although we were able to conduct some limited testing of the system, we were not able to test the system's ability to scale to hundreds or thousands of nodes. This lack of resources was unfortunate because we integrated functionality into the system designed to compensate for the increased time required to manage a large number of nodes, but we were unable to thoroughly test it. Although we do not have any test data for large clusters, we predict that at some point there would simply be too many nodes for a single Configuration Agent and Optimization Agent to effectively manage. If a group wishes to use our system in some real-world application, we suggest that the configuration and optimization tasks be further divided and disseminated among a greater number of nodes to ensure that these tasks will be completed in a timely and efficient manner. Of course the partitioning of the configuration and optimization tasks into smaller sub-tasks would require more advanced algorithms for coordination between each sub-task, but it would probably be necessary for a system with a large number of nodes.

7.1.1.5. Improving the Usability and Aesthetics of the User Agent

The ACMS was not designed to be a finished product ready for use in a production environment; throughout the project we were focused more on functionality than usability and aesthetic qualities. However, these properties are essential in a system that is intended for use on a large scale. Any future work on the ACMS should be concerned with evaluating the usability of the User Agent, as this agent is the only way for a user to interact with the system. The User Agent should comply with established standards of usability, such as those proposed by the World Wide Web Consortium (W3C), in addition to any government or NASA specific guidelines. Special considerations should be taken for users with disabilities, and a command line version of the agent might be useful for system administrators.

7.2. Possible Extensions to the ACMS

As we have previously mentioned, extensibility has always been a major consideration in the design of the ACMS. We have put forth every effort to implement the system in a modular way that can be easily extended. Given this flexible property of the ACMS, we believe that its system management abilities could be modified or extended to be used in numerous applications.

7.2.1. Management of Grid Systems

The management of entire grid systems is a natural extension of the ACMS. In fact, the ACMS is already capable of managing simple grid systems, if the definition of a grid is taken to be a dynamic, heterogeneous network of computers. There are, however, more complex forms of grids that the ACMS in its present form may not be able to manage. For example, a grid may consist of a large network of workstations on different subnets, a large cluster on a private network segment, and several mid-range servers in a demilitarized zone (DMZ). The complex issues that arise in this case are the potentially different broadcast addresses that may exist on the grid, as well as the fact that there may not be a route from each node to every other node. Although some grid configurations present unique management challenges, we believe that the ACMS could be extended to handle these cases through relatively minor modifications to the Configuration Agent. One can imagine an ACMS configuration in which each network segment contains a complete ACMS and the Configuration Agents on the outward-facing nodes all communicate with each other. These outward-facing Configuration Agents would each have a complete list of every agent in the grid. In addition to an agent's IP address and port, each agent would also provide routing information. In its simplest form, the routing information could be the address of the Configuration Agent that is responsible for the agent. A more sophisticated form of this system would be able to determine the best way to route messages between two networks. There are many ways to implement a grid management system based on ACMS concepts, and because of its modular and extensible design, it would be possible to create such a system with few modifications.

7.2.2. Management of Intelligent Clustered Spacecraft

A practical application of the technology we developed for the ACMS might be an autonomic management system for a cluster of unmanned, autonomous spacecraft. Although we are not experts in the management tasks associated with unmanned spacecraft, it is possible that these tasks could be represented as applications that can execute on a cluster. Even if the cluster analogy does not perfectly translate to a community of spacecraft, autonomic principles could still be applied in a future management and control system for these craft. An important benefit of using autonomic systems is their potential to reduce overall cost. Fewer people would be required for routine administration because many of the spacecraft's functions would be self-managed. If these spacecraft were self-protecting, there would be a much smaller chance of losing them due to a system malfunction, an external impact, or another predictable event.

7.3. Increasing Awareness of Autonomic Systems

One of the goals of our project was to increase awareness of autonomic systems among NASA scientists and engineers by providing a practical application of the autonomic properties. In order for NASA employees to become educated about autonomic systems, they should be able to see examples of working systems. NASA should take the initiative in developing autonomic systems to solve some of its real challenges, and then explain how the autonomic properties allowed for a unique and robust solution. We hope our development of an autonomic system will encourage NASA engineers to use autonomic concepts in their own work.

Appendix A – Diagrams

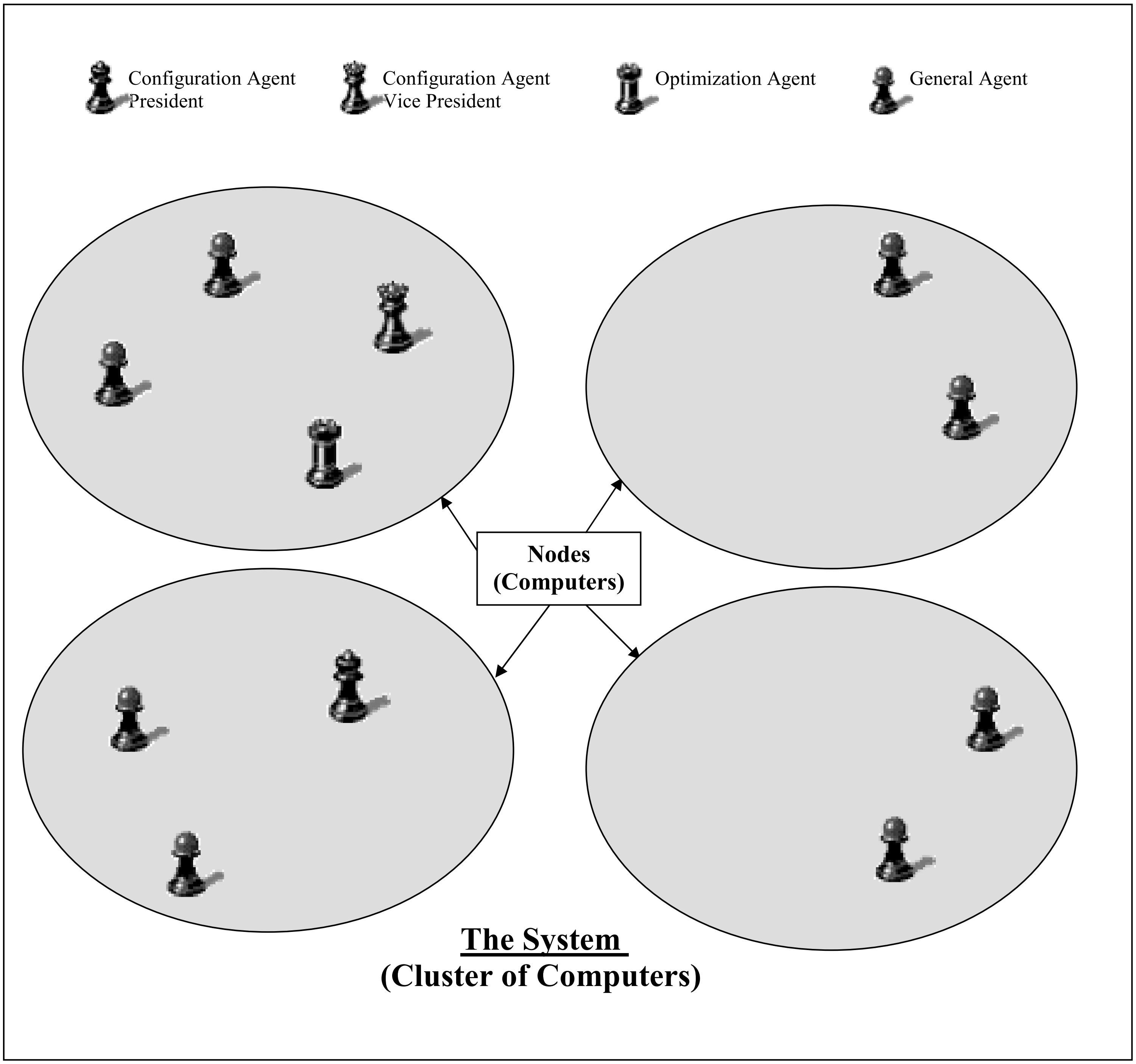


Figure 3: Agent System Design

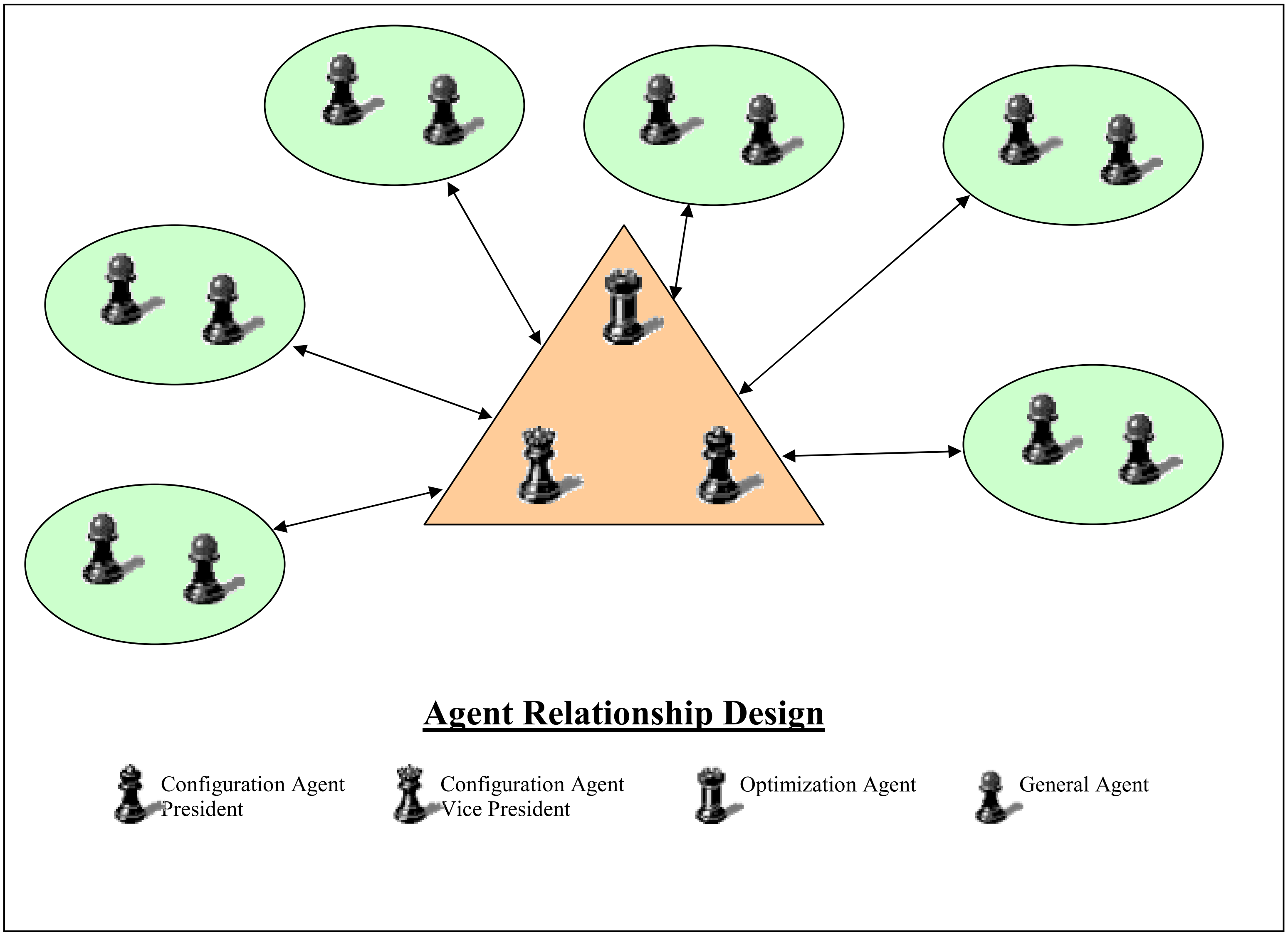

Agent Relationship Design
Configuration Agent President
Configuration Agent Vice President
Optimization Agent
General Agent

Figure 4: Agent Relationship Design

Appendix B – UML

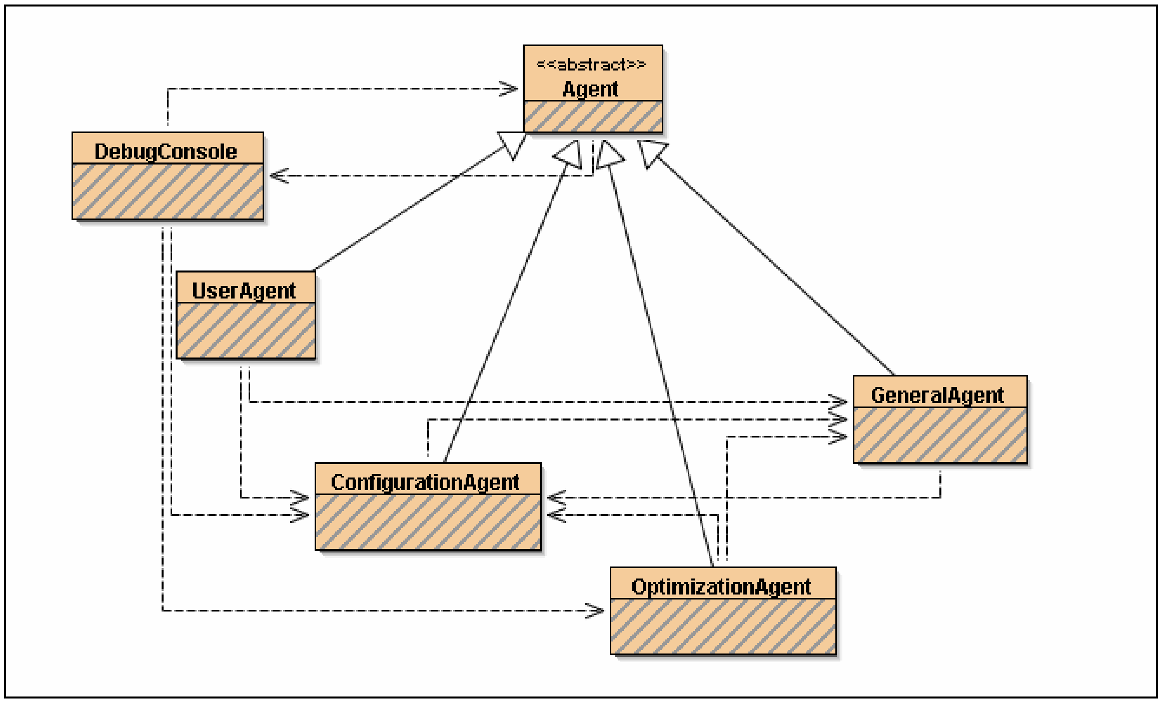


Figure 5: Agent UML Diagram

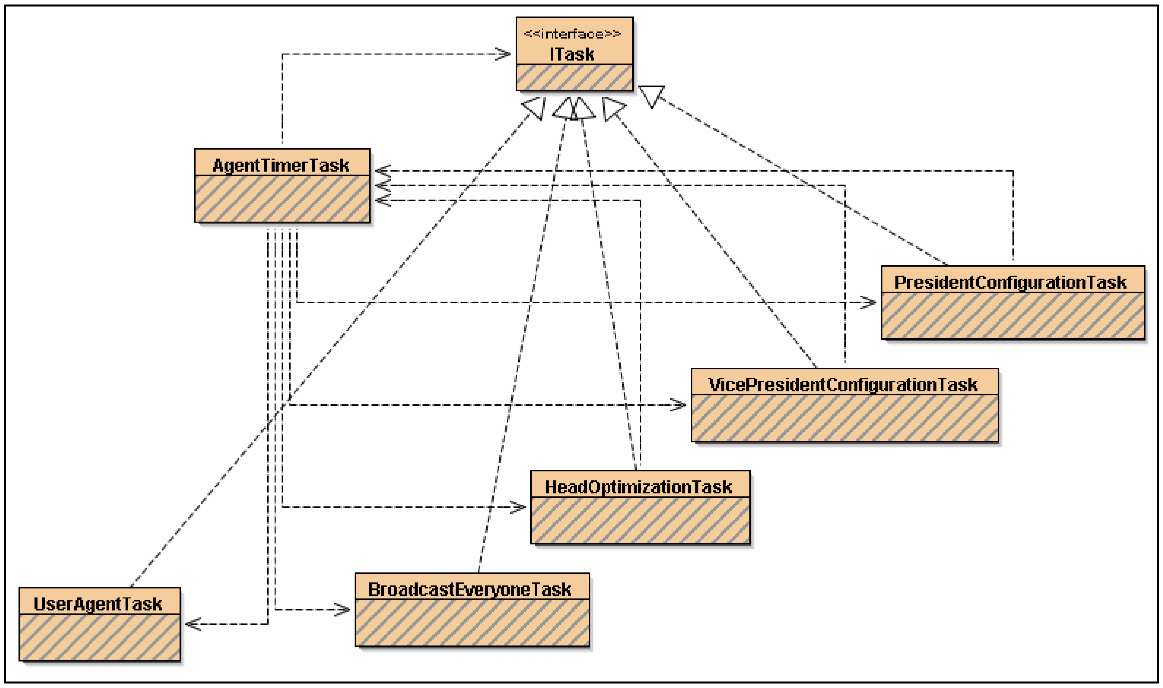


Figure 6: TimerTask UML Diagram

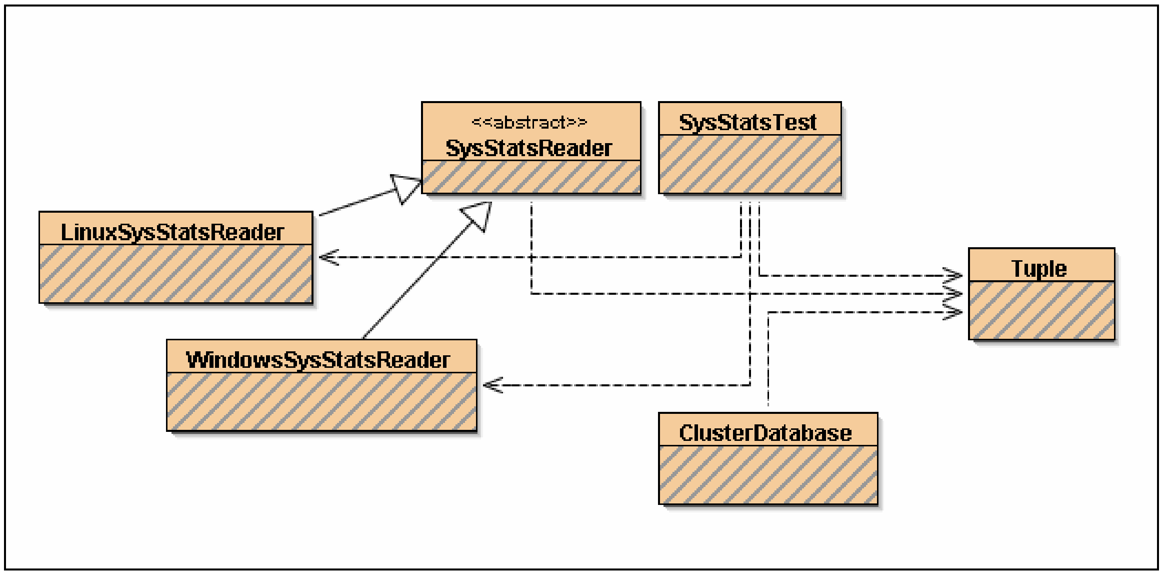


Figure 7: Database UML Diagram

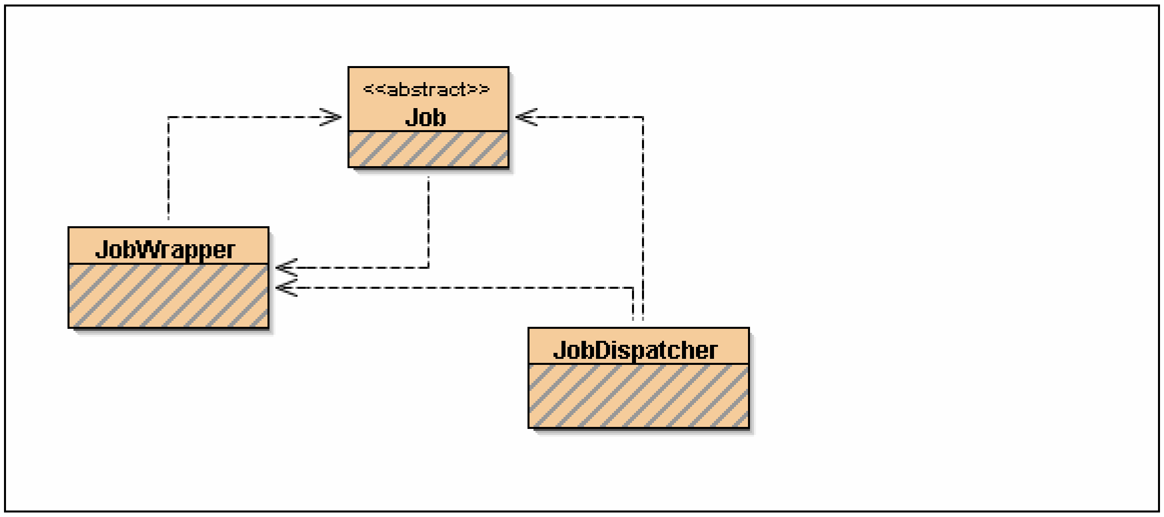


Figure 8: Job UML Diagram

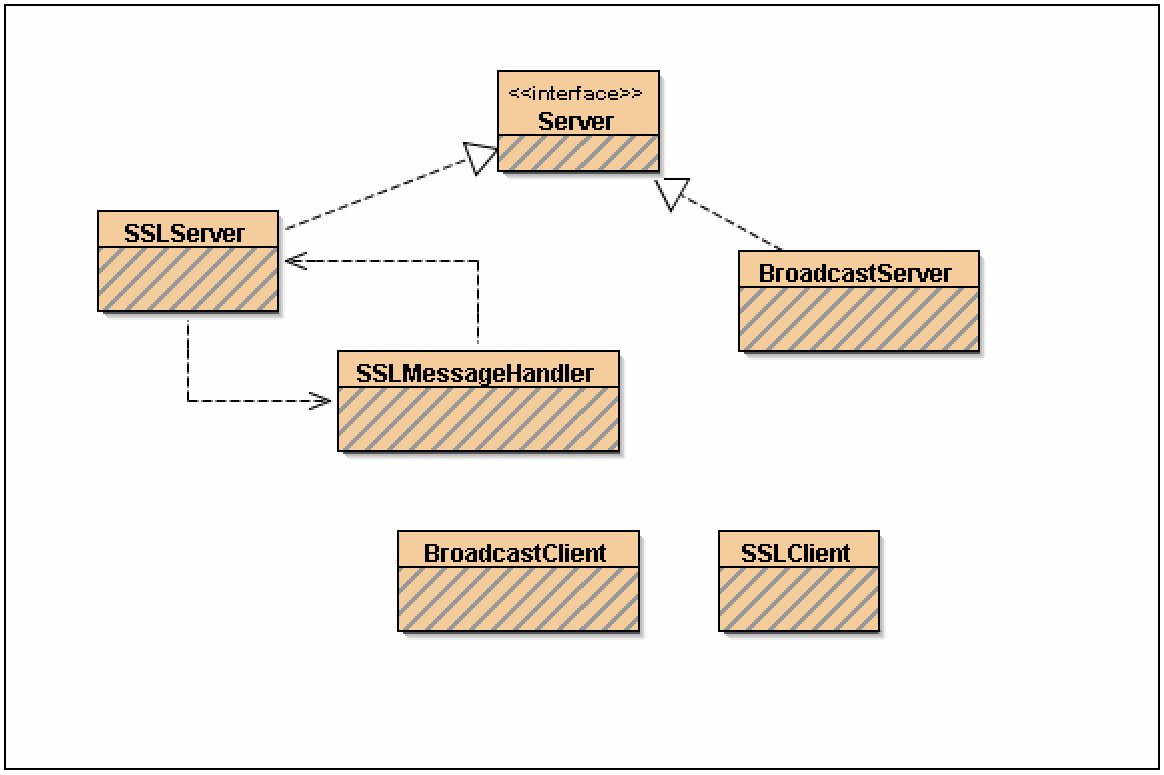


Figure 9: Messaging System UML Diagram

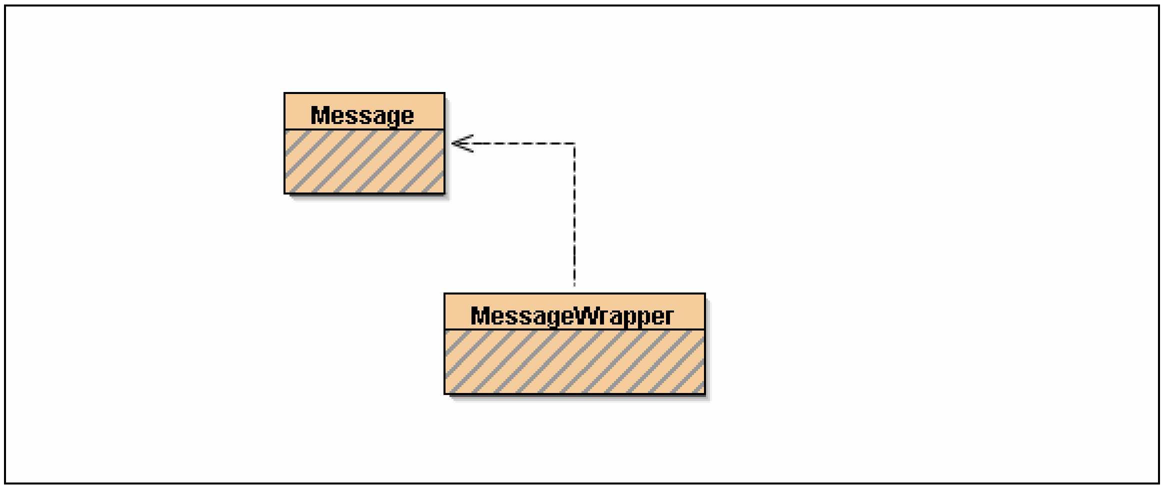


Figure 10: Messages UML Diagram

Appendix C – Properties

Self Configuring

- Nodes and Jobs can be added to the infrastructure without disruption to the system
- Discovers agents automatically
- Maintains a database (directory) regarding agent locations for communication and information specific to the agent's system
- Applies rules to the system regarding specifications required to be stable (i.e. Two Configuration Agents, one Optimization Agent and two General Agents)

Self Healing

- Maintenance is performed to obtain current information from all nodes in the system and determine if any agents are not functioning
- Detects if an agent in the system is no longer functioning correctly; if found it will remove the agent from the system and reintroduce the agent without any disruption
- New agents are started on systems able to handle the load of an extra agent

Self Protecting

- The use of Redundant agents on different nodes in the system to prevent loss of information
- All redundant agents are as secure as the original agents since they all contain the

same code as the primary agents which can be executed if necessary

- Access to the system is managed by use of encryption in communication, sending information across SSL and digital certificates
- Agents will not be started on systems unable to handle the load of another agent

Self Optimizing

- The ability to choose good locations to start new agents and applications (uses knowledge of the system to make choices)
- Maximizes the performance from the system by distributing processes between nodes
- Performs balancing techniques on agents and applications if nodes start becoming very heavily loaded

Appendix D – Job Implementation Guide

How to Write a Job:

Step 1: Create a new class that extends acms.job.Job

Step 2: Define the constructor

An application must have three lines in order to be setup correctly to interact with ACMS. These three lines are to provide a reference to the new application, determine the number of applications to distribute, and create the serialized version of the application. An example code snippet is shown below. The first and last line should be copied verbatim, while the second lines number should be changed depending on the requirements of your application.

Example 1:
```
thisJob = this;                // This line is necessary for all jobs
thisJob.setNumInstances(5);    // Schedule as many of these jobs as the system can
                               // accommodate
thisJob.generateJob();         // This line is necessary for all jobs
```

Step 3: Define the main method

The main method in an application must create a new instance of your *job* class. This is the only line necessary in the main method, as shown below. Alternatively for testing without integration with the ACMS, include a line that invokes the run method; an example of this line of code is shown below as well.

Example 2:
```
Job pc = new PrimesClient();    // Serializes this job when the main() is
                                // called
//pc.start();                   // Necessary for command line testing
                                // otherwise should remain commented
```

Step 4: Define a toString method

Every application must implement the standard toString method in Java. The string returned is the application's name and must match the class name verbatim. For example if your application were named *PrimeCalculator* and contained in the class file PrimeCalcultaor.class, then toString should return the String "PrimeCalculator".

Step 5: The run method

These methods are where the actual application programming takes place. The run method is called when your application is ready to be run. It is in this method that you should start any outside applications or processes necessary for your application. The run method should not return until your application has finished all of its processing. After the application is finished, simply have the run method return from the function call and ACMS will consider your application as completed.

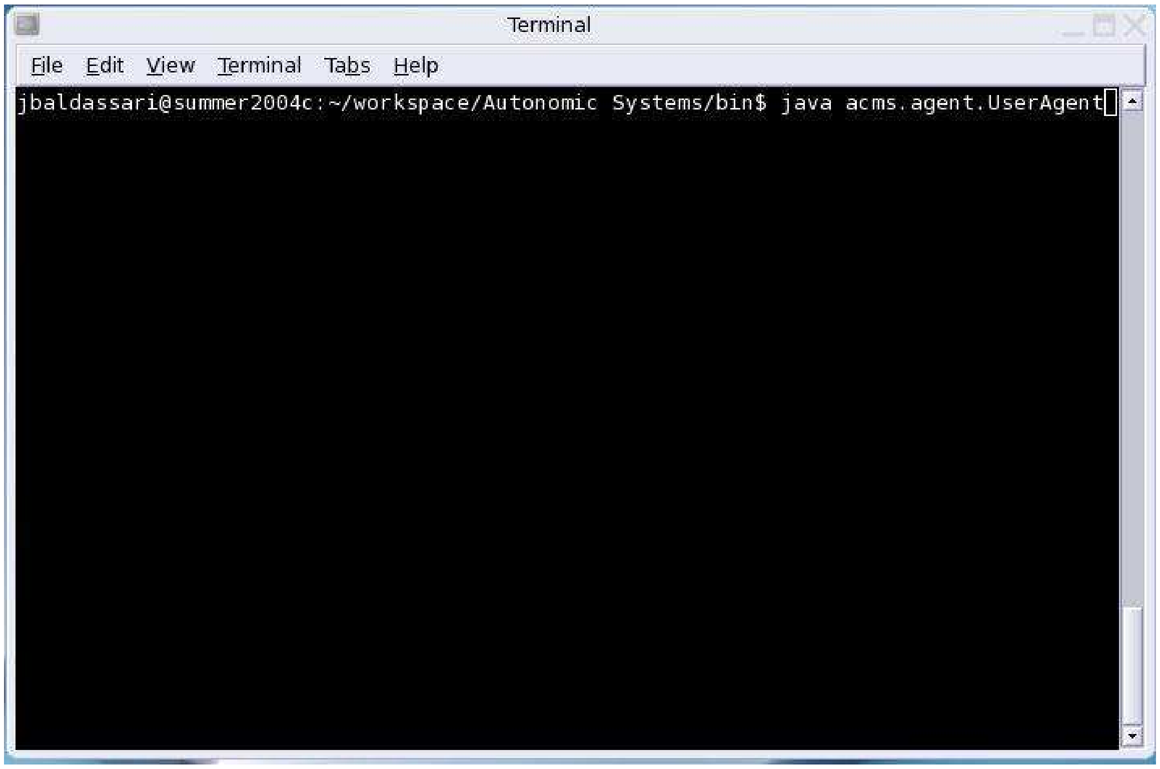
Additional Resources:

If you follow these steps your application is now ready to be launched using the User Agent GUI (See Appendix E). If you require additional help in writing a application take a look at the sample *PrimesClient* and *PrimesServer* located in the *acms.job.primes* package. These have detailed documentation line by line about what is necessary for all applications. It is also a prime example of how to use the basic messaging included with ACMS. Always remember to test your application independently of the ACMS system first to ensure your application is written correctly before submitting it to ACMS.

Appendix E – ACMS Users Guide

Software Requirements: java 1.4.2 or later

1. Starting the GUI
    a. Open a terminal or command shell
    b. Locate the directory where ACMS was installed on your system
    c. Execute the command "java acms.agent.UserAgent"



```
jbaldassari@summer2004c:~/workspace/Autonomic Systems/bin$ java acms.agent.UserAgent
```

Figure 11: Terminal Console window – Command for starting an Agent

2. The Database Panel

The database panel is the default panel when the GUI is loaded. This panel contains information on every agent in the system, including system information. This panel will be empty and gray unless there is a Configuration Agent online.

By Double Clicking on any of the table rows a pop-up menu will appear with several options.

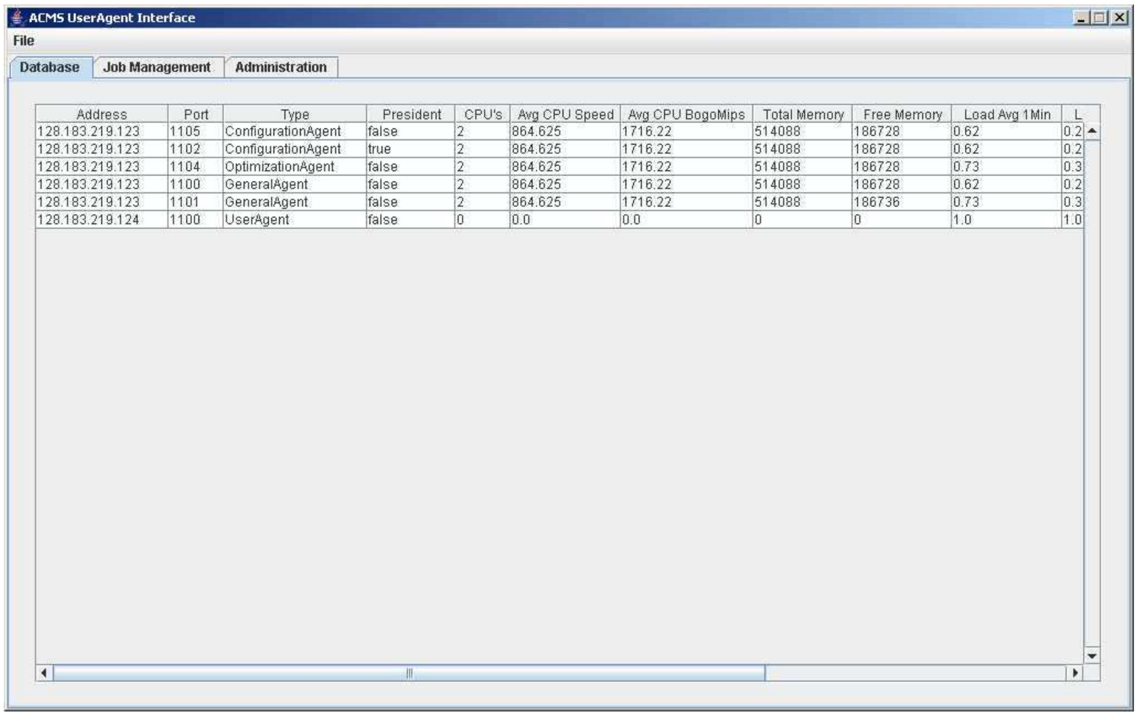
a. Spawn a Configuration Agent

b. Spawn an Optimization Agent

c. Spawn a General Agent

d. Destroy an Agent

The agent will take the action listed on the button. These functions provide administration powers to the user. This is helpful if the administrator would like to remove a computer from the cluster, they can just kill the agents on a particular node.



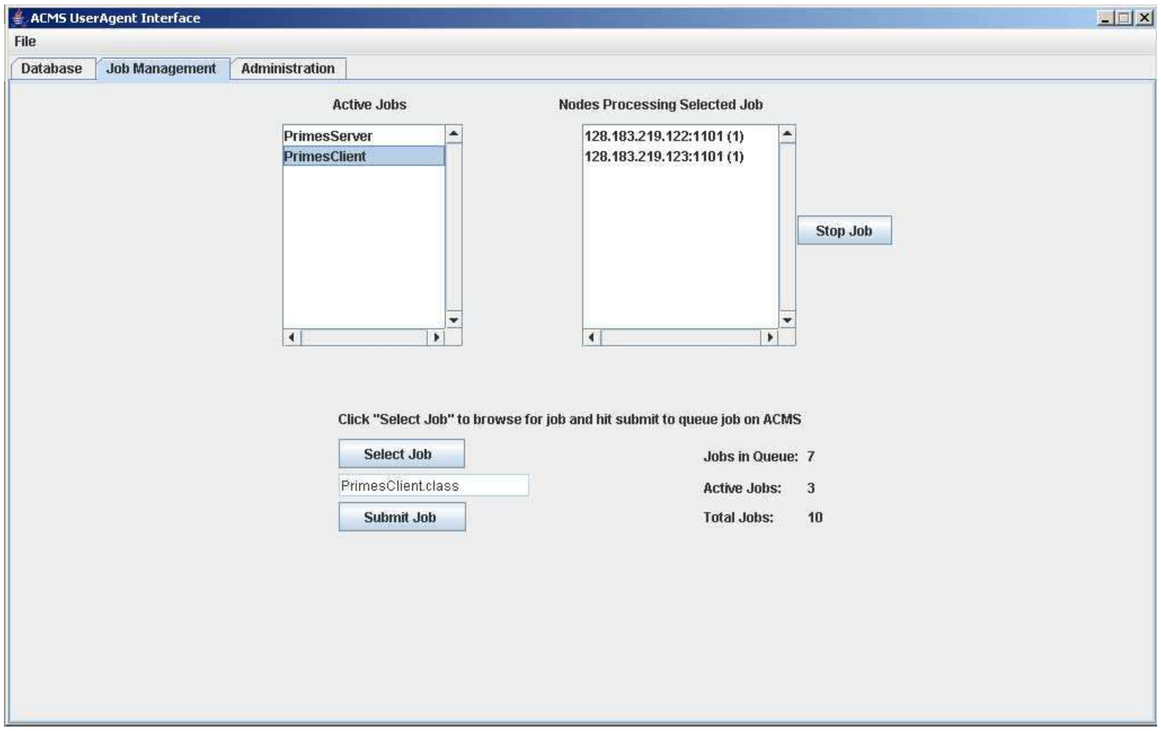
Address	Port	Type	President	CPU's	Avg CPU Speed	Avg CPU BogoMips	Total Memory	Free Memory	Load Avg 1Min	L
128.183.219.123	1105	ConfigurationAgent	false	2	864.625	1716.22	514088	186728	0.62	0.2
128.183.219.123	1102	ConfigurationAgent	true	2	864.625	1716.22	514088	186728	0.62	0.2
128.183.219.123	1104	OptimizationAgent	false	2	864.625	1716.22	514088	186728	0.73	0.3
128.183.219.123	1100	GeneralAgent	false	2	864.625	1716.22	514088	186728	0.62	0.2
128.183.219.123	1101	GeneralAgent	false	2	864.625	1716.22	514088	186736	0.73	0.3
128.183.219.124	1100	UserAgent	false	0	0.0	0.0	0	0	1.0	1.0

Figure 12: User Agent – Database view

3. The Job Panel

The Job Panel is used for viewing active and queued applications on the cluster. This panel is also used for submitting new applications for processing. At the top of this panel are two list boxes. The left box contains a list of all the active processes on ACMS. Selecting one of the processes in the left box will populate the right list box. The right box contains a list of addresses and a number in parentheses. The addresses correspond to agents that have been assigned the selected application. The number indicates how many instances of the application that particular agent has been assigned. The stop job button will stop the application or applications selected in the right list box.

To submit an application click the select job button and locate the class file of the application you would like to submit. When you have selected it the name of the file should appear in the text box just below the select job button. Double check to make sure this is the correct file. Next select the submit job button, the GUI now automatically sends your application for processing. In a few moments the job will appear in the queue on the GUI. Shortly thereafter the job is removed from the queue and will appear as active job in the left list box.



PrimesServer
PrimesClient

128.183.219.122:1101 (1)
128.183.219.123:1101 (1)

Click "Select Job" to browse for job and hit submit to queue job on ACMS

PrimesClient.class

Jobs in Queue: 7
Active Jobs: 3
Total Jobs: 10

Figure 13: User Agent – Job Management view

4. The Administration Panel

The Administration Panel contains a single button that can stop ACMS. Pressing this button initiates a dialog which asks for conformation before shutting down ACMS. Once the action is confirmed there is no way to stop it. All agents and applications will be stopped immediately. In order to restart the system it will be necessary to start a General Agent on each node individually.

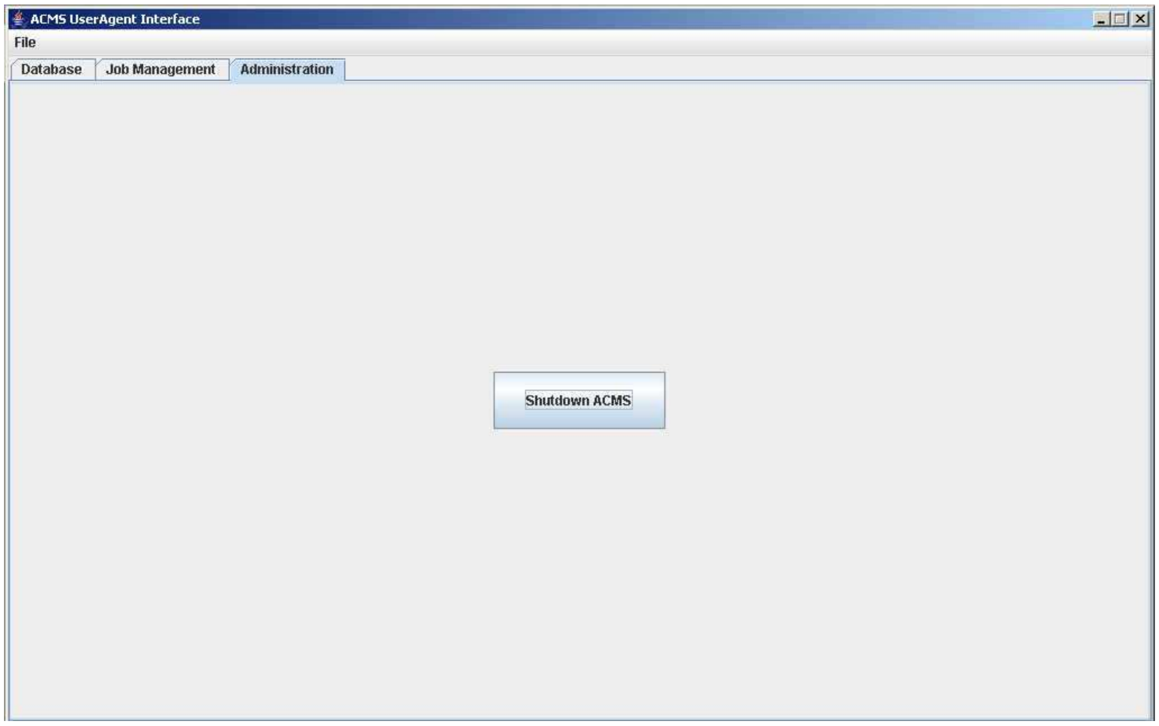


Figure 14: User Agent – Administration view

Appendix F – GetGoodGeneralAgent Algorithm

The purpose of *getGoodGeneralAgentNoConfig* and *getGoodGeneralAgent* algorithms is to determine a well suited node for an agent or application. Each algorithm applies a list of rules to determine if there is a "good" node in the system. These can be used to place new applications and agents when a node enters the system or to determine if moving an agent in necessary. It allows for foreshadowing to see if there exists a more equipped node for an agent. An example of this would be if there were two nodes, one available and one unavailable. The unavailable node had the President Configuration Agent. Since the President Agent does a large amount of processing and message passing it would be beneficial to have the President on the available node. Therefore the *getGoodGeneralAgent* algorithm would return the available node as a result. The term "good" is used to define a node that these methods would return as results.

getGoodGeneralAgentNoConfig Algorithm - Agents

The purpose of this method is to return a node that does not contain Configuration Agents. There are a number of rules that the method applies in order to determine a "good" node in the system. The algorithm will first try to find a node in the system that is available and only contains General Agents. If one isn't found it then looks for a node using the following rules: an available node that does not have Configuration Agents, an unavailable node with only General Agents, and an unavailable node without Configuration Agents. If none of these conditions are met it will return null. This method is mainly used for placement of the two Configuration Agents and the

Optimization Agent when they enter the system or need to be relocated.

getGoodGeneralAgent Algorithm - applications

The purpose of this method is similar to the previous with slight modifications. For choosing a "good" node for application location there is no need to exclude Configuration Agents like in the previous algorithm. Rather the exclusion here is with unavailable nodes. If a node is unavailable it should not receive an application. There is a similar rule set as the first algorithm had, it will start off by choosing a node with General Agents and/or Vice President Configuration Agent while running no applications. If an agent is not found it will apply the following rules while trying to obtain a node: a node with any type of agents and no applications, a node with only General Agents and/or Vice President Agent and the fewest applications, and a node with any agents with fewest applications. If none of the rules are applied then the method will return a null result. The fewest applications factor was added into the algorithm so that applications are distributed evenly as each agent obtains more to run.

Along with the fewest applications factor for nodes, there is also a *generalAgentCache*, which is a *Vector* maintained in the Optimization Agent. It acts as a cache so that no agent can receive consecutive application assignments too quickly. This algorithm takes the cache into account when determining a location. When an agent is assigned a new application its node IP address is entered into the cache and remains there for a set time. When the time is finished it is removed from the cache and will then be available for another application assignment.

getGeneralAgent - All

This method is used in conjunction with the other two algorithms. The other two algorithms will not always return non-null results, therefore if the result is null there needs to be a method to obtain a non-null result. This function will always return a result unless there are no General Agents in the system. The *getGeneralAgent* method will return an available node; if no available node exists then it will return an unavailable node in the table. This method is used with the *getGoodGeneralAgentNoConfig* algorithm since unavailable nodes can still be accepted unlike with the application algorithm.

Appendix G – Scoring Algorithm

In order to determine which nodes in the system were able to execute new agents or applications, we needed a standardized method to evaluate the availability of each node. We implemented this evaluation in the form of the scoring algorithm. The scoring algorithm examines multiple aspects of the computer's resources, including the number and speed of the CPUs, the total and free memory (in kilobytes), and the system load averages. Below is the equation we use in computing the score:

$$\text{Score} = \frac{\left[\left(\text{Average CPU Bogomips} \times \text{Number Of CPUs}\right) + \left(\text{Free Memory} \times 0.002\right)\right]}{\left[\dfrac{\left(\text{5 Minute Load Average}\right)^2}{\text{15 Minute Load Average}}\right]}$$

We decided not to use clock speed as a measurement of CPU performance because performance is much more dependent upon architecture than clock speed. Linux provides *bogomips* as an approximate measure of CPU performance. Although bogomips should not be used as an extremely accurate measurement of CPU performance (hence the name “bogus MIPS”), it is a reasonable method for performing approximate comparisons of processors, especially when the processors have the same architecture. We multiply the bogomips by the number of CPUs, because Symmetric Multiprocessing (SMP) systems are usually better equipped to execute multiple processes than single processor systems. The system memory is reported in bytes; we multiply the free system memory by 0.002 because we found by experimentation that doing so allows the memory to affect the score while not being the dominant factor. The load average factor of the score is somewhat more complex. The load average in UNIX and Linux systems is an exponentially-damped moving average of the number of processes currently running in addition to the number of processes in the run queue [19]. In other words, the load average is not a measure of processor utilization, but of the number of processes that are waiting for CPU time. Therefore, a lower average means that the system is more available, because there are fewer processes waiting in the run queue. Nodes with a low load average should receive a higher score than nodes with a high average. For this reason, we placed the load average factor in the denominator of the equation. Low load averages are typically less than or equal to one, so small loads will result in either no score change or a higher score because the numerator would be divided by a number less than one. Linux provides three measures of system load: the load for the past one minute, five minutes, and fifteen minutes. We found that the one minute load average fluctuates too rapidly to be useful in our scoring algorithm, so we were limited to using the five and fifteen minute load averages. We wanted to reward systems with a decreasing load and penalize systems with an increasing load. We accomplished this by dividing the five minute load by the fifteen minute load. If the five minute load is smaller than the fifteen minute load, the result will be a smaller number, which will increase the score because it is in the denominator of the equation. However, the fraction in the denominator of the score function presented a problem. If both the five minute and fifteen minute load average were 4.0, the resulting denominator in the score function would be a one, and this case would not penalize the score, even though both load averages were quite high. To solve this problem, we squared the five minute load average. In this case, the denominator would be sixteen divided by four, resulting in a four in the denominator and a lower score. Below are some examples to show how the load averages affect the score:

Average CPU Bogomips = 2000
Number of CPUs = 2
Free Memory = 300,000 kilobytes

Stable, Low Load Average

5 Minute Load Average = 1
15 Minute Load Average = 1

$$\text{Score} = \frac{\left[\left(2000 \times 2\right) + \left(300000 \times 0.002\right)\right]}{\left[\frac{\left(1\right)^2}{1}\right]} = 4600$$

Decreasing Load Average

5 Minute Load Average = 0.5
15 Minute Load Average = 1

$$\text{Score} = \frac{\left[\left(2000 \times 2\right) + \left(300000 \times 0.002\right)\right]}{\left[\frac{\left(0.5\right)^2}{1}\right]} = 10400$$

Increasing Load Average

5 Minute Load Average = 2
15 Minute Load Average = 1

$$\text{Score} = \frac{\left[\left(2000 \times 2\right) + \left(300000 \times 0.002\right)\right]}{\left[\frac{\left(2\right)^2}{1}\right]} = 1150$$

Stable High Load Average

5 Minute Load Average = 4
15 Minute Load Average = 4

$$\text{Score} = \frac{\left[\left(2000 \times 2\right) + \left(300000 \times 0.002\right)\right]}{\left[\frac{\left(4\right)^2}{4}\right]} = 1150$$

One final consideration in computing the score is that, because the load average factor is in the denominator and load averages can reach very low or high levels, the load average becomes the dominant factor in the score. During our testing we found that the node with a single 266 MHz Pentium II processor and 256 MB of system memory would often attain a higher score than the node with dual 1.6 GHz Pentium IV Xeon processors and 512 MB of system memory. The dual Xeon node is obviously a better choice when assigning agents or applications, but the Pentium II node could have a higher score because it was idle and had a lower load average. The actual problem was that there was a lower bound of zero on the score, but no upper bound. We realized that the most influential factor in the score should be the CPU performance. To ensure that this was the case, we limited the denominator of the equation so that it could never be smaller than 0.5.

Appendix H – Performance Evaluation Results

This appendix contains all of the results from our system performance evaluations. Additionally, we provide information about the configuration of the cluster we constructed. We did not have access to an actual cluster, so we had to assemble a small one ourselves using only our personal computers and networking equipment borrowed from GSFC. The tables below contain information about each node in the cluster and the network configuration.

Table 3: Node Configuration

	CPU	Cache (KB)	Clock (MHz)	Bogomips	RAM (MB)	Operating System	Kernel
Node 1	AMD Athlon 64 3000+	1024	1600	3162.11	512	Slackware Linux 10.0	2.6.7
Node 2	AMD Athlon XP 1800+	256	1500	3022.84	512	Slackware Linux 10.0	2.6.8.1
Node 3	Intel Pentium M	1024	1600	3170.30	512	Knoppix 3.4	2.6.6
Node 4	Intel Pentium III	256	866	1687.55	112	Knoppix 3.4	2.6.6
Node 5	Intel Pentium IV	512	2000	N/A	256	Windows XP Pro	N/A

Table 4: Network Configuration

Network Medium	100 Base-T Category 5 Ethernet
Network Address	192.168.0.0
Netmask	255.255.255.0
Broadcast Address	192.168.0.255
Network Device	Linksys EtherFast 10/100 Hub
	Model EFAH08W
	Version 3.0

During the system evaluation we conducted three trials for each of our four tests. The results from each trial appear in the tables below; the averages for each test can be found in the Results chapter. Tests 1 and 2 required only a single node; we used Node 1 for these tests. All times are in hh:mm:ss format.

Table 5: System Evaluation - Trial 1 Results

	Start Time	Stop Time	Run Time
Test 1	8:30:18	9:20:13	0:49:55
Test 2	19:05:06	19:57:13	0:52:07
Test 3	15:43:34	15:54:34	0:11:00
Test 4	18:14:16	18:25:34	0:11:18

Table 6: System Evaluation - Trial 2 Results

	Start Time	Stop Time	Run Time
Test 1	12:55:12	13:45:01	0:49:49
Test 2	21:01:01	21:53:10	0:52:09
Test 3	15:59:47	16:10:48	0:11:01
Test 4	18:46:48	18:58:26	0:11:38

Table 7: System Evaluation - Trial 3 Results

	Start Time	Stop Time	Run Time
Test 1	14:12:43	15:02:33	0:49:50
Test 2	22:13:42	23:05:58	0:52:16
Test 3	16:15:30	16:26:34	0:11:04
Test 4	19:57:22	20:08:39	0:11:17

Appendix I – IBM Paper

Autonomic Cluster Management System (ACMS)
An Introduction and Design Overview

James Baldassari
jdb@wpi.edu

Chris Kopec
chris@wpi.edu

Eric Leshay
ericl@wpi.edu

Worcester Polytechnic Institute

David Finkel
Project Advisor
Worcester Polytechnic Institute
dfinkel@wpi.edu

Walter F. Truszkowski
NASA Mentor
Goddard Space Flight Center
Walter.F.Truszkowski@nasa.gov

1.0 Introduction

Scientists and engineers at the National Aeronautics and Space Administration (NASA) often require significant computational power to accomplish their research objectives. The computational capabilities needed for the simulation and modeling of complex systems can be provided in several ways. A traditional High Performance Computing (HPC) approach to solving large computational problems has been the use of a single, powerful supercomputer. However, recent trends in HPC have been towards highly scalable, cost-effective solutions such as clusters and grid computing.

The NASA Goddard Space Flight Center (GSFC) in Greenbelt, MD was the birthplace of the first Beowulf cluster in 1994. Following the success of the first cluster system, GSFC has continued its research into the field of distributed computing. GSFC's recent focus has been developing autonomous, self-managing systems that would reduce the need for frequent human intervention. IBM's Autonomic Computing initiative correlates well with GSFC's research goals. The self-configuring, self-optimizing, self-healing, and self-protecting properties of an autonomic system could benefit many of

GSFC's current and future projects.

We are currently working at GSFC for ten weeks to complete our senior design project, an undergraduate degree requirement for Worcester Polytechnic Institute (WPI). Continuing GSFC's research into autonomic systems, we have designed and begun implementing an autonomic system for the management of distributed systems. Throughout this paper we will refer to our system as the Autonomic Cluster Management System (ACMS). The ACMS is a prototype for future endeavors at GSFC. The main goal of the ACMS is to display the four autonomic properties. The ACMS must plainly present and distinguish among the autonomic properties. The scientists at GSFC are interested in concrete examples of the autonomic properties; our system will allow them to judge if the application of these same properties is appropriate for their own work. The secondary goal is the development of a system that can manage a cluster. The outcome of our system will help engineers at GSFC to decide about the incorporation of autonomic principles in their own work.

2.0 System Design

We are developing the ACMS in Java on Linux using the Eclipse development environment. We needed an object-oriented language that had support for multi-threading and networking. One aspect of Java that we found beneficial was the Java Virtual Machine (JVM). The JVM allowed us to execute multiple instances of our agents without interfering with each other or affecting the underlying system. Additionally, the JVM allowed the majority of our code to be platform independent, so that the ACMS could be used seamlessly in a heterogeneous environment. After choosing the programming language, we began designing the system.

In the context of the ACMS we defined a cluster as a system comprised of one or more nodes connected by a network, and we defined a node as a single computer. We designed the ACMS to be a multi-agent system. Using a multi-agent system model allows us to distribute the management functions of a cluster. This design removes the constraint of having certain critical nodes on which the functionality of the entire system depends. The ACMS has no single point of failure. In fact, the entire system can operate with only a single node. From this high-level design we began planning the substructure of the ACMS. We designed the network communications and message-passing systems first; these are the foundation of our system because they enable all the agents in the cluster to communicate.

2.1 Communication System

Message passing is a necessary function in all distributed systems. The components of the system must be able to coordinate their actions for the system to be effective. We developed a custom Message Passing Interface (MPI) for all communications between agents in the ACMS. The interface supports two methods for sending and receiving messages. In peer-to-peer communication between two agents, messages are sent using the Transmission Control Protocol (TCP) and are encrypted using Secure Sockets Layer (SSL). All system commands and sensitive data are sent using this encrypted method, and in this way the system satisfies the self-protecting autonomic property. The second method in our MPI is used when a single message needs to be broadcast to multiple recipients. However, rather than broadcasting a message to every agent in the system, we use defined multicast groups so that messages are only received by the agents for which they are intended. One disadvantage of using multicast messages is that there is currently no way to encrypt them. Given this inherent weakness, no sensitive data about the system or commands for agents are sent using this method. Once we had designed the communications system, we began designing the agents that would use it.

2.2 Agent Design

The system consists of three types of agents; each has functionality exemplifying autonomic system properties. The three agent types we designed are called General Agents, Optimization Agents, and Configuration Agents. The ACMS is comprised of two Configuration Agents and one Optimization Agent per system, and two General Agents per node. The agents' goal is to manage a distributed application while maximizing its performance by implementing load-balancing techniques on the system.

2.2.1 Configuration Agent

The purpose of the Configuration Agent is to make the system self-configuring. The functionality of the Configuration Agent consists of maintaining a current list of all the agents in the system and making this information available to other agents. When an agent first comes on-line it broadcasts to the Configuration Agent's multicast address stating that it has joined the system. When this message is received, the Configuration Agent examines the table to ensure that the new agent is needed. For example, if there are already two Configuration Agents in the system and a third comes on-line, the system might become unstable. If the new agent does not belong in the system, a termination message is sent back to the agent. Periodically the Configuration Agent cycles through its list of agents and sends them messages to verify that each is still functioning properly.

If the Configuration Agent is unable to establish a connection with an agent, it can be assumed that the agent is no longer functioning correctly and will therefore be removed from the database. Otherwise, the agent responds with a list of information such as the address and port number of the agent's server, the agent type, and its system statistics (processor speed, number of processors, free memory, etc.). This list of information can be easily expanded to include requests for other information if necessary in the future. When the Configuration Agent receives this information it is updated in the table.

The system contains both a primary and a secondary Configuration Agent to support redundancy and the self-healing autonomic property. Ideally, both of the Configuration Agents would be on different nodes in the system so that if one node stops responding, there would be at least one Configuration Agent in the system. The reason for redundancy is that the database is stored locally by the agent in memory. Therefore, if the agent stopped functioning for any reason all the information within the database would be lost. To prevent this occurrence, the Vice President Configuration Agent synchronizes with the database of the primary agent. Only the primary agent performs the system configuration tasks. However, if the primary agent were to stop functioning, the Vice President Agent would be able to continue the primary agent's role. The Optimization Agent would detect that there is only one Configuration Agent functioning and recreate a second Configuration Agent.

2.2.2 Optimization Agent

The purpose of the Optimization Agent is to make the system self-optimizing. The role of the Optimization Agent within the system is first to contact the Configuration Agent for a current copy of the database. Once the database is received, the Optimization

Agent begins analysis of the database to ensure that there are the correct number and types of agents in the system. It verifies that there are exactly two Configuration Agents in the system, one Optimization Agent in the system, and two General Agents on each node in the system. If it finds this information to be incorrect, it sends commands to create or kill one or more agents, stabilizing the system. After performing a brief analysis of the system, it then begins observing the loads and statistics of each node, noting the lightly and heavily loaded systems. When the application needs to start a new process, the Optimization Agent finds a node that is not heavily loaded. It contacts a General Agent on the corresponding node and commands it to start the requested process. The Optimization Agent has the capability to move agents and processes from one node to another; allowing processing power to be utilized over multiple systems for a task, rather than having one system perform all of the processing. No redundancy is built in to the Optimization Agent because it does not store any important information in memory. If the agent were to stop responding, it could be easily recreated by the Configuration Agent. Once recreated, it would continue functioning properly with no loss of critical data. The only loss that occurs is any analysis of the table that the previous Optimization Agent had completed.

2.2.3 General Agent

The main functionality of each General Agent is to execute the commands sent by the other agent types. These commands are either to start or stop processes running on its system, to spawn a new agent, or to terminate itself. This method gives configuration and Optimization Agents the ability to start any type of agent on any node in the system, since all nodes contain at least one General Agent at all times. Redundancy, as with the

Configuration Agents, is built into the General Agents. The reason for redundancy in this case is not to preserve data, but rather to ensure that a node will remain part of the system. If there were only one General Agent on a node, and that agent stopped responding, the entire node would be disconnected from the system. However, if there are two General Agents per node, and one fails, the remaining agent can recreate the failed General Agent. Once again this behavior satisfies the self-healing autonomic property. The self-healing property of the General Agents reduces the chance that a node will be removed from the system due to agent failure, thus requiring less maintenance by human intervention.

3.0 Concluding Remarks

The ACMS is a unique system that has many potential applications. Using its autonomic multi-agent framework as a foundation, the system can be easily extended to perform a diverse set of management tasks in a heterogeneous environment. Although the current focus of the system is ground-based HPC facilities, a future application of this technology might be an autonomic management system for a cluster of unmanned spacecraft. It is our hope that the ACMS will advance GSFC's research efforts in autonomous and autonomic systems.

Appendix J – JavaDoc

wasp.data
Class SysStatsReader

java.lang.Object
  wasp.data.SysStatsReader
Direct Known Subclasses:
LinuxSysStatsReader, WindowsSysStatsReader

---

public abstract class SysStatsReader
extends java.lang.Object
The SysStatsReader class is used in writing classes that retrieve operating system specific information, such as CPU, memory, and load information.
Author:
jbaldassari
See Also:
LinuxSysStatsReader, WindowsSysStatsReader

Field Summary	
(package private) Tuple	t

Constructor Summary
SysStatsReader()

Method Summary	
protected abstract java.util.Vector	getCPUStats() Retrieves information about the node's CPU(s)
protected abstract java.util.Vector	getLoadStats() Retrieves information about the node's current load
protected abstract java.util.Vector	getMemStats() Retrieves information about the node's system memory
void	populateStats(Tuple t) Updates a Tuple object with the current system information

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

t

Tuple t

Constructor Detail

SysStatsReader

public SysStatsReader()

Method Detail

populateStats

public void populateStats(Tuple t)
Updates a Tuple object with the current system information
Parameters:
t - A Tuple object

---

getCPUStats

protected abstract java.util.Vector getCPUStats()
Retrieves information about the node's CPU(s)
Returns:
A Vector containing the number of CPUs, average CPU speed, and average CPU Bogomips.

---

getMemStats

protected abstract java.util.Vector getMemStats()
Retrieves information about the node's system memory
Returns:
A Vector containing the current total and free memory.

---

getLoadStats

protected abstract java.util.Vector getLoadStats()
Retrieves information about the node's current load
Returns:
A Vector containing the current load averages.

---

wasp.messaging
Interface Server
All Known Implementing Classes:
BroadcastServer, SSLServer

public interface Server
This is a common interface for all servers.
Author:
jbaldassari

Method Summary	
Agent	getAgent() Gets the agent that is running the server
int	getPort() Gets the server port.
void	start() Starts the server.
void	stop() Stops the server.

Method Detail

start

public void start()
Starts the server.

stop

public void stop()
Stops the server.

getPort

public int getPort()
Gets the server port.
Returns:
The port number that the server is listening on.

getAgent

public Agent getAgent()
Gets the agent that is running the server
Returns:

The agent that instantiated the server object.

---

wasp.agent
Class GeneralAgent

java.lang.Object
  wasp.agent.Agent
    wasp.agent.GeneralAgent

---

public class GeneralAgent
extends Agent
GeneralAgent is a type of Agent that is responsible for running tasks and reporting system information to the Configuration Agent.
Author:
ckopec

Field Summary	
private java.util.Vector	activeJobs
private java.lang.String	lastKnownConfigAddr
private int	lastKnownConfigPort
static java.lang.String	multicastGroup
static int	multicastPort
private Tuple	tp

Fields inherited from class wasp.agent.Agent
broadcastServer, sslServer

Constructor Summary	
private	GeneralAgent() Constructor calls Agent constructor.

Method Summary	
java.lang.String	dequeueJob(Job j) Dequeues the first instance of a Job
void	enqueueJob(Job j) Enqueues a Job in this agent's Tuple
java.lang.String	getLastKnownConfigAddr() Returns the last known address of the President Configuration Agent
int	getLastKnownConfigPort() last known port of the President Configuration Agent
java.lang.String	getMulticastGroup() Returns the General Agents multicast group address
int	getMulticastPort() Returns the General Agents multicast group port
Tuple	getTuple() Gets this agent's tuple
protected void	jobHandler(MessageWrapper mw) The method used when a job message is recieved.
static void	main(java.lang.String[] args) If executed, creates a new GeneralAgent in a new Java VM
protected void	systemHandler(MessageWrapper mw) Recieved by the Configuration Agent when checking to see if agents are alive and collecting their system statistics.
java.lang.String	toString()

Methods inherited from class wasp.agent.Agent
agentHandler, databaseHandler, dequeueMessage, destroy, getSslServer, handleMessages, println, queueMessage, receiveMessage, recievedSystemHandler, sendBroadcastMessage, sendMessage, sizeMsgQueue, spawnAgent, startDebugConsole

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, wait, wait, wait

Field Detail

multicastGroup

public static final java.lang.String multicastGroup
See Also:

Constant Field Values

---

multicastPort

public static final int multicastPort
See Also:
Constant Field Values

---

tp

private Tuple tp

---

lastKnownConfigAddr

private java.lang.String lastKnownConfigAddr

---

lastKnownConfigPort

private int lastKnownConfigPort

---

activeJobs

private java.util.Vector activeJobs

Constructor Detail

GeneralAgent

private GeneralAgent()
Constructor calls Agent constructor. It broadcasts and waits 5 seconds then times out and creates a new Configuration Agent on the node.

Method Detail

main

public static void main(java.lang.String[] args)
If executed, creates a new GeneralAgent in a new Java VM
Parameters:
args -

---

getMulticastGroup

public java.lang.String getMulticastGroup()
Returns the General Agents multicast group address
Specified by:
getMulticastGroup in class Agent
Returns:
multicast address
See Also:

Agent.getMulticastGroup()

---

getMulticastPort

public int getMulticastPort()
Returns the General Agents multicast group port
Specified by:
getMulticastPort in class Agent
Returns:
multicast port
See Also:
Agent.getMulticastPort()

---

toString

public java.lang.String toString()
Returns:
The name of the Agent

---

systemHandler

protected void systemHandler(MessageWrapper mw)
Recieved by the Configuration Agent when checking to see if agents are alive and collecting their system statistics. The General Agent makes a new tuple which it fills and sends back to the Configuration Agent with the correct information.
Overrides:
systemHandler in class Agent
Parameters:
mw - MessageWrapper of the message that was just recieved by the current agent.
See Also:
wasp.agent.Agent#SystemHandler(wasp.messaging.messages.MessageWrapper)

---

jobHandler

protected void jobHandler(MessageWrapper mw)
The method used when a job message is recieved. Will either start a new job on the agents node or stop a running job on the node.
Overrides:
jobHandler in class Agent
Parameters:
mw - MessageWrapper of the message that was just recieved by the current agent.
See Also:
wasp.agent.Agent#JobHandler(wasp.messaging.messages.MessageWrapper)

---

getTuple

public Tuple getTuple()

Gets this agent's tuple
Returns:
the agent's tuple

---

getLastKnownConfigAddr

public java.lang.String getLastKnownConfigAddr()
Returns the last known address of the President Configuration Agent
Returns:
Returns the last known address of the President Configuration Agent

---

enqueueJob

public void enqueueJob(Job j)
Enqueues a Job in this agent's Tuple
Parameters:
j - The Job to enqueue

---

dequeueJob

public java.lang.String dequeueJob(Job j)
Dequeues the first instance of a Job
Parameters:
j - The name of the Job to dequeue
Returns:
The name of the job that was dequeued

---

getLastKnownConfigPort

public int getLastKnownConfigPort()
last known port of the President Configuration Agent
Returns:
Returns the last known port of the President Configuration Agent

---

References

1. United States. National Air and Space Administration (NASA). About Goddard. 3 Apr. 2004 <http://www.gsfc.nasa.gov/about_mission.html>.

2. IBM Research Communications. "Glossary." Autonomic Computing. 23 Feb. 2001. IBM Research. 3 Apr. 2004 <http://www.research.ibm.com/autonomic/glossary.html>.

3. Casanova, Henri. "Distributed Computing Research Issues in Grid Computing." ACM SIGACT News 33.3 (2002): 50-70.

4. Minar, Nelson. "Distributed Systems Topologies: Part 1." 14 Dec. 2001. O'Reilly Network. 13 Sept. 2004. <http://www.openp2p.com/pub/a/p2p/2001/12/14/topologies_one.html>.

5. Merkey, Phil. "Beowulf History." Beowulf.org. 21 Apr. 2004. <http://www.beowulf.org/overview/history.html>.

6. Prasad, Dr. K. V. Principles of Digital Communication Systems and Computer Networks. Hingham, MA: Charles River Media, 2003. Ch. 20.4

7. Prasad, Dr. K. V. Principles of Digital Communication Systems and Computer Networks. Hingham, MA: Charles River Media, 2003. Ch. 20.5

8. Garms, Jess and Daniel Soerfield. Professional Java Security. Berkeley, CA: Apress, 2003. Ch. 9

9. How SSL Works. 1999. Netscape. 20 Aug. 2004. <http://developer.netscape.com/tech/security/ssl/howitworks.html>.

10. Yang, Kung, et al. "Towards efficient resource on-demand in Grid Computing." ACM SIGOPS Operating Systems Review 37.2 (2003): 37-43.

11. Rajkumar Buyya. "Grid Computing Info Centre: Frequently Asked Questions (FAQ)." GRID Infoware. 29 Apr. 2004. <http://www.gridcomputing.com/gridfaq.html>.

12. Schoeman, Martha and Elsabe Cloete. "Architectural Components for the Efficient Design of Mobile Agent Systems." Proceedings of the 2003 ACM Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on Enablement through Technology. (2003): 48-58. University of South Africa. Pretoria, Gauteng, South Africa.

13. Farley, Jim. Java Distributed Computing. Sebastopol, CA: O'Reilly & Associates, Inc., 1998. Ch. 1.2.

14. IBM Research Communications. "Autonomic Computing: Overview." Autonomic Computing. 23 Feb. 2001. IBM Research. 21 Apr. 2004. <http://www.research.ibm.com/autonomic/overview/>.

15. Stojanovic, L., et al. "The Role of Ontologies in Autonomic Computing Systems." IBM Systems Journal. 21 Jul. 2004. IBM Research. 22 Sept. 2004. <http://www.research.ibm.com/journal/sj/433/stojanovic.html>.

16. IBM Corporation. "Tivoli Software." Autonomic Computing. 2001. IBM Autonomic Computing Products and Services. 29 Sept. 2004. <http://www-306.ibm.com/autonomic/tivoli.shtml>.

17. IBM Corporation. "About IBM Autonomic Computing: Autonomic Deployment Model." Autonomic Computing. 25 Jun. 2003. 25 Aug. 2004. <http://www-306.ibm.com/autonomic/levels.shtml>.

18. IBM Corporation. "The Tivoli Software Implementation of Autonomic Computing Principles." 2002. IBM Corporation Software Group. 30 Aug. 2004. <http://www-306.ibm.com/autonomic/pdfs/br-autonomic-guide.pdf>.

19. Gunther, Dr. Neil. "UNIX Load Average Part 1: How It Works." 26 Feb. 2003. TeamQuest Corporation. 5 Oct. 2004. <http://www.teamquest.com/resources/gunther/ldavg1.shtml>.

20. Purdy, Doug. "Exploring the Factor Design Pattern." Feb. 2002. Microsoft Developer Network. 3 Oct. 2004. <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/factopattern.asp>.

21. Grand, Mark. Patterns in Java: A Catalogue of Reusable Design Patterns Illustrated with UML. 2nd ed. Vol 1. John Wiley & Sons, 2002.

22. Feibel, Werner. The Network Press Encyclopedia of Networking. Sybex, 2000.

23. Kindlein, Armin. "Grasshopper 2: The Agent Platform." 2003. IKV++ Technologies AG. 16 Aug. 2004. <http://www.grasshopper.de>.

24. Eclipse Project PMC. "Eclipse Project." 2004. 14 Aug. 2004. <http://www.eclipse.org>.

25. CollabNet, Inc. "Concurrent Versions System: The Open Standard for Version Control." 2003. 24 Aug. 2004. <http://www.cvshome.org>.

26. The Madkit Project. “Madkit.” Nov. 2002. 3 Apr. 2004. <http://www.madkit.org>.

27. IBM Corporation. “Log and Trace Analyzer for Autonomic Computing.” 8 Apr. 2003. 3 Apr. 2004. <http://www.alphaworks.ibm.com/tech/logandtrace>.