Robustness Infrastructure for Multi-Agent Systems

Ronald D. Snyder
Dr. Douglas C. MacKenzie
Mobile Intelligence Corporation
ron@mobile-intelligence.com
doug@mobile-intelligence.com

Raymond S. Tomlinson
BBN Technologies
rtomlinson@bbn.com

Abstract

When used for mission critical applications multi-agent systems must be capable of sustaining uninterrupted operations in the face of both hardware and software failures. Applications of this type are often distributed over a large number of geographically dispersed hosts, which make them susceptible to a wide range of failures. A fault tolerant infrastructure must be able to detect and adapt to these failures to provide continuity of processing. This paper discusses the approach used to provide such an infrastructure in the Cougaar multi-agent system.

1. Introduction

A multi-agent system consists of autonomous computational entities (agents) that interact with one another towards a common goal that is beyond their individual capabilities. A multi-agent architecture provides many benefits in the construction of large distributed applications. Foremost is their inherent scalability and flexibility. However, the greater the number of agents and hosts, the higher the probability that one of them will be subject to failure. Since multi-agent applications rely on the collaboration of agents, a failure of even one agent has the potential to bring all processing to an end.

While multi-agent systems are modular and therefore good platforms for building fault tolerant systems, they are also non-deterministic, making it difficult to guarantee specific behavior in fault situations. Components of a distributed system can fail in different ways [1]. In the simplest case of a *crash failure*, a component simply ceases to function. A special case of a crash failure is a *fail-stop failure* in which a crash results in the component transitioning to a state that allows other components to detect that it has failed (e. g., via the absence of a heartbeat message). More complex failure modes are also possible, such as when a component fails by not functioning "correctly" due to a design flaw or some other non-fatal condition. In this paper, we focus on the problem of fail-stop crash failures.

Multi-agent systems designed for asynchronous operation are inherently tolerant to fail-stop failures in that they are able to continue processing while fault detection and recovery take place, rather than waiting until such actions have completed. Multi-agent systems are also more forgiving of temporary fluctuations in network speeds than systems designed to assume some synchrony. The challenge then is the development of a suitable failure detection and recovery capability.

2. Fault Detection

The implementation of a failure detector in a multi-agent system is problematic. It is known that some problems, such as Consensus, cannot be solved in a distributed asynchronous system. In the consensus problem, all processes propose values and later decide a value. The key requirement is that all processes eventually agree on the decided value. In message-passing, asynchronous systems each process runs according to its own clock and no assumption is made about the relative speeds of different clocks. Processes communicate solely by sending messages which are neither corrupted nor lost, but there is no guarantee on the time of delivery. This scenario is quite prevalent in many types of real networked applications including multi-agent systems. Intuitively, reaching consensus in such systems is impossible because processes cannot tell whether another process is dead or just temporarily isolated from the outside world because its messages are delayed; if they wait they might do so forever, and if they decide they might find out that the other process already came to a different decision. This rough intuition was formalized by Fischer, Lynch, and Paterson [2]. They concluded that there is no deterministic protocol for consensus in message-passing, asynchronous systems. That is, no protocol can tolerate even one process failure. This finding highlights a fundamental property distinguishing synchronous and asynchronous systems. The unsolvability of consensus poses a significant challenge in the design of a fault-tolerant asynchronous distributed system as it is used to implement essential fault-tolerant functions such as leader election and voting.

In an asynchronous system where network latencies are unbounded, it is impossible to distinguish between a failed component and one that is simply taking a long time to respond. Thus, it is impossible to implement a fault detector that is both 100% accurate and complete. Completeness means that every failure is eventually detected, while

This work is supported by DARPA through Department of the Interior contract #NBCHC10004. Approved for Public Release, Distribution Unlimited.

accuracy means that each detected failure corresponds to a dead process. An approach to address the unsolvability of Consensus in asynchronous systems was proposed by Chandra and Toueg [3]. They augmented the asynchronous model of computation with the notion of an *unreliable failure detector:* a failure detector that is allowed to erroneously indicate that a component has failed, as long as any such errors are eventually corrected.

The use of unreliable failure detectors is a good fit for multi-agent systems as it means the detector does not have to be centralized, nor does a globally consistent state need to be maintained across detectors. Each component of the system has access to its own detector and each detector can potentially produce a different view of which system components have failed. A correct detection can be achieved as long as the distributed detector meets some minimal requirements for completeness and accuracy. In particular:

- all failed components are eventually discovered, and
- at least one functioning component is known to be functioning by all functioning components.

The use of unreliable failure detectors has a significant limitation though. Algorithms can guarantee termination by a combination of iteration and the fact that the failure detector will *eventually* identify all failed components and at least one functioning component, but cannot provide a guarantee on how long this will take. In real systems this unbounded wait is unacceptable. Ultimately, some threshold must be established as to when the information provided by a failure detector is to be believed. Decisions must then be based on the probability of the failure report being in error. Later we show that the probability of an erroneous report is relatively low and decreases the longer one waits. In general, this believability threshold corresponds to the point at which the cost of waiting for an absolutely correct determination exceeds the cost that would be incurred if an incorrect action was performed. This leads to the requirement that the system be capable of recovering from an improper action performed as a result of an incorrect diagnosis.

3. Failure Detection in Cougaar

3.1. Cougaar Overview

Cougaar is a Java-based architecture for the construction of large-scale distributed agent-based applications. It is an Open Source product of a multi year DARPA research project exploring large-scale, scalable, heterogeneous, distributed and survivable Multi-Agent Systems. Current work is focused on supporting high-quality robustness, security and scalability mechanisms in the infrastructure in the context of a large military logistics system. This section provides a brief overview of the Cougaar architecture summarized from the Cougaar Architecture Document [4]. Detailed information can be found in the Cougaar Developers Guide [5].

The *Agent* is the principal element in the Cougaar architecture. An agent typically models a particular organization, business process or algorithm. An agent consists of two primary components, a *blackboard* and *plugins*. The plugins are software components that provide behaviors and business logic to the agent.

A blackboard is an agent-local memory store that supports publish/subscribe semantics. Components within an agent can add/change/remove objects from the blackboard and subscribe to local add/change/remove notifications. Infrastructure components monitor the local blackboard and transparently send messages to other agents and alter the blackboard when the local agent receives messages. Thus, a benefit of an agent blackboard is that it hides the details of the message transport from plugins. The blackboard is also a key architectural element supporting an agent restart capability. Cougaar agents are designed to be restartable which means that if an agent stops execution it can be started again and has access to state data from its prior invocation. It is thus able to resume processing with little or no rework. In support of this capability, plugins are intended to be stateless. That is, they rely on the blackboard for all state. In this way state can be transparently preserved and reconstructed if necessary during a restart.

A Cougaar *Society* is a collection of agents that interact to collectively solve a problem or class of problems. Within a Cougaar Society, agents execute on a *node,* which is a single Java Virtual Machine that can contain multiple agents. All agents on the same node share the same processor, memory pool, disk, and communication channels. The allocation of agents to nodes is not necessarily domain related, but rather based on a distribution of agents to available resources.

The remainder of this paper discusses failure detection and recovery in the context of the Cougaar multi-agent system. We describe the mechanisms employed in Cougaar to provide node level fault tolerance. In particular, we look at the approach used in Cougaar to detect and restart agents on failed nodes.

3.2. Management Agents (Sentinels)

The failure detection approach employed by Cougaar involves the use of *sentinels* and distributed (unreliable) fault detectors. Sentinels [6] are watchdog entities responsible for protecting the system from undesirable states. They form a control structure that monitors system state and intervenes when necessary, according to specified guidelines. An advantage of a sentinel approach is that it allows developers to implement system functionality and then add on a control system that can be modified with little or no disruption to the rest of the system. The alternative is to include logic in each agent for dealing with failures in collaborating agents. This approach, sometimes referred to as the *survivalist* approach, to building fault tolerant distributed systems has serious shortcomings. In particular, developing survivalist agents greatly increases the burden on agent developers. For this to be an effective approach, all agents have to include carefully coordinated and potentially require rather complex mechanisms for handling the loss of a collaborating agent.

In a sentinel-based control structure, the sentinel maintains the authoritative system status and is solely responsible for initiating corrective action, agreement by other entities is not required. Consensus is only required when a sentinel is lost and must be restarted. In this situation, some other entity must be responsible for restarting the sentinel. Otherwise, a single point of failure exists, weakening the systems fault tolerance. The mechanism employed by Cougaar is to elect a temporary leader to restart the dead sentinel requiring agreement on the status of the sentinel and election of temporary leader to perform the corrective action.

In Cougaar the sentinels are called *robustness managers*. They monitor agent liveness and restart agents that are unresponsive. Larger systems often partition the agents into multiple robustness communities that are monitored by individual robustness managers. A robustness community typically includes the agents that are associated with a set of computing resources, based on their physical location or organizational responsibilities. The role of the robustness manager agent is to continuously monitor the status of the community members and initiate corrective action (agent restarts) when anomalies are detected. The manager may be a special agent dedicated to performing management tasks or could alternately be a regular agent that is assigned this additional responsibility.

Cougaar employs a two-tier robustness architecture consisting of a node-level and agent-level monitoring as depicted in Figure 1. This figure show 12 regular agents and 3 nodes grouped into a robustness community. At the lowest level, agents send heartbeats to a special node agent that is responsible for monitoring the health status of local agents. This node agent periodically sends an aggregate heartbeat to the robustness manager and other node agents in its *robustness community*. Using this approach, each node in a robustness community has access to status information about all entities in the robustness community, though the robustness manger is the only entity permitted to act on this information.

When the robustness manager is lost, the surviving node agents are responsible for detecting this condition and electing a temporary leader to restart the manager. All active node agents must agree on the identity of the temporary manager. Once the temporary manger is selected it will restart the dead manager and then immediately relinquishes this role once the manager becomes active.

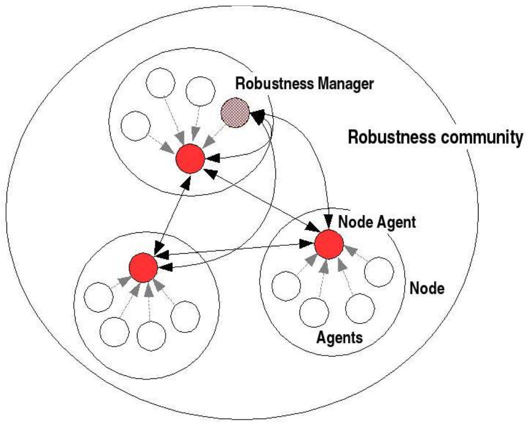

Figure 1. Health Monitoring

4. Failure Recovery

4.1. Failure Recovery Strategies

Failure recovery involves the replacement of a failed component with a suitable replacement via some sort of replication strategy. Replication strategies are typically characterized as being either active or passive. Active replication involves the use of a "hot standby" and provides for fast recovery. Active replication is most applicable to systems with strict time constraints. The drawback to active replication is in its resource utilization which limits the overall scalability and throughput of systems using this strategy. Passive replication is typically accomplished by creating snapshots of critical state that can be used to recreate failed components. Passive replication imposes less of a tax on the execution environment at the expense of longer recovery delay. Active replication is often chosen when the failure rate is expected to be high or when an application has real-time constraints. In most other situations passive replication is probably more suitable.

4.1.1. Active Agent Replication

The active replication recovery strategy involves maintaining standby copies of agents in another location. The approach utilizes a mechanism wherein all state changes are transmitted to the standby agents as they occur and before the effect of those state changes is allowed to affect the other agents in the society. This approach imposes a significant performance impact since it requires the continual exchange of a potentially large amount of data with the standby agent, and during these exchanges, the agent is effectively at a standstill because it can neither start processing newly arrived tasks nor send tasks to other agents.

4.1.2. Passive Agent Replication

The passive replication strategy use checkpointing, rollback and recovery which is a general, powerful approach to handling errors in a system [7]. The technique requires that a system record its state (checkpointing) periodically during normal operation and, upon failure, restore a previous consistent state (rollback), and restart the execution from the restored state. Since agents in a multi-agent system do not share memory, a global state of the system is thus composed of a set of local states of the agents in the system and the system state must be checkpointed distributively over all the agents. A local checkpoint is a saved copy of an earlier local state of one agent. A global checkpoint of the system is a set of local checkpoints, one for each agent.

5. Failure recovery in Cougaar

Cougaar uses passive replication via the checkpointing, rollback and recovery strategy for failure recovery. While this approach does not provide as rapid a recovery as the agent replication mechanism, it is more scalable and delivers much better runtime performance. In Cougaar this approach is called *persistence* and *rehydration*. Persistence and rehydration allows agents to recover from failures when they operate over long periods of time. The failures could be caused by power disruption, deliberate reboots, accidental reboots, and hardware failures. Persistence is achieved by saving the state of every published object and the state of the subscribers relative to those objects on non-volatile media. That media can be a simple flat file, a local database, a remote database, or just about any other form that can be populated with serialized Java objects. When an agent is restarted, its state is restored from the persisted data in a process called rehydration. Since the restored state only consists of an agent's blackboard, components can continue executing as if there had been no interruption. In means that, if a plugin maintains no internal state, except that which is published to the blackboard, it will resume execution after rehydration without knowing that rehydration even occurred.

5.1. State Persistence

Cougaar's built-in persistence mechanism supports agent recovery after a failure by enabling the system to recreate an agent in a state saved prior to the failure. This avoids the need to completely recreate state data developed by that agent and by all the other agents to which the failed agent was collaborating. Cougaar persistence supports both optimistic and pessimistic modes of operation. When operating in pessimistic mode, interactions with other agents are rigidly synchronized with persistence snapshots. In this mode, restarted agents are quickly integrated into a running society with minimal or no loss of state requiring rework. As with the agent replication strategy, this mode incurs a significant performance penalty and is not normally used. In the optimistic or “lazy” mode, the system assumes that restarts will not occur and that the persisted data is not absolutely essential. By making these two assumptions, the state of an agent can be saved much less frequently and rigid inter-agent persistence synchronization is not required. This has the benefit that the objects to be saved will have had an opportunity to evolve through several stages and only their most recent state will need to be saved. In this mode, the states of an agent and the agents with which it interacts are allowed to progress without absolute assurance that the current state can be recovered in the event of a restart. The downside of this is that, if an agent is restarted, it will be in a state that is inconsistent with the states of the other agents with which it interacts. This inconsistency must be reconciled or else incorrect operation will ensue.

5.2. State Rehydration and Reconciliation

The reconciliation process depends on intrinsic redundancy in Cougaar agent interactions. In all cases, these interactions are in terms of objects that are (logically) shared between the blackboards of the agents. Furthermore, this sharing comprises a master-slave relationship. A typical example is a task that has been allocated to another agent. A copy of the task is sent to that other agent when the allocation is first created and then updated copies are sent as changes are made to the source task.

These shared objects have a well-defined life cycle: they are created in an initial state, progress through a number of intermediate states and then are destroyed. The destruction of a Task can actually signify two things: the plan has changed and the Task is no longer desired and time has progressed and the Task is no longer relevant. In the latter case, the agents should have already permanently factored the effect of the deleted task into their planning. The Task deletion process marks Tasks to distinguish these cases. There is *no* semantic significance to the events that signal this progression, but the events do allow the agents to perform incremental updates. Since there is no significance to the path by which an object arrived in its current state (including its non-existence), the resynchronization process is not tasked with detailing these intermediate states; it need only insure that the slave object has the same state as the master.

In normal operation, the slave tracks the master because the message transport insures reliable, ordered delivery of the state changes. When a restart occurs, however, there are several opportunities for the slave and master states to diverge. The three most obvious are:

1. state changes sent just prior to the restart were not delivered,
2. state changes received just prior to the restart were not persisted, and
3. the restarted agent has reverted to an earlier state of the object.

While there may be other ways for the states to diverge, the exact mechanism is not important. It is only important that agreement be restored between the master and slave copies.

Ideally, the state given by the copy that was not subject to restart would be used. However, this poses difficulties when that copy is the slave copy because the agent that had the master copy has lost the causal links leading to the value of the slave copy. (If the causal links were not lost the master and slave copies would still be equal.) Recovering these causal links is problematic and is not attempted.

5.3. Reconciliation Procedure

The resynchronization procedure occurs between pairs of agents as follows: Both agents must realize that a restart has occurred. The restarting agent knows this trivially. For another agent, it is more difficult. Currently, this is achieved by periodically checking the incarnation number of the agent in the white pages.

Both agents perform the same resynchronization procedure though not necessarily at the same time. The procedure seeks to establish two invariants: all objects that an agent sent to another agent exist in that other agent with the same values and an agent has no object (Task, Transferable, etc.) received from another agent that does not exist in that other agent. To this end, each agent resends all the objects that it previously sent to the other agent and sends verification requests about all the objects it previously received from the other agent. If the resent objects are already present in the other agent, their values are updated and, if changed, a "change" event is processed. If the resent objects are not already present, they are added to the blackboard and an "add" event is processed. If a "verification" message refers to an object that is no longer on the blackboard, a "rescind" message is sent back just as if it had been removed from the blackboard. The rescind message is processed and removes the now spurious object. This handles two cases: the original "rescind" message was lost because the agent reverted to a state prior to receiving the "rescind" message and the other agent reverted to an earlier state prior to the creation of the task. In both cases, the task should not exist, so sending the "rescind" message is appropriate.

Every task that is removed from an agent that did not restart represents lost work. The restarted agent will ultimately create new tasks that will be equivalent to the lost tasks, but they will usually not be identical. This is "ok" because of the non-deterministic nature of Cougaar. Also, performing the reconciliation steps in a certain order can be more efficient than some other order. For example, ascertaining that an incoming task has been rescinded before re-sending the resultant tasks would avoid the effect on downstream agents of sending tasks and then almost immediately rescinding those same tasks.

Correct operation of the reconciliation procedure (and Cougaar, in general) depends on ordered delivery of messages between agents. This is clear in the case of a succession of changes to an object. An earlier change arriving after later one will leave the object in the wrong state. When an agent restarts, it is essential that the other agents not receive messages from earlier incarnations of the restarted agent after beginning to receive messages from the new incarnation. This requirement is satisfied by the current message transports.

When an agent is restored after a failure, its state may not be in synchrony with other agents. To rectify this, the recovering agent and the other agents reconcile their respective states so that they are in agreement. The action is greatly facilitated by the design of the inter-agent protocols. However, the same problem exists with respect to external entities (databases, EDI systems, etc.). These, too, need to be brought back into synchrony. Ideally, this resynchronization process would mimic that which occurs between Cougaar agents, but there is no general solution. Every external agent could be different so each could, potential, require a unique solution. A middle ground exists where the external interactions are allowed only when based on reliably persisted data.

6. Tuning and Adaptation

As discussed earlier, failure detection in a distributed asynchronous system must be performed using unreliable detectors. The detectors are unreliable in the sense that they cannot guarantee both completeness and accuracy. In other words, the detector will eventually identify all failed components, but can never be absolutely certain that a suspected component has actually failed. The decision of when to act on this information must therefore be based on probability and a trade-off between the cost of inaction versus the cost of recovering from an incorrect action.

The use of an unreliable fault detector also implies that the system is able to tolerate duplicate entities as 100% accuracy and completeness cannot be guaranteed by the fault detector. Indeed, the existence of duplicate agents in a multiple agent system must be dealt with such a way that computational accuracy is maintained.

With respect to the Cougaar robustness infrastructure, the cost of performing an incorrect action (starting a duplicate agent that wasn't actually dead) includes:

- time required to start the agent,
- resources consumed by duplicate agent,
- lost work that must be re-accomplished when duplicated agent's state is rolled back and resynchronized with collaborating agents, and
- effort to kill the now superceded agent.

In most cases, the contribution of the first two elements is relatively minor, the bulk of the cost is associated with the rollback and resynchronization of state across the restarted agent and any collaborating agents. At the top level, a rough approximation of this cost can be quantified by measuring the time required to complete a known process.

In the Cougaar implementation, a heartbeat latency is measured for each node in a robustness community. Using this latency data, thresholds can be established for each node based on a target fault detector accuracy rate. For instance, if an accuracy rate of 95% was desired, the threshold could be calculated using a statistical mean and deviation corresponding to this rate. A number of experiments can then be run at different accuracy values to determine an optimal level. While this approach will produce a value that is optimized for a given configuration, the value will likely be reasonable for a broader class of applications. Figure 2 is a graph that depicts the trade-off between accuracy and cost (expressed as the time to complete a task). When the restart threshold is set to a low accuracy (high incidence of unintended restarts), the cost will be high as the system will be performing frequent (unnecessary) restarts and resynchronization based on heartbeat timeouts that are too aggressive. On the other end of the scale, a high accuracy level will similarly extend completion times as detector wait times are increased and agents have actually terminated.

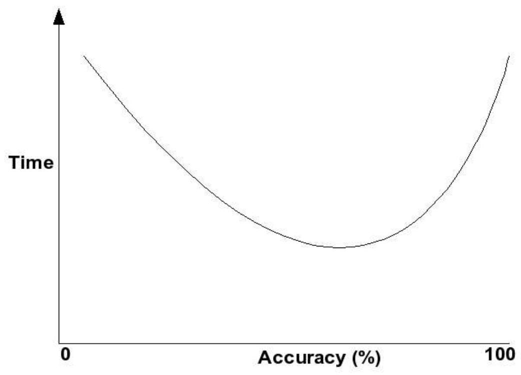

Figure 2. Accuracy vs Cost Tradeoff

7. Summary

In this paper we've discussed some of the issues that must be addressed when adding fault tolerance capabilities to a multi-agent system. A primary challenge lies in how to deal with the uncertainty inherent in detecting failures in a distributed asynchronous architecture such as that used by most multi-agent systems. In this type of architecture, fault detection must be based on the use of unreliable detectors and the system infrastructure must be capable of handling erroneous diagnoses. We explored the implementation of a robustness infrastructure in the Cougaar multi-agent system and its use of a sentinel-based fault detector and a checkpoint, rollback, and resynchronization recovery strategy. Ultimately, the strategy used by a particular system will be governed by its problem domain. In the implementation discussed in this paper the approach took advantage of special properties of the application domain. For instance, the reconciliation process discussed in section 5 took advantage of special properties of task work flows and planning inherent in a logistics or supply chain application, permitting the generation of lost state. This is ideal for the presented application but is not a general property. Other techniques, such as the use of a sentinel agent, represent general concepts that should be applicable to a broader range of applications. An implementation of the robustness infrastructure described in this paper has been applied to a large-scale logistics application being developed as part of a DARPA funded project. The application includes well over 1000 agents running on approximately 100 host computers. The system is able to recover from events that disable up to 40% of all agents. In most cases the detection and restart of dead agents occurs in less than 5 minutes with low a incidence (< .1%) of false restarts (restarting agents that aren't actually dead).

8. Acknowledgments

The work described here was sponsored by the DARPA UltraLog contract #NBCHC10004. These ideas represent contributions made by Qing Lin and many other individuals who have participated in the DARPA ALP and UltraLog programs.

9. References

[1] S. Mullender, *Distributed Systems.* ACM Press, 1993

[2] M. J. Fischer, N. A. Lynch, and M. S. Paterson. "Impossibility of distributed consensus with one faulty process", *Journal of the ACM*, pp. 374-382, Apr. 1985

[3] T. D. Chandra, V. Hadzilacos, and S. Toueg, "The weakest failure detector for solving consensus", *Journal of the ACM* (JACM), vol. 43, no. 4, pp. 685 722, 1996.

[4] BBN Technologies, *Cougaar Architecture Document V11.0*, 8 March 2004, http://www.cougaar.org/

[5] BBN Technologies, *Cougaar Developers Guide V11.0*, 8 March 2004, http://www.cougaar.org/

[6] S. Hagg. "A sentinel approach to fault handling in multi-agent systems", *Proceedings of the Second Australian Workshop on Distributed AI, Cairns*, Australia, 1996.

[7] J. Cao, G. H. Chan, W. Jia, and T. S. Dillon. "Checkpointing and Rollback of Wide-Area Distributed Applications using Mobile Agents", *15th International Parallel and Distributed Processing Symposium (IPDPS'01)*, April 23 - 27, 2001 San Francisco, California, USA