Watching Your Own Back:  
Self-Managing Multi-Agent Systems

Michael Thome, Aaron Helsinger, Todd Wright, BBN Technologies, Cambridge, MA  
{mthome,ahelsinger,twright}@bbn.com

Abstract—We describe a category of multi-agent applications that address the problem of managing widely distributed Multi-Agent Systems (MAS) by introspectively managing both themselves and other co-resident MAS applications. The result is an extremely fault-tolerant system that has well-defined survivability parameters without requiring deep coordination between the primary application and the management layer. We argue that such an integrated systems-management solution can be both effective and simpler than an out-of-band solution would allow, and can leverage all the flexibility of a MAS infrastructure. We describe and analyze an exemplar of such a self-managed system.

1. INTRODUCTION

There are very few standard approaches to management of distributed multi-agent systems (MAS). A distributed MAS-based application should be able to be widely distributed, as massively parallel as the application logic allows, and insensitive to temporary infrastructure failures, among other attributes. These goals imply a management, monitoring, and control infrastructure with similarly optimistic goals.

Application management is usually done external to the application, often in an extremely ad-hoc fashion. Other times it is handled at a very low level (e.g., SNMP), or through a simple replication and voting scheme (e.g., seti@home). In general, however, non-distributed multi-agent systems do not require management systems. Furthermore, centralized management approaches fail to satisfy because they inherently imply single points of failure.

We propose that an effective approach to this problem is to build a DMAS application specifically for the purpose of managing its own survivability, and then extending its umbrella of influence to cover any other co-located MAS applications.

In this paper we outline the general principles required of an application that will manage itself, as developed under the UltraLog program [1]. We then argue for the general applicability of this approach to other MAS. Finally, we describe our experience building part of such a management application, which we undertook by extending the Cougaar [2], [7] agent architecture to support an embedded agent restart mechanism.

2. GENERAL APPROACH

The general approach we propose can be summed up as the designing of a MAS application to manage itself, then extending the management application’s responsibility to cover one or more non-management applications.

We divide management functionality into at least three groups: 1) situational information gathering, 2) decision making, and 3) action implementation. Each of these groups of functionality can be implemented in a variety of ways, but certain minimal features must be maintained across the board:

• No management function may itself be subject to any single points of failure.
• All management functions must be recoverable.
• All management information must be recoverable.

In addition, at decision points (where a single decision must be made and implemented), there must be a mechanism to decide which of (potentially) many management voices will be heard. Such decision points we call Management Agents (MA).

Although we require that managers be decentralized to avoid inherent single points of failure, it is critical that choices be made coherently. This may be satisfied either by a decision voting mechanism (where a number of decision makers collaborate on a choice before allowing action), or by leader election (where a temporary decision maker is chosen from a group of candidates via a simple group algorithm).

A widely distributed application, particularly one that is comprised of heterogeneous parts, is best managed by dividing it into relatively similar and/or local regions of influence. If these areas align with local-area network boundaries then there are more efficient optimizations available (e.g., LAN-broadcast of status information) that also mitigate
security concerns.

Information about the health and environment of the managed application needs to be available to decision makers. While it is often a useful optimization to gather such information in one place and redistribute it as needed, care must be taken to avoid over-dependence on any sort of central repository of such information. At minimum, we require management algorithms that will do little damage when only partial information is available.

Required features of an in-band MAS Management Application include:

- Communication between management agents and agent (control) services. Minimally, MAs need to have some mechanism to control and monitor agents. Cougaar places a full-fledged agent on each agent life support server yielding, in effect, complete in-band access to a proxy for the host computer, and full control of the hosted agents. While it is simpler for such communication to be in-band, this is not strictly required, as similar levels of functionality may be available via traditional app-server or web-services interfaces (for instance).

- In-band communication between all management agents and managed agents. Manager and managed agents need to be able to communicate freely under normal circumstances. Deviations from this standard do not preclude management, but limit the level of service available in such instances. For instance, an agent that is only periodically connected to the network (by design) may have only much-delayed opportunities for recovery.

- Management API. If a non-management agent can influence management choices by requesting a service or hinting at its state, the management layer can make more optimal and less disruptive choices (e.g., “I’m in the middle of something important right now” as a hint to avoid a highly disruptive agent move.)

- Cooperation with other Management-like applications. While this paper focuses on issues of robust operations, similar arguments may be made for issues of security: the opportunity for cooperation between such management-like applications is highly desirable.

- Visibility into neighboring management spheres of influence. As is true in organization management, it is often helpful to know how adjoining and overlapping management regions are performing, particularly when your non-management applications depend on or are depended-on by the agents in those regions.

Any manageable agent must have a set of features:

- A dependable recovery strategy. This may be anything from a simple reconstruction of functionality for a stateless agent to a full-fledged transactional persistence system with peer reconciliation for a heavyweight agent.

- Stored state must be portable. If an agent requires that any saved state information exist in order to recover from a failure, than that state must itself be shareable or portable so that the agent could be recovered on hardware other than the original. For example, an agent that keeps persisted state on disk could use a Storage-Area Network (SAN).

- A deep measure of health. Host existence measures (ping response) are useful but not sufficient. Better measures focus on liveness (is the agent actually performing its duties) and liveliness (how well is the agent performing).

- Health visibility. There needs to be some dependable measure of agent health visible from outside the agent. Typically, this would be accomplished in-band via some combination of posted status updates and health queries.

- In-band mobility. If the agent infrastructure allows agents to move themselves (or to move others), then the management infrastructure may be able better to optimize and load-balance the agents under its control over the resources available to it.

- Agent lifecycle control. Agent restart functionality is the basic function that needs to be accessible to the management application. If agents can be killed, stopped, or slowed in addition to being started and moved, then the control infrastructure has more control opportunities available.

3. Applicability of Approach

We have described how to build a MAS that is self-managing. We argue that this solution is quite good, because it leverages the strengths of multi-agent systems in general, and the strengths (and weaknesses) of the specific MAS as well as the management functions we are building themselves.

We argue that building a MAS that manages itself is both a) effective and b) efficient. In particular, building the MAS to manage itself means implementing the MAS management system both a) as agents, and b) in-band to the MAS application. There are good reasons for doing each. We will break down our analysis of this approach to MAS management into two pieces. First, we discuss why building a MAS management system using agents is an effective approach in general. Second, we describe why building the management system in-band to the MAS itself is efficient.

3.1. Effectiveness of Agent Architecture

In order to manage a distributed multi-agent system, there are several particular things required. First, the requirement to manage a distributed system drives the design requirements. To manage such a system requires communications and control points to be similarly distributed. Therefore, the management system must be distributed, must have some
portion of its function co-located with the various agents, and must share requirements for efficient communications (of both information and control data). Each component of the management system resident with an agent requires sensors aware of its environment (“Is the agent operating?”), intelligence to reason about the implications of that data, and communications to interact with other portions of the management system.

Satisfying these requirements can be accomplished very effectively with an agent-based architecture. To understand why, consider the requirements. First, the basic functional requirements for distributed processing, reasoning, and communications all match nicely with an agent architecture. Additionally, a management system should be capable of making independent decisions for local agents, particularly in the face of disrupted communications. This clearly suggests an agent architecture.

For example, consider the ability to restart agents in the MAS. In order to do so, the management system requires some ability to launch processes on each host available in the system, the ability to communicate (possibly indirectly) with each agent in the system, etc. These requirements suggest that the management system would require components on every host and distributed communications. In other words, a restart capability requires a distributed system made of multiple components; so using agents is clearly an effective solution.

Implementing our management system using agents has several other benefits. This approach provides resilience if any piece of the management system fails, and makes it easier to manage a distributed application. This approach also permits the management system easily to scale to support larger applications, and it provides the management system the ability to make dynamic changes to the control and communications patterns if necessary (for example, if part of the system loses contact with the rest).

A key benefit of building an agent-management system using agents is that the management system understands the semantics of agent interactions. By working on the time and network scale of agents, the management system can effectively monitor agent communications and reason about the semantics of network load on agent function, for example. In other words, a management system built from agents is “application architecture aware” for an agent system.

By implementing the management system using agents, we leverage all of the pros (and cons) of building systems using agents (redundancy, distributed processing, mobility, network vulnerabilities, etc). As a result, an agent architecture is an effective means for building a management system for a multi-agent system.

### 3.2. Efficiency of In-Band Management

An agent-management system developed using agents can be effective, but building that system as part of the same agent architecture results in efficiency gains in design, implementation, debugging, and ongoing operations. For example, the management sub-system re-uses many of the functions required by the agent application itself. Additionally, by building management as internal to the application, coordination among subsystems is more efficient.

### 3.3. Reuse

The primary motivation for managing an agent system using an in-band sub-system is the efficiency gains of reusing functions. To manage an application requires stable messaging, security, and other functions that are also required by the application itself. By re-using these components, we achieve efficiency of design, implementation, and debugging. Additionally we achieve efficiency of operations, by avoiding the overhead of extra systems. By using fewer sub-systems, there are fewer components to develop, debug, secure, and make robust. As a result, the entire system is easier to manage and more survivable and trustworthy.

In addition, the re-use of subsystems makes management of the management system itself easier. For example, our security components get robustness management and vice versa. Note therefore the inductive nature of this approach: by managing our application in-band, we also manage the management system, giving us a survivable survivability system.

However, this approach does leverage both the pros of the agent application as well as the cons. A change to a single sub-system affects both the application and its management, making piecewise upgrades more difficult. Additionally, a fault in the MAS itself is more likely to affect the management system as well. On the other hand, resolving an issue once resolves it in all contexts. Depending on the application and context, this could balance out in different ways. Our experience indicates that this approach incurs greater initial costs to get the design and implementation correct, but has larger future benefits.

Another possible drawback to an integrated management solution is that the detailed requirements of the management system may not match those of the MAS. As a result, the management system might be constrained by the MAS. For example, the management system might benefit from a broadcast protocol such as UDP, which the MAS does not need or support. However, by requiring the MAS infrastructure to support two applications - the MAS itself and the management system - we build increased flexibility and feature support into the infrastructure. This results in a more powerful infrastructure with more robust features, which is often more stable and easier to manage itself. In our experience [2], requiring the management system to use the features provided by the MAS usually does not greatly constrain the management capabilities, and the benefits in implementation and maintenance simplicity are substantial.
As a final argument for the benefits of in-band agent application management, consider the alternative. If the management system was not built in-band with the MAS itself, separate robustness and security components would be required in every application host, and entire separate (from the MAS) mechanisms would be needed from the MAS for basic underlying functionality, imposing great development, maintenance, and runtime costs. Although stand-alone distributed application substrates do exist to varying degrees of maturity, any such system would require various “bridges” for coordination with the MAS.

3.4. Deep Coordination

A survivability or management system requires various forms of information about the application that it is managing, many of them updated continuously in real-time. In an agent application this data is inherently distributed. By making the management function part of the application, no separate communication and coordination mechanisms are required. While the management function can thus avoid separate query mechanisms, the timing, detail and semantic content of the information connection can be much deeper using in-band management. This is because in-band managers have access to internal communications, interfaces, and constants of the agent application, giving the managers “first-hand” knowledge of the system state. Thus the management system is more than simply “application aware”, but intrinsically understands the application, and at any point can balance the needs of the application with those for robustness, security, or other maintenance.

Thus, a system to manage a multi-agent system is more efficiently built in-band to the MAS itself, and is highly effective by utilizing the strengths of a MAS architecture. Therefore, our approach to build MAS that are self-managing is a powerful and effective solution to the MAS-management problem. Next, we describe how this approach works in practice.

4. CASE STUDY: ULTRALOG RESTART LAYER

In this section we describe the UltraLog Restart Management Infrastructure, which is a distributed agent application that monitors and automatically restarts failed or isolated agents. We summarize the robustness and survivability requirements of the UltraLog logistics application, which guided the design requirements for the Restart Management Infrastructure. Next, as an example of how to make a self-managing MAS application, we present the restart-management design, which leverages the core capabilities of the underlying Cougaar agent architecture, and runs within the agent society as a generic, highly robust monitoring and restart service.

4.1. The UltraLog Logistics Application

UltraLog is a Defense Advanced Research Projects Agency (DARPA) sponsored research project focused on creating survivable, large scale, distributed-agent systems capable of operating effectively in chaotic environments. Currently, the project is pursuing the development of technologies for operating in wartime environments. The objective of the UltraLog project is to create a comprehensive capability that will enable a massive scale, trusted, distributed-agent infrastructure for operational logistics to be survivable under the most extreme circumstances.

UltraLog’s primary application is the planning and simulated execution of military-logistics operations. The distributed logistics application is composed of hundreds (1200+ in 2004, distributed over 53 hosts) of autonomous agents, each representing a combat or logistics organization. The agents coordinate to plan the detailed inventory supply and transportation for a period representing 180 days of sustained military operations in a major regional contingency. As the plan is generated, we simulate execution of the plan, with associated changes in requirements and deviations between expected and observed behaviors.

By itself, the logistics application is a highly capable planning and execution system, but it is not survivable or robust. The goal of the UltraLog project is to explore both logistics and highly survivable agent systems, capable of surviving stresses such as the unexpected temporary or permanent loss of hardware or network resources. We define survivability as the ability to maintain function in the face of environmental stress, where function utility is a measurable runtime property of the logistics application. For more details on the UltraLog survivability approach, see [4], [5].

In UltraLog, defense agents are added to the agent system to maintain system survivability and robustness. These defenses are implemented as agent applications that both manage the logistics agents and manage one another. In this section, we will focus on the restart infrastructure, which is responsible for monitoring agent liveness, automatically restarting, and load-balancing agents. Other agent applications include an agent-based security system, denial-of-service-detection-and-response agents, and defense-coordination agents, plus basic Cougaar infrastructure agents that function as naming services.

4.2. Restart Management Design

The requirements of the UltraLog program guided many of the design points of the agent-restart infrastructure. The primary requirement was to handle the unexpected temporary or permanent loss of hardware or network resources, requiring the restart of agents to new hosts. The restart mechanism must use in-band sensors to monitor agent liveness and activate agent restarts. The restart design must be highly scalable and make efficient use of network and hardware resources. Additionally, the restart infrastructure is subject to the same stresses as the logistics application, including the loss of restart-management agents, and must guard against security intrusions.
The above requirements led us to an agent-based design, due to the need for autonomous distributed processing and peer communications. We decided to use the same Cougaar agent architecture that the logistics application uses, since it provided many of the agent-infrastructure capabilities necessary to build the restart agents. In addition, due to the requirement for in-band sensors and security, we designed the restart agents to run as normal peers of the logistics and security agents.

The restart mechanism imposed relatively few requirements on the logistics application, many of which were already provided by the Cougaar architecture. A basic requirement is to avoid dependencies upon specific hosts or hardware, to allow greater flexibility in the choice of restart locations, and which was aided by our use of the Java language. We also made use of Cougaar's state persistence and rehydration capabilities to restore agent state, and in case of state loss we rely upon agent peer-wise reconciliation to recreate lost state. In such a case, reconciliation is crucial, since hardware loss may result in the loss of persisted data, and the nature of logistics planning allows reconciliation to regenerate lost workflow plans.

Management agents are specified in the agent society configuration just like any other Cougaar agent. They leverage the basic capabilities of the Cougaar architecture, such as reliable, robust messaging, naming, servlet-based user interfaces, and blackboard-based agent communication channels. Management agents can be dynamically added, moved, and removed from the system. In particular, the restart-management agents greatly benefit from the survivability capabilities added by other defense agents, such as security agents that issue and renew agent cryptographic certificates for encrypted and non-reputable inter-agent communication, avoiding the need for a duplicate implementation in the restart infrastructure.

Several forms of “health” sensors are used to monitor agent liveness. Passive agent messaging traffic metrics are used to monitor agent liveness with minimal performance impact. Active “heartbeat” messages are periodically “pushed” from monitored nodes to the active restart management agent. If all the above fail, the restart management agent sends active “ping” messages to unresponsive nodes to check for liveness. These sensors are adaptively tuned to minimize message traffic, balanced against the risk of false non-liveness detection, which would cause undesired agent restarts.

Undesired restarts can result in duplicate agents, which is inevitable in some forms of network partition. This is partially handled by the Cougaar architecture, which tracks agent versions, guards against duplicates, and forces reconciliation to recover from duplication. The naming services also arbitrate between duplicate agents, since only one agent can bind its transport addresses in the registry to a given agent name.

Once an agent is detected as dead, a genetic-algorithm based load balancer determines which host and node should restart the agent. This choice is guided by policy, performance metrics such as bandwidth and memory use, and security restrictions. The design is pluggable, allowing for alternate load-balancer implementations.

As noted above, multiple restart-management agents are loaded into each enclave, where only one restart manager is active at any time. The active manager is determined by a bully style, leader-election algorithm [6], where the active manager must renew a lease to maintain control over the enclave. The non-active managers participate in the elections, and take over the restart management responsibilities if the active management agent is lost. The behavior is configurable to require varying degrees of concurrence, which determines the behavior in case of large-scale enclave network partitions.

By default, the restart management sensors use generic in-

---

**Figure 1 - Cougaar Restart Management Information Flow**

For scalability, we split agents into non-overlapping groups called “enclaves”, which modeled network and hardware administrative boundaries such as local area networks (LANs). Each enclave includes a separate set of management agents that are responsible for only their enclave’s agents. This design choice allows us to ensure scalability by reducing the number of agents in an enclave, which is limited by communication delays, management processing overhead, and more general administration boundaries.

Each enclave contains one or more management agents. Only one management agent within each enclave is active at any given time, controlled by an election algorithm that is described below. As an example, in the 2004 test configuration, there are nine enclaves containing on average 16 hosts, each of which is capable of standing up a Management Agent if elected to do so. The active management agent is responsible for monitoring the liveness of all agents in the enclave, deciding which agents to restart and where to restart them, and avoiding undesired restarts by coordinating with other defense agents (e.g., avoiding the restart of a compromised agent).
rastructure metrics to determine liveness and to guide load-balancing choices. However, it is relatively straightforward to add custom sensors and control mechanisms, since the restart-management agents use the same messaging and naming infrastructure as the logistics and defense agents. For example, the logistics agents can pass planning-specific information to the load balancer, which can be used to optimize the balancing beyond system-level metrics.

We plan to further reduce the sensor overhead by optimizing the collection of both passive and active metrics, and by implementing adaptive-tuning mechanisms. We plan to improve cross-defense coordination by adding optional tighter integration with coordination agents, in addition to carefully exploring the correct actions to take when coordinator agents are lost or unreachable. We also plan to support cross-enclave agent mobility, which will initially use a simple manager hand-off, but will be enhanced to use transactions in case either or both managers crash during or after the move.

Beyond these improvements to our restart mechanism, there are several other areas of future research in self-managing MAS. In particular, we are currently investigating techniques to allow layered defense and management while avoiding destructive interactions among the management functions. Additionally, we must apply these techniques to significantly different MAS applications to demonstrate the general utility of the approach.

5. Conclusion

The importance for the adoption of multi-agent architectures is clear. Distributed multi-agent systems are difficult to manage and manage well. By building a distributed management system within our agent application, we realize clear efficiency gains, making the MAS management problem more tractable.

6. Acknowledgment

This work was performed under DARPA UltraLog [2] contract #MDA972-01-C-0025.

7. References


