Towards an Autonomic Cluster Management System (ACMS) with Reflex Autonomicity

Walt Truszkowski[1], Mike Hinchey[1] and Roy Sterritt[2]

[1]*NASA Goddard Space Flight Center*
Information Systems Division
Greenbelt, MD, USA
walt.truszkowski@nasa.gov
michael.g.hinchey@nasa.gov

[2]*University of Ulster*
School of Computing and Mathematics
Jordanstown Campus
Northern Ireland
r.sterritt@ulster.ac.uk

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 providing a fault-tolerant environment and achieving significant computational capabilities for high-performance computing applications. However, the task of manually managing and configuring a cluster quickly becomes daunting as the cluster grows in size. Autonomic computing, with its vision to provide self-management, can potentially solve many of the problems inherent in cluster management. We describe the development of a prototype Autonomic Cluster Management System (ACMS) that exploits autonomic properties in automating cluster management and its evolution to include reflex reactions via pulse monitoring.

1. Introduction

NASA 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 areas such as advanced knowledge management, data/information visualization, semantic-web and 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 essential [3].

In an increasingly cost-focused environment, less expensive options are now needed. We report on continuing work on providing HPC through a cluster, its requirement to be self-managing, and the ongoing efforts to achieve this autonomicity.

2. Cluster Computing to Provide HPC

Traditionally, 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, upgrading such systems 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 research has been to find new approaches to overcome the cost and scalability issues associated with MPP systems, such as clusters and grids.

The concept of a cluster is to take two or more computers and organize them to work together to provide higher availability, reliability and scalability. When failure occurs, resources can be redirected and the workload can be redistributed. As such, 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. However, the approach opens up an arena of different complexity challenges in terms of management, configuration, and security. Rather than administering a single computer, management and configuration tasks on a cluster must be performed on every node. As a consequence, Autonomic Computing is particularly relevant.

3. Autonomic Cluster Management

Autonomic computing 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 self-managing properties of being self-configuring, self-optimizing, self-healing, and self-protecting, through self-awareness and environment awareness [1]. Like the autonomic nervous system of the human body, an autonomic system should react to events without conscious thought, but rather as a reflex [2]. Using this initial set of autonomic properties as a guide, we have designed and implemented a prototype Autonomic Cluster Management System (ACMS).

3.1 Autonomic properties

Obviously the main autonomic properties required in a cluster system to provide continual load balancing are self-configuration and self-optimization. Self-healing is critical to ensure dependability, continuity of availability, and reliability. Self-protection is utilized to safeguard the cluster system from misuse; this is especially vital when one considers that nodes may be end-user machines distributed throughout the organization [3].

Self- and environmental awareness are achieved by virtue of the properties of agents, *viz.* being autonomous, reactive, goal-driven and temporally continuous [7].

3.2 The 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, namely *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. 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 illustrates the architecture of the prototype system.

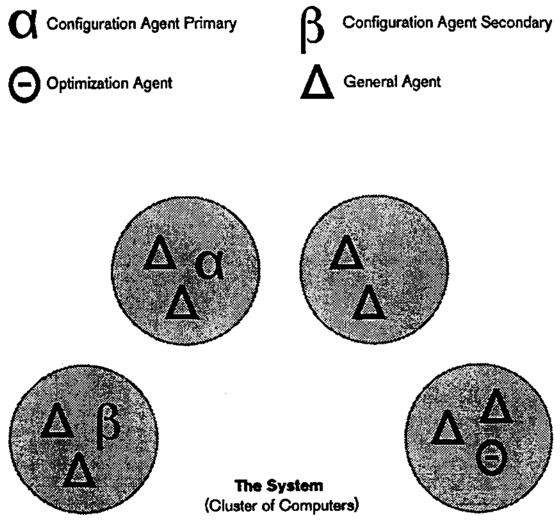

Figure 1. Architecture of the Prototype

3.2.1 Configuration Agent

The purpose of the Configuration Agent (CA) is to provide the ability to self-configure within the system, providing self-healing and self-protecting facilities.

Its functionality 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 CA's multicast address stating that it has joined the system. The CA examines the table to ensure that the new agent is needed (e.g., an unneeded third CA attempting to come on-line, in which case a termination message is sent back to the agent).

In addition, the CA cycles through the database of agents asking each if it is still functioning properly. If the CA is incapable of establishing a connection with an agent, it is assumed that the agent is no longer functioning correctly and will be restarted. Otherwise, the agent responds with a list of information such as the address and port number, 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, in the future, if necessary. When the CA receives this information it updates the database.

The system contains both a primary and a secondary CA to support redundancy and the self-healing autonomic property. Ideally, the two CAs would be on different nodes in the system so that if one node stops responding there would be at least one CA still functioning. The reason for redundancy is that the database is stored in memory locally by the CA. Therefore, if the CA stops functioning for any reason, all the information would be lost. The secondary CA synchronizes with the primary one and maintains a copy of the database. Only the primary CA performs system configuration tasks. However, if the primary CA were to stop functioning, the secondary one would be able to continue in the role of the primary CA. In this case the Optimization Agent would detect that there was only one CA functioning and recreate a second CA.

3.2.2 Optimization Agent

The purpose of the Optimization Agent (OA) is to make the system self-optimizing. The role of the OA within the system is first to contact the CA for a current copy of the database. Once received, the OA 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 OA 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 OA 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 OA because it does not store any important information. If the agent were to stop responding, the CA could easily recreate it. The only loss that occurs is any analysis of the table that the previous OA had completed.

3.2.3 General Agent

The main function of each General Agent (GA) 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 CA and OA the ability to start any type of agent on any node in the system. Redundancy is built into the GAs.

The reason is not to preserve data, but to ensure that a node will remain part of the system. If there were only one GA on a node, and that agent stopped responding, the entire node would be disconnected from the system. However, if there are two GAs per node and one fails, the remaining agent can recreate the failed GA. Once again this behavior satisfies the self-healing autonomic property, reducing the need for human maintenance and intervention.

3.2.4 System Topology

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

The system acts in a centralized manner, all information being contained in the Primary CA.

The database which is maintained contains information regarding all extant agents in 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 CA. 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.

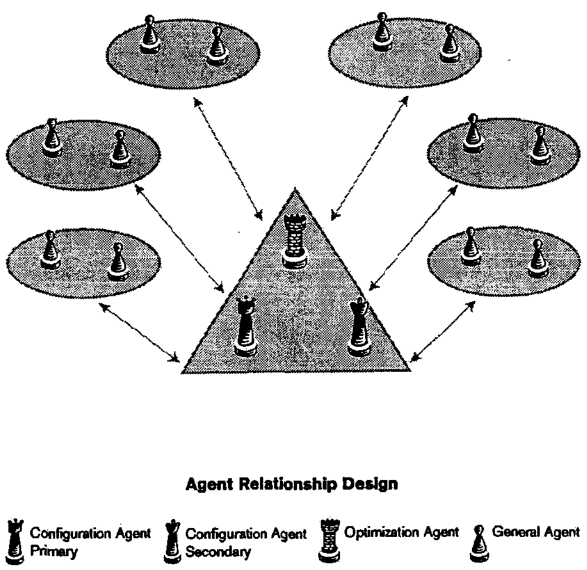
Agent Relationship Design
Configuration Agent Primary
Configuration Agent Secondary
Optimization Agent
General Agent

Figure 2. System Topology

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.

3.2.5 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 [6]. 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 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.

3.2.6 Evaluation of the Prototype

The ACMS prototype has been evaluated both from the autonomic management capabilities (seventeen operational scenarios - such as hanging of general agent) and the performance overhead of the ACMS [3]. While achieving the benefits of a cluster, the overhead associated with running the ACMS was 5% in the worst-case (all management agents running on the same node). As the number of nodes increased the overhead decreased to less than 0.75%.

4. Future Work

Part of the motivation behind this work is to investigate different autonomic approaches and mechanisms and to test these against each other.

It is planned that in the next version of the ACMS (Figure 3) the agents function will be extended to include a heartbeat (sending of a periodic 'I am alive' signal) [8]. This would allow a change in the procedure for fault tolerance, as opposed to the current approach where the configuration agent polls all the agents.

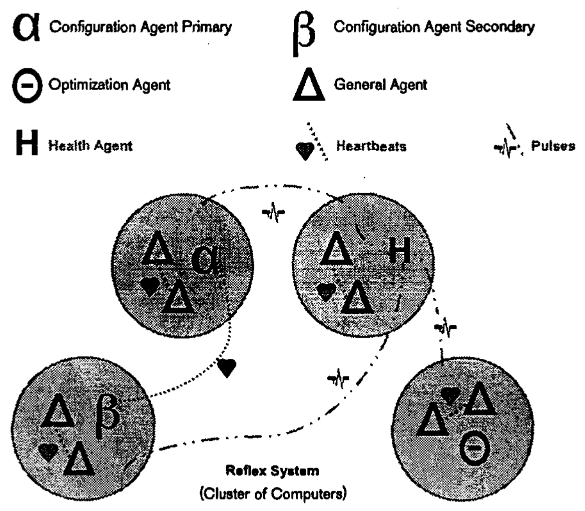

Figure 3. Extending the Architecture

This will also allow the provision of localized fail-over on the node; i.e., instead of the current approach of the CA noticing an agent has failed (through polling) on a remote host and instructing the switch over to the secondary agent on that remote host, this can occur locally via the secondary agent monitoring the heartbeat from the primary agent and thus providing a tighter and situated reflex reaction upon failure.

The primary and secondary CAs will also utilize the same mechanism with heartbeats between them (typically on separate hosts to increase fault tolerance).

Self-healing is currently provided for through redundant agents and the CA polling the agents. The next version will include a new agent type – a health agent. Its function, in collaboration with the CAs and OAs, will be to monitor vital signs on the hosts in an attempt to predict if a host is having difficulties and a failure is imminent.

The health agent will be facilitated by pulse monitoring – the extension of heartbeat monitoring from 'I am alive' signals to include health information within the signal, akin to measuring the pulse [8][9]. In this scenario the local agents may fail-over with a new secondary agent being created, yet if this starts to occur frequently this may indicate the host itself is unstable. The health agent can monitor developing scenarios and work with the configuration agent to avoid allocating work to unstable hosts.

5. Related Work

A HIVE is a software environment built from large numbers of dedicated, commodity computers. A HIVE is intended for running transaction-based applications [4] and provides properties such as self-organizing, self-healing and self-maintaining. HIVE is built on the principle that all nodes share the workload and if any one (or more) fails, then the work is restarted at a suitable point and split into as many equal parts as there are available nodes. This means that all machines are doing the same amount of work but at no time will any machine be working at its full potential, thereby reducing the risk of operating system failure.

Microsoft clustering technologies [7] in Windows 2000 and Windows Server 2003 uses a three-part clustering strategy that includes *Network Load Balancing,* designed to address bottlenecks caused by front-end Web services; *Component Load Balancing,* designed to address the unique scalability and availability needs of middle-tier applications; and *Server Cluster Load Balancing* designed to provide fail-over support for back-end applications and services, such as provided by database servers.

IBM's pSeries has self-configuring features such as Reliable, scalable cluster technology (RSCT) and PSSP cluster management.

A server cluster provides high availability by making application software and data available on several servers linked together in a cluster configuration. If a server stops functioning, automated fail-over is provided shifting the workload of the failed server to another server in the cluster. The fail-over process is designed to ensure continuous availability for critical applications and data.

While clusters can be designed to handle failure, they are not fault tolerant with regard to user data. Typically, the recovery of lost work is handled by the application software.

6. 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 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 autonomic extensions of this work will include adding a health agent, heartbeat monitoring and pulse monitoring. Clusters tend to have (relatively) static resources, as opposed to the dynamic nature of the grid. As requirements from the scientific community grow in the future, this research will likely evolve from an autonomic cluster to an autonomic grid.

Acknowledgements

This work is based on extending the ACMS prototype system developed for NASA GSFC by the WPI students: J.D. Baldassari, C.L.Kopec and E.S. Leshay [3]. It is supported in part in the NASA Goddard Space Flight Center Information Systems Division, and the University of Ulster Computer Science Research Institute and the Centre for Software Process Technologies (CSPT), funded by Invest NI through the Centres of Excellence Programme under the European Union Peace II initiative.

References

[1] P. Horn, "Autonomic computing: IBM perspective on the state of information technology", NY, 15th October 2001.

[2] IBM, "An architectural blueprint for autonomic computing", IBM Corporation, April 2003.

[3] J.D. Baldassari, C.L. Kopec, E.S. Leshay, W. Truszkowski, D. Finkel, "Autonomic Cluster Management System (ACMS): A Demonstration of Autonomic Principles at Work", Proc. IEEE Workshop on the Engineering of Autonomic Systems (EASe 2005) at ECBS 2005, MD, USA, 4-8 April, pp 512-518.

[4] Hive Computing http://www.tsunamiresearch.com/, 2004.

[5] Microsoft, "Windows Clustering Technologies – An Overview", Microsoft Corporation, 2004. http://www.microsoft.com/windows2000/techinfo/planning/clustering.asp

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

[7] D.B. Lance, M. Oshima, "Programming and deploying Java Mobile Agents with Aglets", Addison-Wesley, 1998.

[8] R. Sterritt, "Pulse Monitoring: Extending the Health-check for the Autonomic GRID", Proceedings of IEEE Workshop on Autonomic Computing Principles and Architectures (AUCOPA 2003) at INDIN 2003, Banff, Alberta, Canada, 22-23 August 2003, pp 433-440.

[9] R. Sterritt, D. Gunning, A. Meban, P. Henning, "Exploring Autonomic Options in an Unified Fault Management Architecture through Reflex Reactions via Pulse Monitoring", Proc. IEEE Workshop on the Engineering of Autonomic Systems (EASe 2004) at ECBS 2004, Brno, Czech Republic, 24-27 May, pp 449-455.