Autonomic Cluster Management System (ACMS): A Demonstration of Autonomic Principles at Work

James D. Baldassari[1], Christopher L. Kopec[1], Eric S. Leshay[1], Walt Truszkowski[2], David Finkel[1]

[1]*Worcester Polytechnic Institute*
Computer Science Department
Worcester, MA 01609
{jdb, chris, ericl, dfinkel}@wpi.edu

[2]*NASA Goddard Space Flight Center*
Advanced Architectures & Automation Branch
Greenbelt, MD 20771
walt.truszkowski@nasa.gov

Abstract

Cluster computing, whereby a large number of simple processors or nodes are combined together to apparently function as a single powerful computer, has emerged as a research area in its own right. The approach offers a relatively inexpensive means of achieving significant computational capabilities for high-performance computing applications, while simultaneously affording the ability to increase that capability simply by adding more (inexpensive) processors. However, the task of manually managing and configuring a cluster quickly becomes impossible as the cluster grows in size. Autonomic computing is a relatively new approach to managing complex systems that can potentially solve many of the problems inherent in cluster management. We describe the development of a prototype Automatic Cluster Management System (ACMS) that exploits autonomic properties in automating cluster management.

1. Introduction

NASA's Goddard Space Flight Center (GSFC) conducts research and development in a wide range of topics and areas in the field of information technology. These include research in areas such as advanced knowledge management, data/information visualization, semantic-web technologies, sensor-web technologies and grid computing, amongst others.

The primary aim of this research is to support NASA missions and projects. This includes applications that involve the collection and management of extremely large datasets and the use of very complex models for manipulating and interpreting science data collected by various NASA instruments and missions.

The successful completion of GSFC's science data and information processing objectives often entails the solution of large distributed computational problems, such as the management and simulation of complex Earth-modeling systems. Many of these problems are so computationally demanding that some form of High Performance Computing (HPC) is required to solve them.

2. HPC and Cluster Computing

Traditionally, at NASA and elsewhere, Massively Parallel Processing (MPP) computer systems have been used to meet high performance computing requirements. MPP computers may contain hundreds or thousands of processors within a single computer system. Typically, these types of computer systems are extremely expressive and upgrading typically requires a complete rebuild of the system. They are, however, relatively simple to manage, and they certainly perform very well. A recent trend in high performance computing has been to overcome the cost and scalability issues associated with MPP systems by replacing these with a different type of HPC system called a cluster.

A cluster is composed of a collection of inexpensive individual computers, referred to as 'nodes', that are connected together via a network and configured so as to appear to the user as a single powerful computer.

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, however, 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 highly 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 consequently cannot coordinate its activities with the other nodes. A severed network connection or an otherwise unresponsive node could cripple the cluster if there is no mechanism to recover from failures. Finally, unauthorized access to the cluster is a constant concern for system administrators, due to the significant number of potential intrusion points.

3. Autonomic Computing and Cluster Management

Autonomic computing represents a relatively new approach to the management of complex systems that offers the potential of solving many of the problems inherent in cluster management.

By definition, an autonomic system is one that, at a minimum, exhibits the properties of being self-configuring, self-optimizing, self-healing, and self-protecting [2]. Like the autonomic nervous system of a human, an autonomic program should react to events without conscious thought, but rather as a reflex [3]. Using this set of autonomic properties as a guide, we have designed and implemented a prototype Autonomic Cluster Management System (ACMS).

In the remainder of this paper, we will describe the autonomic aspects of this system, address the architectural issues of the ACMS, and describe our results from the evaluation of the prototype.

3.1 ACMS Prototype

The ACMS is a mobile agent system composed of a number of agent processes communicating across a network of nodes.

The system consists of three types of agents, each with functionality implementing autonomic system properties. The three agent types are:

- *General Agents*,
- *Optimization Agents*, and
- *Configuration Agents*.

Specifically, the ACMS is comprised of two Configuration Agents and one Optimization Agent per implementation, and two General Agents per node.

Each agent is designed to be specific purpose, and to perform a particular task. Together the community of agents collaborates to achieve a common goal, specifically providing autonomic management of a cluster, while simultaneously maximizing performance by implementing load-balancing techniques on the system.

Figure 1 shows the architecture of the prototype system.

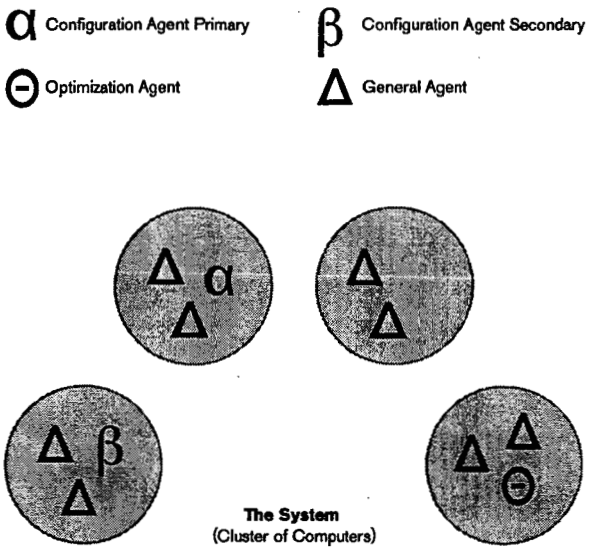

Figure 1. Architecture of the Prototype

3.2 Autonomic properties

A system is said to be autonomic if it incorporates the four key autonomic properties: self-configuring, self-healing, self-optimizing and self-protecting. Like the autonomic nervous system of the human body, an autonomic system should react to events as a reflex, without conscious thought. We examine now how a cluster management system makes use of the key autonomic properties.

Self-Configuring

Updating a cluster's configuration and parameters is a time consuming task that plagues system administrators whenever a change is made. Systems that can adapt to system changes without human intervention are self-configuring and allow administrators to focus their attention on more important tasks.

In a cluster environment these changes often deal with topology, such as a node failure, the addition of a new node, or the relocation of a node. As topologies grow larger and modification becomes more frequent, the need for a system that can recognize change and automatically adapt accordingly is essential. This is one goal that the ACMS achieves in its pursuit of autonomous operation.

The ACMS assigns the task of recognizing changes, analyzing the configuration, and generating new plans to execute to the Configuration Agent. In order for the Configuration Agent to recognize change it must first be aware of the current environment. This is accomplished through polling, in which a broadcast message is sent to

determine where all other nodes in the system are located. This is performed at regular intervals to discover new or removed nodes.

The Configuration Agent maintains this information in a database for use by other agents in the ACMS. By polling the cluster and comparing it with previously stored configurations, the system is able to make informed decisions about its current capacity and carry out various scenarios for recovery or optimization.

Self-Healing

In today's world of on-demand computing, the terms *availability* and *quality of service* have taken on new meanings.

It is no longer sufficient for a program to be defect-free; in addition, it must be operational at all hours and running at peak performance for months, or even years, at a time. In practice, this necessitates hiring staff to monitor the service and fix any problems with as little disruption to the user as possible. If the system could recover from faults and errors on its own, then the need for monitors and around-the-clock staff would be eliminated, and resources could be applied elsewhere.

This is the idea behind self-healing, that is, developing a system that can recover from faults and resume service seamlessly without human interaction.

The ACMS achieves self-healing through the use of redundancy and monitors. The Configuration Agent takes on the role of the monitor when it is polling nodes. If it determines that a node and the agents who reside there do not respond, it takes action to remedy the problem. The ACMS incorporates the ability to start new agents in the architecture of each agent so that any agent may spawn any other type of agent.

The Configuration Agent determines the type of agent that must be created and instructs another agent on the network to spawn that agent. Armed with this capability, the system can be returned to full operational status.

One difficulty is spawning agents on a node after all agents on the node have been failed. To handle this problem the ACMS maintains a minimum of two (general) agents on every node so that a backup agent is available if any one agent fails. Currently the ACMS has a single fault tolerance, but this can be expanded by instantiating additional redundant agents on each node.

Self-Optimizing

The performance of a system or an application depends on its goal and current configuration. When the configuration changes or a new application is introduced, a system often requires fine-tuning in order to get the best performance. This is another task system administrators are faced with when change is introduced to a networked environment such as a cluster.

An autonomic system is self-optimizing and once again frees the system administrator from this burden. In the context of clusters, the key optimization issue is load balancing. Load balancing is deciding where to assign new processes so that the resources in the cluster are efficiently used to provide the best performance.

Through the use of autonomic properties, the ACMS is able to accomplish optimization in an environment where changes are frequent. The task of self-optimization is performed by the Optimization Agent. This agent uses the information gathered by the Configuration Agent to determine load statistics for each node. The ACMS gathers information used for load balancing before a new process is scheduled; it does not move processes once they have begun. Each process is assigned to a machine based on a predetermined availability threshold generated through a performance evaluation function. By making informed decisions based on knowledge of the system's current state, the Optimization Agent is able to efficiently distribute user processes as soon as they are scheduled, so there is no need to use preemptive scheduling techniques.

The ACMS also performs load balancing on the two Configuration Agents and the Optimization Agent. ACMS is a mobile agent system and has the power to relocate any agent from one node to another. The system favors placing the Configuration Agents and the Optimization Agent on nodes with a low load.

In addition, ACMS will attempt to distribute the three agents among separate nodes to provide better fault tolerance and recovery of key components.

Self-Protecting

An autonomic system must provide security in order to prevent attacks and protect private information.

The system must take a proactive approach and be self-protecting. It can recognize intrusion attempts and prevent them by itself. This avoids the unnecessary t the loss of time inherent in current systems, where an intrusion attempt must first be found and then patched.

If the system alone can handle the encounter, the intrusion can be stopped immediately and the damage contained. In this way, the ACMS offers greater security and confidence than traditional approaches.

Redundancy and encryption are used to realize self-protection. Redundancy is used to protect the system from failure so that it can use its properties of self-healing to recover. System redundancy includes

redundant agents as well as extra copies of the database detailing the topology of the cluster.

The ACMS uses 2048-bit RSA encryption to prevent rogue agents from joining, or communicating with, the system. Except for broadcast messages, all communication is conducted over Secure Sockets Layer (SSL). Any program that attempts to instruct an agent to take an action or attempt to join the system will have to communicate with the Configuration Agent over SSL with the correct certificate. Without the required certificates and keys the (rogue) agent will be refused admittance and be unable to decrypt communication across the cluster.

Another benefit of encrypted communication is the ability to build a cluster as part of an existing network; the ACMS will remain secure and function while running alongside nodes that are not part of the ACMS topology.

4. Agent Design

4.1 Configuration Agent

The purpose of the Configuration Agent is to provide the ability to self-configure within the system.

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 upon request.

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 new agent is not needed. If the new agent does not belong in the system, a termination message is sent back to the agent.

In addition, 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 restarted. Otherwise, the agent responds with a list of information such as the address and port number of the agent's location, 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 updates the database.

The system contains both a primary and a secondary Configuration Agent to support redundancy and the self-healing autonomic property. Ideally, the two 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 of agents is stored locally by the agent in memory. Therefore, if the agent stopped functioning for any reason all the information would be lost. The secondary configuration agent synchronizes with the primary configuration agent and maintains a copy of the database. Only the primary configuration agent performs the system configuration tasks. However, if the primary agent were to stop functioning, the secondary agent would be able to continue in the role of the primary agent.

In this case the Optimization Agent would detect that there were only one Configuration Agent functioning and recreate a second Configuration Agent.

4.2 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.

If it finds the configuration 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 nodes.

When an application needs to start a new process, the Optimization Agent searches for the first node that is not heavily loaded. It contacts a General Agent on that 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 for load balancing of processes.

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.

4.3 General Agent

The main function of each General Agent is to execute the commands sent to it by the other agents.

These commands are either to start or stop processes running on its node, to spawn a new agent, or to terminate itself. Termination gives the configuration and optimization agents the ability to start any type of agent on any node in the system. 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 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, reducing the need for human maintenance and intervention.

5. System Topology

Our prototype system utilizes a hybrid centralized and decentralized design, as shown in Figure 2.

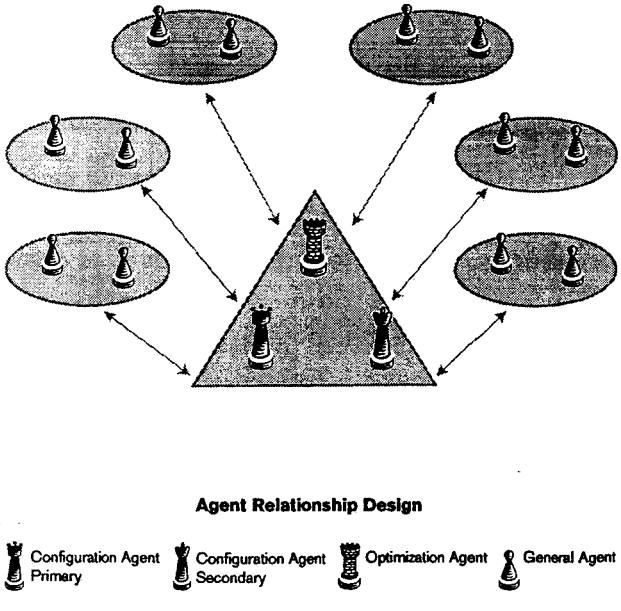

Figure 2. System Topology

The system acts in a centralized manner, with all information being contained in the Primary Configuration Agent.

The database which is maintained contains information regarding all living agents within the system. Since all information is held 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 can be insecure because nodes can join at any point and start sending data that may be incorrect. However, all message transfer in our prototype is encrypted. Therefore, any node that joins the system would not be able to communicate with other agents unless the correct certificates were used.

Although the system's decentralized topology creates some security challenges, this type of topology facilitates scalability. Any new nodes with the correct certificates can join the system and immediately begin communicating with other agents.

5.1 Network Communication

The communications system is important in any distributed or clustered system, but its role in an autonomic system is of even greater significance.

In addition to providing a mechanism for transferring data across a network, our prototype also has 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 unauthorized 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. 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) used by secure protocols.

We decided that the messages that needed to be broadcast to the entire system would not contain any sensitive information, so they could be transmitted unencrypted.

Java has built-in support for SSL, a popular and trusted method for transferring encrypted data across networks. We decided that SSL was sufficient to meet the needs for our secure peer-to-peer communication because it is capable of using strong 2048-bit encryption. 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

Table 1: 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%**

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 certain group of agents, and a broadcast message will never need to be sent to all agents in the system. Since broadcasting to the entire system is not necessary, we decided instead to use multicasting. 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 and port.

6. Evaluation of the Prototype

The ACMS prototype was evaluated using two types of test.

The first type of test was designed to verify the operational capabilities of the ACMS. For this operational test we developed seventeen operational scenarios that the ACMS might encounter. For each scenario, we specified an initial system state, an event that occurs, and the expected result.

The following are two examples chosen from the seventeen scenarios:

Scenario 4:
Unexpected Termination of a General Agent
Initial state: The ACMS is active.
Event: A general agent on one of the nodes is terminated.
Expected result: A new general agent is created on the same node (self-healing).

Scenario 10:
Introduction of a Third Configuration Agent
Initial state: The ACMS is active and contains two configuration agents.
Event: A third configuration agent attempts to join the system.
Expected result: The ACMS terminates the third configuration agent, rather than allowing it to join the system (self-protecting).

Through careful evaluation of these operational scenarios we were able to validate our implementation of the autonomic properties, verify that the ACMS functioned as we expected, and ensure that the ACMS was able to handle unexpected events.

The second type of test was designed to measure the performance and overhead of the ACMS. We created a simple distributed application that we executed in different configurations on a small cluster managed by the ACMS. This distributed application calculates all the prime numbers between one and one million. Although the application is simple, its operation is characteristic of many distributed applications in the scientific and academic communities.

Most of these applications are very compute-intensive, and the time required to transfer the data set is small in comparison to the time required to analyze the data. Our distributed application used a client-server model. The server partitioned the one million numbers into discrete data sets of ten thousand numbers each. The clients connected to the server, received a data set, searched the set for prime numbers, and reported their results back to the server. We executed our distributed application in four configurations:

(1) 1 node without the ACMS
(2) 1 node with the ACMS
(3) 5 nodes without the ACMS
(4) 5 nodes with the ACMS

In each configuration we measured the total amount of time required to check the entire range of one million numbers. We executed the application three times per configuration and averaged the run times.

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.

The average times for each test are given in Table 1. These results, although they should not be overstated as they are applied to a very limited sample, nevertheless 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.

Since 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 that 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.

7. Conclusions

We have described an experimental project to develop a prototype Autonomic Cluster Management System, suitable for use in cluster-based high-performance computing.

The general purpose of the project was to gain some insight into what the four key autonomic properties would involve in the implementation of such a management system. We have successfully implemented these.

The prototype has been evaluated and demonstrated to be scaleable. While the sample space for our experimentation was small, we are encouraged by seeing a decrease in overhead for the ACMS as the cluster size grows, with a simultaneous (almost 100%) expansion in processing power.

Future extensions of this work will include adding "learning", and examining its effect on autonomicity.

Acknowledgement

We wish to acknowledge the constructive criticism given an earlier version of this paper by Patricia Rago and her colleagues at IBM.

References

[1] J. Garms and D. Soerfield, *Professional Java Security*, APress, Berkeley, CA., 2003.

[2] IBM Corp., "About IBM Autonomic Computing", IBM Autonomic Computing; 4 January 2005, http://www-03.ibm.com/autonomic/about.shtml.

[3] IBM Research Communications, "Glossary", http://www.research.ibm.com/autonomic/glossary.html, 23 February 2001