

Experience with Task/Allocation Coordination Primitive for Building Survivable Multi-Agent Systems.

Sarah Siracuse, Ray Tomlinson, Todd Wright, John Zinky

BBN Technologies

ssiracus@bbn.com, rtomlinso@bbn.com, twright@bbn.com, jzinky@bbn.com

Abstract

A key goal in building survivable agent frameworks is to create application-level coordination primitives that fit naturally within the application's domain and are capable of being made robust and efficient by the agent framework. The Cougaar agent framework supports several types of application-level coordination primitives, including Task/Allocation. Under the DARPA-funded UltraLog project, BBN used the Cougaar agent framework to create survivable agent societies based on these primitives. The Cougaar logistics application that was produced used the Task/Allocation coordination primitive to decompose work among multiple agents. Other coordination primitives were used to monitor and control the agent infrastructure itself. When combined and run together, the UltraLog societies were able to recover from a substantial, 45% infrastructure loss and were still able to complete their jobs with minimal impact on performance.

1. Introduction

Cougaar [1] is the product of an eight-year US Defense Advanced Research Projects Agency (DARPA) funded effort to explore the potential of distributed multi-agent systems for military logistics. Under the Advanced Logistics (ALP) and then UltraLog [2] programs, DARPA challenged a collection of companies and universities to build a detailed US military logistics plan for a major 180-day deployment, and then to execute that plan, including irregular changes to that plan. This application requires managing large quantities of data, and responding to frequent changes in requirements and external conditions with dynamic re-planning. Characteristics of the application include being data intensive, security sensitive, and time critical, while required being robust to expected failures and adversary attacks compromising 45% of infrastructure, with minimal performance and capabilities degradation [3].

To test such a significant application, the UltraLog program distributed over 1000 agents across nearly 100 machines. This application was subjected to numerous stress tests and annual independent evaluations [3].

The difficulty in building scalable distributed applications resides in handling systemic issues, such as security, robustness, and performance. Distributed system middleware strives to hide systemic issues from the application programmer, but this approach usually results in extreme cost in runtime efficiency and infrastructure complexity. The Cougaar agent architecture takes the approach that while failures happen they are rare, so some awareness of failures by the application is tolerable. The Cougaar failure recovery mechanism is optimistic in the sense that it precedes forward with little overhead, until a failure is encountered and then the Cougaar recovery mechanisms relies on the application to ultimately reconcile inconsistencies. While having the application participate in the recovery is usually considered a burden to application developers, in this case the reconciliation process uses the same primitives that are used by the application to perform its business processes. Thus to the application, the recovery is just another contingency to handle within the normal order of business.

Using this model the Cougaar agent architecture provides a reliable, scalable, and configurable platform for deploying distributed applications. Cougaar's survivability infrastructure puts restrictions on how applications are written. If an application adheres to these restrictions, the Cougaar infrastructure manages run-time execution and increases the application survivability to resource failure and security attacks. This paper will concentrate on survivability issues from the application developer perspective, i.e. how a complex application can be written within the discipline imposed by Cougaar Task/Allocation coordination primitive. Other Cougaar survivability features are addressed in other papers (Section 5).

In this paper we describe our experience using high-level coordination primitives to make scaleable and survivable agent-based applications. Section 2

discusses the characteristics of the logistics planning application and the fault model for the resources on which it runs. Section 3 will describe the Cougar agent framework and its application development API. Section 4 describes the Task/Allocation coordination primitive and how it is used to recover from faults in logistics planning applications. Section 5 has pointers to further reading about Cougar survivability mechanisms

2. Characteristics of Logistics Planning

The class of non-deterministic applications has properties that can be exploited by coordination primitives to help make the fielded application more survivable. A non-deterministic distributed application is structured or decomposed into a set of outstanding remote tasks for which the order of completion of the tasks is not important. Remote peers can process these tasks in parallel and asynchronously, and return the results in any order.

Logistics planning is conducive to these properties and is an excellent example of a non-deterministic distributed application. The primary function of logistics planning involves processing storage, organization, and transportation requirements of items or personnel and issuing plans to fulfill those requirements. Planning poses an optimization problem in that, many plans can meet the requirements, but some are more costly than others. Determining an optimal plan can be impractical because both the requirements and the availability of transportation resources change over time. A logistics planning application determines an inexpensive plan, tracks changes in requirements and resources, and modifies the plan to match the new situation.

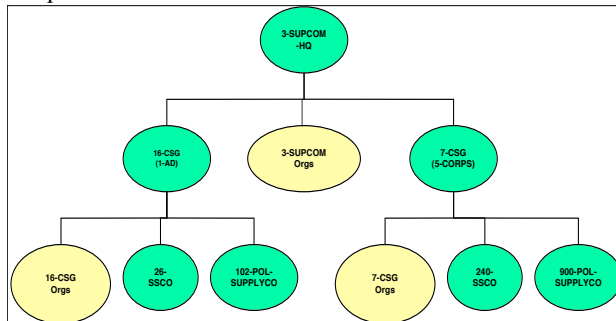


Figure 1. Task Decomposition Among Agents

Our specific logistic planning application decomposes the planning process into many high-level tasks, which are allocated to remote agents to solve. These agents in turn, refine their task into multiple subtasks and pass these off to other agents. Agents also manage inventories of parts and transportation equipment. At the lowest level are agreements to use specific parts and transportation resources to fulfill the

tasks. Figure 1 depicts the task decomposition, which follows the organization units in the supply chain. Inventory agent cross cut the command hierarchy, to work among peers to determine the availability of parts and equipment.

The Task/Allocation coordination primitives we discuss in this paper exploits the following properties of the planning application:

Non-Determinism: When a series of tasks complete, the order of completion is not important. This means that if the application were to run the same problem multiple times, the order for the task completions may be different for different runs. Also, the final result may be different, but the result must still be valid. So from a reliability standpoint, the intermediate states are not causally linked. Only the final state is important. If the application needs to causally link intermediate states, then those states must be explicitly represented. Thus the application is responsible for maintaining causal order and not the framework. This gives the framework leeway to order events to efficiently use resources or improve performance.

Parallel and asynchronous: For a tasking agent, its sub-tasks can be run in parallel. Without parallelism, an application's performance would be limited by the processing delays associated with a series of synchronous calls. Some of the inter-agent communication delay can't be removed no matter how efficient the framework. For example, speed of light delay limits coast-to-coast synchronous remote procedure calls (RPCs) to a maximum of 30 calls per second. To achieve higher throughput many calls must be asynchronously processed in parallel.

Intrinsic redundancy: The allocation agreements are between pairs of agents. When two agents share the same state and one agent fails, then the state can be recovered by querying the remaining agent. If the agent state does not have side effects, then if both agents fail the state can be ignored and it will be reconstructed when the agents are rerun. Note that due to the non-determinism property, the end state might not be the same end state as the original run.

Rescind or rollback previous results: When requirements change or parts or equipment is no longer available, old agreements must be rescinded. Undoing old agreements may be an expensive operation, but may not happen often. For example, over booking flights may result in a penalty to the airlines, but overall the penalty is less than the loss in revenue due to canceled reservations. Also, partial results may be given and the full results filled in later. This allows applications to start processing in anticipation of the final results.

2.1 Logistic Application Fault Model

The logistics planning application is expected to run in a hostile environment. The underlying computer resources will be fielded in remote areas with frequent power failures and can be turned off and moved to a new physical location. Communication is likewise unreliable, and may be subject to intermittent connections and long delays. Finally, the equipment may be subject to security attacks, which will deny service to resources.

A major goal of the agent infrastructure is to hide these faults from the application or at least convert the faults into a form that is compatible with the application business logic. As we discuss later (Section 4.5), the Cougaar framework handles most failures transparently, but those failures that it can't handle are seen as rescind operations to the application-layer business logic.

3. Cougaar Application Programming Model

To understand how Cougaar exploits the logistics programming characteristics described in Section 2, one must first review the underlying application programming model. Cougaar uses a blackboard publish and subscribe model as its main coordination medium and communication mechanism. A Cougaar agent is implemented as a collection of plugins that share the same blackboard. Plugins can publish objects onto the blackboard and other plugins can subscribe to changes in these objects. A primary benefit of the blackboard is to abstract out the inter-agent communication message passing mechanism and unify it with the local plugin coordination medium. Plugins can mark blackboard objects for copying to other agent's blackboards. The Cougaar infrastructure handles the transfer of the objects between blackboards. Plugins can be ignorant to whether an object was published by a local plugin or by a plugin on a remote agent. The rest of this section will explain the Cougaar blackboard API.

3.1 Blackboard

The blackboard defines an asynchronous publish/subscribe API with pluggable domain-specific behavior. This frees developers to concentrate on the domain-specific issues of their application. Cougaar blackboards are agent-local to assure scalability. A globally shared blackboard (e.g. Java Spaces or JMS) is a single point of failure and a considerable performance bottleneck. Cougaar blackboards also support transactions, persistence, re-hydration, and dynamic reconciliation. Additional details can be found in the Cougaar Developers Guide. [4,5]

All access to the Blackboard is transaction-controlled. Blackboard transactions group objects into logical collections—Transactional safety is not guaranteed for sub-object changes, only for addition and removal of objects from the Blackboard. In addition, the mechanism that delivers collections of Add and Remove events also delivers Change events (with details), which may be used to track sub-object level changes [6].

3.2 Blackboard Transaction Internals

Plugins do not usually communicate directly with the Blackboard. Instead they are given a proxy object called a *Subscriber*, which manages most of these interactions. This separation of functionality allows Plugin developers to either extend one of several base classes or to write their own Plugin classes from scratch without risk of damaging the delicate interactions between the infrastructure and the Subscriber. Subscribers manage the interaction between Transactions *and* Subscriptions, by bundling the change events for a Plugin. When a Subscriber receives a new collection of changes from the Blackboard, it either queues the changes for later handling (if the Plugin is currently running) or updates the Subscriptions with the changes specified (if the Plugin is idle).

Transactions: A Blackboard Transaction is similar to a traditional database transaction over a collection of reference objects. Blackboard Transactions do not attempt to protect the integrity of internal object state – rather, they only protect the consistency of the set of Blackboard Objects visible at a given time. Essentially, a Blackboard Transaction may be represented as a collection of “add object,” “remove object,” and “change object” functions to be applied atomically to the Blackboard. Rollback is not supported. Pending change events are not visible even to the entity making the changes until the end of the Transaction. Note that it is add/remove/change events, which are transaction-controlled, never the internal state of any blackboard objects. This implies that either blackboard objects should be immutable, or that the application must be certain that only one component may modify and/or examine internal state at a time (e.g., via synchronize). Transactions both protect the Blackboard from asynchronous modification and provide a framework for tracking individual changes made or requested. Each subscriber supports exactly one open transaction at a time. Open Transactions may not be passed between threads: if a Plugin has multiple running Threads of execution, they must either multiplex on a single Subscriber/Transaction/Thread or have separate Subscribers.

Subscriptions: All access to the blackboard is via Subscription objects. A Subscription is logically a

“slice” of the Blackboard as specified by a *Predicate* that selects the objects of interest. In addition, most Subscriptions both track changes to the Subscription’s members since the previous Transaction and maintain a collection of the subscribed elements. Plugins may define their own variations on Subscription to maintain additional sorts of information about the elements or to keep them organized in different ways. Any storage of elements associated with a Subscription is entirely separate from any other Subscription as well as the Blackboard. Subscriptions always manage their own copies of the membership sets. Subscriptions are initialized from the entire Blackboard set and then they are updated only by Transactions.

4. Task/Allocation Coordination Primitive

The Task/Allocation coordination primitive [7,8,9] is a negotiated agreement between two agents. The master agent gives its subordinate a task by publishing the task on its blackboard. The master also creates an allocation object on its blackboard for the subordinate to put its reply. Notice each participant has a unique copy of these objects, which may be slightly out of sync due to communication delays. The local copy of these objects represents the state of the negotiation as seen by the local participant. At any time either of the participants can rescind their objects, and the other side must be able to undo their work. The rest of this section explains the Task/Allocation coordination in more detail.

4.1 Example of Task/Allocation

The typical operation of a Task Allocation is as follows: (Figure 2 is a visual of this process).

1. The Agent receives a Task. An Expander Plugin that has a Blackboard subscription matching the Task is notified.
2. The Expander Plugin processes the incoming Task and generates a Workflow (a set of subtasks that must be accomplished in order to accomplish the original Task).
3. The Expander Plugin creates a PlanElement that contains the incoming Task and the Workflow it created. The Expander Plugin publishes the PlanElement and each new subtask to the Blackboard.
4. The Agent notifies all Plugins that have subscribed to any of the new Blackboard elements (Tasks). In this example, the Allocator Plugin has a subscription matching the new Blackboard element and is notified.
5. The Allocator Plugin allocates assets among all the tasks that have been expanded, both those from the newly created Workflow and the tasks from any previous Workflows.

6. The Allocator Plugin creates Allocation PlanElements, each of which contain a Task and the Asset assigned to perform the Task. An allocation may assign physical assets to accomplish the Task, or it may assign the Task to another Agent. The Allocator Plugin publishes the Plan Elements in the Blackboard.
7. The Agent infrastructure sends Tasks allocated to other Agents to those Agents.
8. The Assessor Plugin reviews the Allocations made by the Allocator Plugin.
9. The Assessor can send Directives to the Agent that originated the Task, if the Allocation score exceeds a specified threshold and action external to this Agent is required.

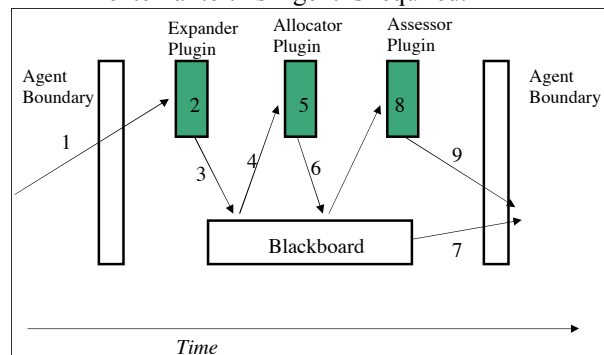


Figure 2. Plugin Operation Flow

4.2 Non-determinism in Task Allocation

It is expected that each Plugin will do its job well, that is, working towards some local minima of the small part of the space they control (giving the job to find the best available airplane, for example). Through this flow of information up and down a processing chain, however, we have many negotiations among different Plugins and Agents to perform a more global optimization over a larger space. By changing task allocation and task preference, an Agent can search for an optimal solution between Agents, or manage relationships with multiple providers to optimally satisfy aggregate requirements. This is a completely asynchronous and non-deterministic approach – there is no consistency of asset state nor ordering of message delivery, which affords Cougaar immense flexibility in data transference, but at the same time requires checks for information assurance (IA). [10] Inherent to the Cougaar Task primitive is a built-in context of persistence, including a concept of “lazy” persistence.

4.3 Innate Persistence in Task Allocation

There exists, within the Task infrastructure of Cougaar, a built-in persistence mechanism, described as “lazy persistence”. This mode is optimistic in that it assumes that restarts will not occur and that the persisted data is not absolutely essential. By making

these two assumptions, the state of the agent can be saved much less frequently. 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 will need to be saved. Lazy mode operates in the same way as non-lazy (or pessimistic) mode except the interactions with other agents are not rigidly locked into step with persistence. Instead, 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 obvious upshot 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.

4.4 Reconciliation Rationale

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 actually can signify two things: the plan has changed and the Task is no longer desired, or 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 responsible for detailing these intermediate states; it need only insure that the slave object have 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. 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.

4.5 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 naming service. This is less than ideal, but suffices for the moment.

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. This optimization is not currently performed.

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 [5,11].

5. Further Reading

The Task/Allocation coordination primitive is a “sweet spot” allowing both non-obtrusive application development and highly survivable infrastructure. Cougaar supports other coordination primitives, specifically “Relays”, which were used to monitor and control the agent infrastructure itself [12]. An open research question for Cougaar is: what other types of coordination primitives can be used effectively by applications and still have efficient run time support? We feel that open-source Cougaar [1] is an effective infrastructure for exploring this issue and we invite other researchers to join the Cougaar community.

Other papers describe how Cougaar manages survivability at multiple levels. At the Java process level, the Cougaar middleware is componentized with extensions to add QoS-adaptive features [13,14]. At the agent level, a control society of agents was created to manage the robustness and security of the Cougaar runtime [6,12,15,16]. At the highest level, a *deconfliction* society resolved conflicts between the constraints imposed by control societies and requirement desired by the application’s mission [17].

6. Conclusions

Success in scaling distributed applications depends upon the survivability of the agent system’s underlying infrastructure in the face of network and application stress. Using simple coordination primitives like the Cougaar Task/Allocation, Cougaar and UltraLog has achieved enormous success in scaling its massive application across 100+ hosts and 1000+ autonomous agents. [3] Embedded into this architecture’s design is a robust hierarchy of control and adaptable systems. Despite the tradeoff of transparency for flexibility, when combined with this survivability infrastructure, these primitives have proven themselves very powerful as application models for creating successful logistics and distributed multi-agent systems.

7. Acknowledgements

This paper builds on designs & development of the Cougaar Agent Architecture. [5,6] The work described

here was sponsored, in part, by DARPA UltraLog contract number #MDA972-01-C-0025. [2] These ideas represent contributions by the many participants in the DARPA ALP and UltraLog programs, and in no way represent the opinions of DARPA

8. References

- [1] Cougaar Agent Architecture, BBN Technologies. <http://www.Cougaar.org/>
- [2] DARPA Ultralog Program Home Page <http://dtsn.darpa.mil/ixo/programdetail.asp?id=64>
- [3] A. Helsinger, R. Lazarus, W. Wright, and J. Zinky, “Tools and Techniques for Performance Measurement of Large Distributed Multiagent Systems,” 2nd International Conference on Autonomous Agents and Multiagent Systems (AAMAS), 2003.
- [4] “Cougaar Developer’s Guide” v. 11.4. http://cougaar.org/docman/view.php/17/171/CDG_11_4.pdf
- [5] “Cougaar Architecture Document” v. 11.4. http://cougaar.org/docman/view.php/17/170/CAD_11_4.pdf
- [6] A. Helsinger, M. Thome, T. Wright. “Cougaar: A Scalable, Distributed Multi-Agent Architecture.” In proceedings of IEEE SMC04 at The Hague.
- [7] P. Scerri, A. Farinelli, S. Okamoto, M. Tambe “Allocating Tasks in Extreme Teams” In proceedings of AAMAS’05, July 25-29, 2005.
- [8] B. Gerkey, M. Matari. “Murdoch: Publish/Subscribe Task Allocation for Heterogeneous Agents.” In proceedings of Agents 2000 Barcelona Spain.
- [9] P. Sander, D. Peleshchuk, B. Grosz. “A Scalable, Distributed Algorithm for Efficient Task Allocation.” In proceedings of AAMAS’02, July 15-19, 2002, Bologna, Italy.
- [10] A. Helsinger, W. Ferguson, R. Lazarus. “Exploring Large-Scale, Distributed System Behavior with a Focus on Information Assurance.” Proceedings of DISCEX II: DARPA Information Survivability Conference and Exposition. Anaheim, CA. 2001
- [11] M. Thome, “Multi-Tier Communication Abstractions for Distributed Multi-Agent Systems,” IEEE Conference on Knowledge-Intensive Multi-Agent Systems, 2003.
- [12] M. Thome, T. Wright, A. Helsinger. “Watching Your Own Back: Self—Managing Agent Systems.” KIMAS 2005 Conference
- [13] J. Zinky, R. Shapiro, S. Siracuse, “Complementary Methods for QoS Adaptation in Component-based Multi-Agent Systems.” IEEE Symposium on Multi-Agent Security and Survivability (MAS&S). 2004
- [14] J. Zinky, R. Shapiro, S. Siracuse, S. Ford, D. Wells, “Case Studies of Node-level QoS Adaptation in Cougaar,” OpenCougaar Conference, 2004
- [15] M. Thome, “Managing Applications Comprised of Untrusted Components,” JavaOne 2002 Proceedings
- [16] A. Helsinger, K. Kleinmann, M. Brinn, “A Framework to Control Emergent Survivability of Multi Agent Systems.” In proceedings of AAMAS, Conference 2004.
- [17] David Wells, Paul Pazandak, Marian Nodine, Anthony Cassandra, “Adaptive Defense Coordination”, IEEE Symposium on Multi-Agent Security and Survivability (MAS&S), 2004.