

Implementing QoS-Adaptation in Coordination Artifacts by Enhancing Cougaar Multi-Agent Middleware

John Zinky, Richard Shapiro, Sarah Siracuse, Todd Wright

BBN Technologies

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

Abstract

Coordination Artifacts are first-class software entities that encapsulate an abstract communication pattern either among Agents or between an Agent and its environment. By separating these Coordination Artifacts from Agent implementations, both are simplified and acquire a nearly parallel structure. Coordination Artifacts provide an ideal means for dealing with systemic issues, including survivability, allowing the remaining Agent code to focus solely on its domain/business logic. Coordination Artifacts can be implemented as a small set of extensions to the existing Cougaar Agent middleware. In this paper, we present a design for Coordination Artifacts and their implementation in Cougaar.

1. Introduction

Coordination Artifacts (CAs) are first-class entities that encapsulate an abstract communication pattern either among Agents or between an Agent and its environment. They are designed to separate communication mechanisms from the domain-specific work of any given Agent[1, 2]. The new objects that encapsulate these communication patterns provide a clear locus for dealing with systemic issues such as security, reliability, and performance. In particular, a Coordination Artifact's inherent properties (specialization, encapsulation, malleability, and controllability [3]) provide the necessary support for handling adaptation to changes in Quality of Service (QoS) requirements or resource constraints. That is, Coordination Artifacts can be *QoS-adaptive* if they change their behavior to overcome resource constraints and meet QoS requirements.

Because systemic concerns can be handled in the Coordination Artifacts, the remaining Agent code can restrict itself to domain issues. Domain *logic* can be represented in the Agent in the form of relationships among the states of the roles it plays in various Coordination Artifacts. Patterns in these relationships can then trigger domain *data processing*, i.e., purely procedural code executed locally. Coordination Artifacts have a similar logical level, but the data processing in this is distributed and must therefore be relatively simple.

In this paper we describe an architecture and a working implementation of Coordination Artifacts that supports *QoS-adaptive* features. We show how Artifacts can adjust their behavior dynamically to meet QoS requirements

within the constraints of the available resources, thereby improving survivability. We will use an existing Agent environment, the Cognitive Agent Architecture (*Cougaar*) [4], to illustrate the construction of Coordination Artifacts. Cougaar has many middleware services that can be combined in various ad hoc ways to implement Coordination-like behavior Artifacts, but has no direct support for Coordination Artifacts as such. We will present a set of extensions to Cougaar that provide that support in an organized way.

The remainder of this paper is outlined as follows. Section 2 defines the key concepts of Coordination Artifacts, augmented by an overview of our extensions. Section 3 presents a useful duality of structure between Coordination Artifacts and Agents. Sections 4-6 describe our implementation of Coordination Artifacts, including an overview of the existing Cougaar architecture and a description of how it can be extended to support Coordination Artifacts. Section 7 looks at Coordination Artifacts from a life-cycle perspective, and examines the kinds of QoS-adaptation that can be added to each phase. Section 8 looks at Coordination Artifacts from the environmental perspective. The final four sections fill out the picture: an example (9), a brief analysis of the overhead introduced by CAs (10), related work (11), and conclusions (12).

2. Definition of the Coordination Artifact

Coordination Artifacts are infrastructure abstractions that support Objective communications. As described in existing literature, Coordination Artifacts have these key features: specialization, encapsulation, malleability, and controllability -- features that are foreign (and sometime contrary) to Agents[3]. "The most obvious embodiment of the notion of artifact for Agent coordination is represented by a dedicated abstraction, provided at design time by the coordination model, and enacted at runtime by the corresponding coordination infrastructure." [5] Coordination Artifacts are therefore first-class entities on a par with Agents.

In the basic model, a Coordination Artifact is characterized by having (i) a set of interfaces which can be accessed by Agents (ii) a local shared state between the Coordination Artifact and an Agent (iii) a coordination behavior specification, in which to describe the artifact's formal behavior[3]. Figure 2 illustrates the structure of a Coordination Artifact.

We extend this model slightly by assuming that Coordination Artifacts explicitly define a set Roles, where a Role is simply a well-defined interface to the Artifact. The notion of a Role makes the abstract conception of “operating instructions” more concrete: a client of a Coordination Artifact always plays a specific Role relative to that Artifact, which the client indicates when it binds to the Artifact. Clients, in other words, are *Role Players*. Each Role in any given Artifact can potentially be played by any number of clients. With this extension, the behavior of a Coordination Artifact can be described as consisting of *actions affecting the shared state among its Roles*. Borrowing some terminology, a *Facet* of a Coordination Artifact is the code that implements a single connection to a given Role, and a *Receptacle* is the client’s interface to that facet.

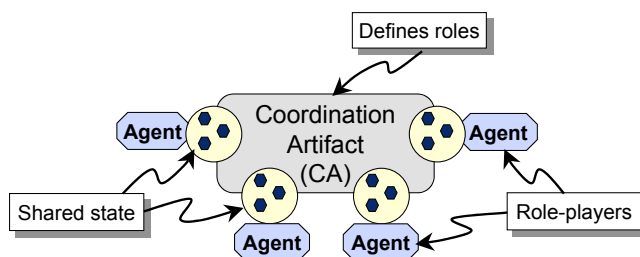


Figure 2: Coordination Artifact Characteristics

The primary advantage of treating Coordination Artifacts as first-class entities is that the instances provide an ideal site for dealing with systemic issues in communications. Extending Coordination Artifacts to make them *QoS-adaptive* enables them to react to the constraints imposed by the physical environment. This may require specializing a generic coordination pattern into a customized component. This context-specific Coordination Artifact uses *specialized knowledge* about the expected coordination behavior and *run-time knowledge* about the resource constraints to pick the best implementation strategy to meet the situation.

In general, Coordination Artifacts are designed to operate across Agents to implement a specific systemic concern. This is analogous to a *cross-cut* in Aspect Oriented Programming. Just as an Aspect cross-cuts the dominant Object Oriented decomposition, so an Artifact cross-cuts the Agent “decomposition” of a society. The resulting advantages are likewise analogous to the advantages of dealing with system issues via AOP[6].

An Agent can use Coordination Artifacts to interact with other Agents or with the environment itself. As such, the Agent becomes a hub of activity for processing information presented to and extracted from Coordination Artifacts. The Agent does not have to worry about the mechanics of coordination, which allows it to concentrate on its domain processing. Figure 3 shows an Agent’s various Coordination Artifacts.

3. Dual Nature of Coordination Rules

The introduction of Coordination Artifacts creates a duality within the society. Once the Artifact’s coordination logic is abstracted from the Agent code, the Agent’s business/domain logic can also be described as *actions affecting the shared state among its Roles*. In this case, the Roles are those played by the Agent in each of the Artifacts to which it is connected. We refer to this abstract behavior as *Coordination Rules*. Both Coordination Artifacts and Agents therefore implement behavior that can be represented in Coordination Rules.

The coordinating logic of an abstract set of Coordination Rules primarily involves dependencies between Roles and actions to be taken when particular collections of dependencies are active at any given time. Declarative rule languages are ideally suited to this form of expression. The use of a rule language has several advantages. First and foremost, rules expose the logical dependency structure much more clearly than a procedural language. They present sets of dependencies together instead of distributing them through procedural (if/then) statements. Also, the fact-based pattern matching used in rule systems does not, in general, care about the order in which facts are asserted. This relieves the programmer of a considerable bookkeeping chore when the exact order in which new information enters the system is not known *a priori*. Rule engines also have an intrinsic modularity that can be very useful in the context of dynamic adaptation: the behavior of a collection of Rules can be modified dynamically by adding Rules, without any need to modify existing ones. Finally, Rules have useful formal characteristics, although we have not taken advantage of them to date: they can be modeled, analyzed offline, and used to generate efficient runtime code[7].

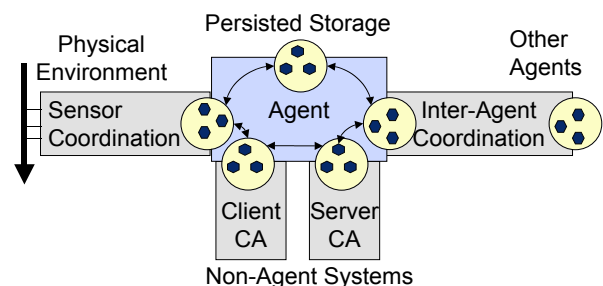


Figure 3 Agent Processing Data from Coordination Artifacts

While a collection of Rules can do a good job at expressing coordination logic, Rules are not well suited to complex data processing, which is a significant part of an Agent’s workload as well. A selected Rule language must therefore interoperate easily with whichever conventional programming language is used for the raw processing. In the case of Cougaar, the rule language must be callable from and able to call into Java. In our prototype implementation, we use Jess[8] as our Coordination Rule language. In principle however, any suitable rule language would work. An example of this kind of interoperability between Jess and Java, in which

abstract logic written in Jess can call out to data processing written in Java, can be seen in the example below in section 9.

So far we have emphasized the structural similarities between two kinds of Coordination Rules, but there are important differences in their implementations. An Agent is a *locally self-contained object*. It might have to do arbitrarily complex domain processing, but all of this processing is co-resident. Agents do not, in general, have to worry about QoS requirements and other systemic issues for their internal data processing. Conversely, Coordination Artifacts are *distributed entities*. Data processing in a Coordination Artifact is relatively simple because this processing deals almost exclusively with data transfer and translation. At the same time, this processing is heavily affected by systemic issues, including quality of service concerns. The ability to adapt to changes in QoS requirements becomes increasingly important, not only to the effectiveness of the Artifact, but to its performance as the society scales to hundreds or thousands of Agents.

These differences and similarities have some implications on the organization of Agent code when the infrastructure supports Coordination Artifacts. Agents will now primarily be designed as interrelated sets of coordinations among the various Roles they play. In Cougaar and other systems that use Blackboards, this in turn has implications for the Agent's style of interaction with its Blackboard. The entities on the Blackboard can now be partitioned in fact-bases, with one fact-base per Role. The fundamental logic of any given Agent then becomes a collection of Rules that coordinate among the fact-bases representing Role-specific subsets of Blackboard entities. In other words, Agent logic becomes a matter of *detecting patterns* across multiple Roles and their associated Artifacts, and *triggering domain specific actions* based on those pattern matches. An example of this kind of coordination can be seen below.

4. Existing Cougaar Infrastructure

Taking a step sideways, this section will review those parts of the Cougaar middleware infrastructure that can be used in the construction of Coordination Artifacts. The following section will describe more specifically how the architecture described above was realized in Cougaar.

Cougaar has been used in the construction of large-scale distributed Agent-based systems[4, 9, 10]. Cougaar provides a comprehensive infrastructure for developing robust societies of distributed Agents, with support for security and scalability built in. Several kinds of components in Cougaar are useful as building blocks for Coordination Artifacts. More specifically, these include Blackboard-based data representation and message routers[6].

Cougaar Agents communicate via a publish-and-subscribe mechanism, implemented in each Agent as a membership-transactional Blackboard with predicate-based publish/subscribe semantics. Logically, the Blackboard is

an arbitrary collection of objects. Communication Plugins called *Logic Providers* subscribe to the Blackboard and look for objects that should be transferred to other Agents' Blackboards. Logic Providers come in many types, each with domain-specific behavior for moving objects between Blackboards. For example, a simple *relay* Logic Provider might merely copy an object to all Agents in a Community. The *task/allocation* Logic Provider will request an allocation from a remote Agent, and also predict the allocation before it officially arrives. This allows the allocation to be rescinded, allowing for recovery if either Agent fails.

Logic providers use a Cougaar service called Message Transport Service (MTS) to send messages between Agents. The MTS takes care of all the communications' details, and Logic Providers treat the MTS as a reliable message transport. The Metrics Service monitors the underlying computer, storage, and communication resources. For example when special network management support is available, the Metrics Service can determine whether a communication path is available to a remote Agent without sending test messages. The MTS adjusts the behavior of its protocols based on the resource constraints exposed by the Metrics Service.

Cougaar is component-based middleware closely modeled on Java Beans Bean Context API[11]. It supports Service Oriented Architecture: in general; inter-component communication happens via *services*. The implementation of a service is provided by a *Service Provider*, which registers its services with a *Service Broker* at any level of the component hierarchy.

Cougaar adds layers of insulation to this Service Oriented Architecture (SOA) in two ways. Binders between parent and child Components control which services are offered to the child, and in what form they're offered. In addition, Service Proxies can be introduced between server and client objects. These proxies can be simple forwarders, or can implement arbitrarily complex adaptive behavior.

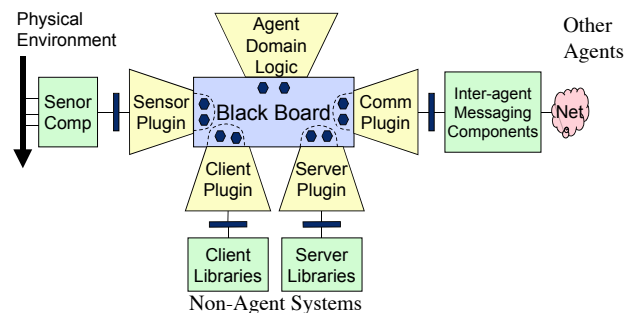


Figure 4: Cougaar Component Model

QoS-adaptation happens at each component layer. For example, the Blackboard object can be tagged with a QoS requirement for a lifespan. If a Logic Provider cannot send a message to the remote Agent within the lifespan, the object is removed from the Blackboard and the message is flushed from the message transport. Cougaar infrastructure has many other components to help implement QoS-adaptation[9]. Some common examples follow. The Cougaar *service discovery* mechanism is a mechanism for finding an Agent that offers a service. Through service discovery, other services can be contracted upon the request of any Agent with access to the service. The distributed *Yellow Pages Service* (YP) in Cougaar helps to prevent bottlenecks and to balance workload. An Agent's information can be contained in multiple YP Servers for retrieval later. The *Metrics Service* provides information regarding resource consumption and performance measures. The Metrics Service especially provides necessary systemic information to be used by the sensors in a QoS-adaptive Coordination Artifact, potentially initiating some level of control over its data

Figure 4 shows how Plugins can be composed into an *ad hoc* Coordinations using these mechanisms. The Blackboard fulfills the function of the "pool of facts" for each Role. Blackboard objects can be tagged with a Role and a Community. This effectively partitions the Blackboard into pools associated with each logical Coordination. The Plugins convert the underlying infrastructure components from an SOA model into a Blackboard model. The infrastructure components interface to the non-Agent system, the physical environment, or other Agents. These components form a kind of connectionless Coordination, since they can be generated and removed dynamically, and they are not explicitly named.

Ad hoc Coordinations of the kind described above have been shown to work in large control societies created for the Ultralog[10] system. The Ultralog project created control societies for managing the security and robustness of a large distributed logistics application, with over 1000 Agents running on over 100 hosts. In one of the evaluations of Ultralog, testers removed 45% of its computer and networking resources. Despite this loss, the control society was able to reconstruct the failed logistics society Agents, and the society continued to process the application. Part of the success of creating a robust logistics society was due to the use of the distributed Blackboard for communication between Agents[12].

While ad hoc Coordinations can be made to work directly on existing Cougaar infrastructure, true Coordination Artifacts, with all the advantages they offer, require extensions. In the next section we will discuss what needs to be added to Cougaar to provide this support.

5. Extending Cougaar Infrastructure to Implement the Architecture

In the previous section we looked at some parts of Cougaar that could be assembled into *ad hoc* Coordinations. Of course Coordinations constructed in this way are not first-class entities and do not realize all the potential that Artifacts have.

To implement true Coordination Artifacts as a new layer of the Cougaar middleware turns out to be a surprisingly simple procedure. The architecture needs to touch the Cougaar infrastructure only at two points. First, we need a new service to register new Artifacts and find available ones – in short, a *Coordination Artifact Broker*. Second, the facets of each Artifact need to translate between Blackboard objects and Rule-engine objects (Facts) in a coherent way (Figure 5). Implementing the Coordination Artifact Broker is obviously trivial. The second point touches at the heart of how Artifacts work in Cougaar: in a very real sense, the procedural code implementing the Facet classes for any given Artifact is all the Java code an Artifact developer needs to write (keeping in mind that the logic of the Coordination itself is expressed in a rule language like Jess).

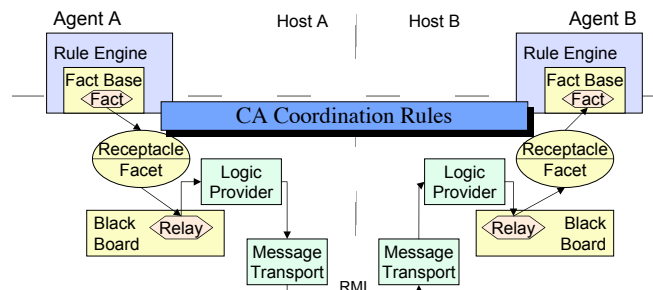


Figure 5: New and Existing Layers

To support this form of development, Role Player entities, which in Cougaar are typically Plugins, use a formulaic body of code for each Blackboard execution cycle: allow each Facet (through its corresponding Receptacle) to deal with changes to the Blackboard (this would typically result in corresponding fact changes); run the rule engine (this could result in changes to another Player's Blackboard); allow each Facet (through its Receptacle) to deal with new state of the fact set.

Two general sorts of issues arise immediately. First, it is important to keep in mind that an Artifact instance is inherently a distributed object. A single conceptual Artifact is implemented as a collection of Java instances, one for each Agent that plays a role in it. This collection of Java instances is loosely coupled through *Facets*, and this distributed collection of Facets need to communicate with one another in a loosely coupled way. Cougaar *Relays* provide the support for this style of communication.

Secondly, inheritance of Coordination Artifact classes introduces some wrinkles both in the inheritance of procedural code (Facet implementations in particular) and

the inheritance of Rules. In general, the hierarchy of Coordination Artifact kinds has to be mirrored by a similar hierarchy of Facet kinds, although not all Artifact layers will introduce new Facet classes for every Role. Although somewhat inelegant, this presents no conceptual difficulties. In possible future developments we will consider generating skeleton code here.

By convention, the Rules at any given layer should be grouped together in a file. The Artifact or Role Player can then load these files dynamically. Using Rule files in this way provides a simple form of multiple inheritance, although care must be taken with firing order.

The ability to load any number of Rule files offers an opportunity for dynamic reconfiguration not unlike Aspects: newly loaded files of Rules can transparently modify existing behavior at the new layer. Other forms of cross-cutting can of course also be used at lower layers, through existing Cougaar mechanisms[6].

6. Coordination Artifact Architecture

In our current implementation of Coordination Artifacts, the top-level structure is isomorphic to Cougaar's service-oriented-architecture (SOA) design:

- *Coordination Artifacts* are roughly analogous to Services in SOA. But unlike the generic Service interface, extensions of Coordination Artifacts (CAs) are used only to describe particular kinds of Artifacts, not in general to define extensions to the generic interface. In this sense Coordination Artifacts are also very roughly analogous to Components in the CORBA Component Model

- *Coordination Artifact Providers* describe entities that can find or make Coordination Artifacts of some particular kind. They are very similar to Service Providers in SOA.

- *Coordination Artifact Brokers* are registries for Coordination Artifact Providers. These registries are therefore very much like Service Brokers in SOA.

In addition, the CA design defines three other generic interfaces of interest:

- *Role Player*: Any given Coordination Artifact kind (as managed by a Coordination Artifact Provider) defines a set of Roles, and any client that wishes to connect to an Artifact must specify which Role it intends to play. All clients must therefore implement the Role Player interface, which describes callbacks that the CA might wish to invoke on a client, for instance to assert or retract facts.

- *Receptacle*: A Role Player's access to a CA is mediated by an entity called a *Receptacle*, a term borrowed from the CORBA Component Model (CCM). A Receptacle is the Role Player's perspective on the connection between it and some CA.

- *Facet*: The CA's access to a Role Player is mediated by an entity called a *Facet*, a term also borrowed from CCM. A Facet is the CA's perspective on the connection between it and some Role Player. Facets and Receptacles are always paired one to one. Facets are also the point at which inter-Agent communication

occurs: a single conceptual Coordination Artifact is actually a distributed object, the pieces of which are linked together by Facets. The implication of this is that most of the real communications work of a Coordination Artifact is implemented in Facets.

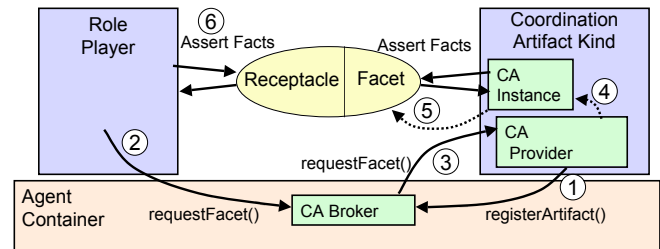


Figure 5: Coordination Artifact Components

The standard protocol for registering a new Coordination Artifact Provider would then be as follows (see Figure 5):

- The Provider looks up the Coordination Artifact Broker as an SOA service and registers itself as being available for use. (Fig 5, 1)

The standard protocol for requesting a connection to a Coordination Artifacts is:

- A Role Player looks up the Coordination Artifact Broker as an SOA service and requests to play a particular Role in a particular Coordination Artifact, where the latter is described by a type name and some qualifiers, possibly including QoS qualifiers. (Fig 5, 2)

- The Coordination Artifact Broker looks for a Provider that supports the given type. (Fig 5, 3)

- That Provider looks for an existing Coordination Artifact that matches the qualifiers, or creates one if none exists yet. (Fig 5, 4)

- The Coordination Artifact creates a Facet/Receptacle pair for the new connection and informs the Role Player of the Receptacle. (Fig 5, 5)

- At this point the Role Player is free to assert facts into the CA through the Receptacle; the Facets are free to communicate between one another; and the Facets are free to assert facts into a Role Player through the Receptacle. (Fig 5, 6)

The use of a rule language implies the existence of one or more rule engines. Where do these engines live? In the current implementation, each Role Player has its own engine. We locate the engine there because the coordination dependency logic in an Agent requires the ability to handle Rules in which the left hand side can refer to multiple Coordination Artifacts. By contrast, the inter-Agent dependencies inside an Artifact do not have this requirement, and in general cannot have it, because the Agents often reside in other JVMs.

The implications of locating the engine in the Role Player Agent are as follows:

- The Coordination Artifacts must be able to assert, retract, and modify Rules in that engine. For this reason, the Role Player interface supports these methods.

- Role Players must likewise be able to assert, retract, and modify Rules in another Player's engine, and

it can only do this through the Artifact. The Receptacle interface therefore must support the same three methods. The duality now becomes particularly obvious.

- The key job of the Artifact now stands out clearly: it must be able to *move* facts from one engine to another, in a transactional way.

- The Role Player Rules need to be able to distinguish the *origin* of any given fact, *i.e.*, which specific Role of all the Roles played by this Player asserted it. Put another way, the collection of Facts in the engine at any given time can be partitioned into equivalence classes, or *fact-bases*, of one per Role. We do this in the current implementation by explicitly defining a `fact-base` Fact type, asserting a `fact-base` fact for each Role, and tagging every other fact with a slot whose value is a fact-base Fact.

7. Coordination Artifact Life-Cycle

So far we have been looking at Coordination Artifacts structurally. In this section we look at them from a life-cycle perspective, with the following epochs: specification, implementation, construction, Role-binding, invocation, Role unbinding, and destruction. An explicit life-cycle gives Coordination Artifacts the hooks for adding QoS-adaptation. We will briefly describe the types of QoS-adaptation that are appropriate for each stage of the life cycle.

Specification time: In our architecture, the specification of an Artifact consists of its Roles and its identifying parameters. Roles are concerned with the underlying logic of the coordination. Parameters are in a general way concerned with distinguishing any pair of Artifacts of the same kind. Parameters are a natural place to specific dynamically determined QoS requirements. At Role-binding time, the Artifact Provider can then use these requirements to construct a suitably configured Artifact instance (this remains for future work).

Implementation time: The bulk of the implementation of a Coordination Artifact type consists of two parts. The coordination logic is implemented in a Rule language, *e.g.*, Jess. This describes the relationships between the Artifact's Roles. The Cougaar infrastructure linkage is implemented in Java in the Facet code, and is generally concerned with the translation between Blackboard objects and Facts. In future work, Facet code might be partially generated from an abstract specification, something like the way the CORBA IDL compiler generates stubs and skeletons from abstract interface descriptions.

The core implementation of the Coordination Artifact class itself is generally very simple: it merely needs to create the right sort of Facet for each Role. The implementation of the Coordination Artifact Provider is the point at which support for QoS-adaptation re-emerges. One of the key features of QoS-adaptive code is the existence of multiple ways of undertaking a task, with each having different resource requirements and QoS properties. The Provider must add code for selecting the best choice based on the state of the underlying resources.

Construction time: Coordination Artifact Providers are created in two ways. Some Coordination Artifacts are loaded as Plugins and are therefore created in the same as any other Plugins. In the other case, some of the standard system/environment Artifact Providers are *created as part of the Artifact* Broker, and can therefore always be assumed to be available. In either case, when a Provider is created it must register itself with the Broker.

Role-bind time: Agents must find a Coordination Artifact and bind to a Role. During binding, the given Artifact Provider must find or make an Artifact matching the given QoS-requirements. Once a suitable Coordination Artifact exists, it binds a Facet/Receptacle pair of the right kind to the requesting Role player. At this point the connection between the Artifact and the client is visible from two perspectives: the Facet represents the Artifact's perspective on the client, and the Receptacle represents the client's perspective on the Artifact. This connection point allows Facts to pass between the Artifact and the client, and gives the Artifact limited access to the client's Blackboard and rule engine.

Invocation time: Once binding is in place, the Agent interacts with the Coordination Artifact by asserting and retracting facts for its Role and by executing Facet methods (through the Receptacle) in a Blackboard transaction. Support for QoS adaptation can also come into play here by encoding QoS requirements, for example lifespan, in a fact's slots. The Coordination Artifacts can use these QoS requirements to make tradeoff decisions for transferring facts between Roles. Groups of facts can be added in a transaction to maintain consistency or to help bundle facts for efficient transfer. High-level services can also be performed when facts are inserted. For example, facts can be translated to a different ontology or data structure based on the destination Role. Finally, if a remote Agent fails, the Coordination Artifact must reconcile the loss of a Role with the other Agents.

Role-unbind time: When an Agent no longer needs to play a Role it can unbind itself from the Coordination Artifact. The Role's fact-base is removed from the Agent Blackboard. Unbinding may also cause facts to be changed or removed in other Roles' rule engines.

Destruction time: When all Agent Roles have been unbound, the Coordination Artifact should be removed from the society.

8 Coordination Artifacts as a Unifying Interface to the Environment

Interaction with the physical environment is a job traditional Agent middleware has left to the Agents themselves, making the workload of these Agents potentially cumbersome. To alleviate this burden Coordination Artifacts should not only perform mediation between Agents but also interface to the underlying physical environment. A physical system may have existing code libraries that give access to some feature of the physical environment. When the library is implemented using a component-based system such as

Java Beans[11], the services are managed through the use of a Service Oriented Architecture (SOA). While the SOA approach offers an advantage over traditional language-based libraries, the services are often still too low-level to be used effectively by Agents, for example because of threading issues involving synchronous calls versus asynchronous callbacks. Therefore an additional layer is needed to convert from SOA services to the software style used to implement the Agent, such as a Blackboard interface. The interface to the physical environment is easily encapsulated via Coordination Artifacts, and consists of three layers: the physical environment itself, the existing interface libraries, and the infrastructure “glue”. Figure 6 shows how Coordination Artifacts can be implemented to interface between various kinds of external environments.

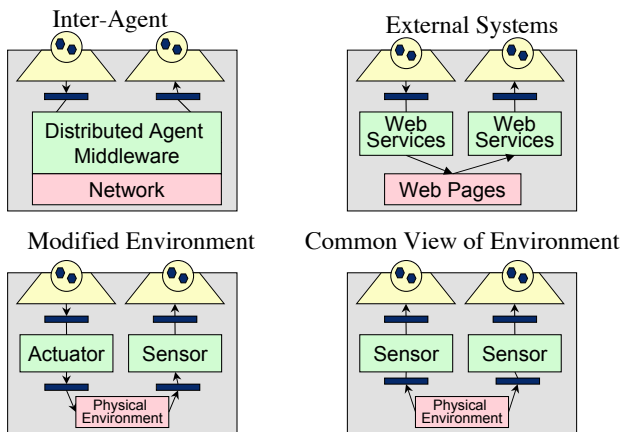


Figure 6: Diversity of Coordination Artifacts

Some examples of physical-layer services that are relevant to QoS-adaptation are: setting Diffserv code points to increase the predictability of communication; interfacing to network management systems (e.g., Hewlett Packard OpenView™) to get resource configurations; and reserving time slices on real-time hosts.

Adding survivability features to a society like this could happen at several layers. Using QoS-adaptive techniques for components[6], Aspects could be added to the Message Transport Service to drop messages. This is an example of the use of aspect-oriented programming (AOP) techniques within the Cougaar infrastructure itself. At another layer, the Facet implementation in one of the Coordination Artifacts could be specialized to support timeouts in its use of Relays. This is an example of classical object-oriented programming (OOP) in the context of Coordination Artifacts. At yet another layer, additional Rule sets could be loaded into the rule engines to alter the logic of the Artifact. This is an example of an implicit characteristic of Rule systems: new Rules can be thrown in to the mix at any time to alter the total behavior of the system without any need to modify existing Rules.

9. Time Sync and Traffic Matrix Example

As prototypical examples, two kinds of Coordination Artifacts were implemented: a wrapper for the Cougaar Alarm Service and a generic multicast query-response.

The alarm Artifact is an example of the use of CA methodology to encapsulate communication between an Agent and the computing environment. The Alarm Artifact allows the use of time-based Rule logic: activity can be enabled only at given times, or delayed for a given time; in either case expressed either as relative offsets or as absolute time. The Alarm CA supports one role, known as *sleeper*. A Sleeper asserts an Alarm, the state of which changes from *sleeping* to *expired* at a specified time. An Alarm CA does not contribute any domain logic directly; rather, if an alarm expires when other domain conditions are satisfied, domain work is triggered. As is generally the case with rule systems, the exact order of these events is not relevant: no bookkeeping is required on the designer’s part to keep track of these various events and conditions.

The second prototypical CA is an abstract query-response multicast, in which each query can generate any number of responses. The query-response abstraction was further extended by two specific forms of query, *TimeSync* and *TrafficMatrix*.

The generic Query/Response Coordination Artifact is concerned with propagating queries to designated members of a Community, and gathering the responses to each query. The content of the query and the response are left indeterminate. The Roles of such an Artifact are clear: those of *requestor* and *responder*. The Facets for these Roles use standard Cougaar Community mechanisms to create relays of the right kind. These relays transmit the query and response facts back and forth. Finally, the rules are very simple: when a query fact appears in a requestor Role, that fact is propagated to all responders; when a response fact appears in a responder Role, that fact is propagated back to the requestor.

Specific kinds of Query/Response handle the next layer. In particular the Rules at this layer must translate between generic query and response facts into more specific varieties. Finally, the client Role Players define Rules of their own: how to respond to a query request (i.e., which response fact to generate) and how to process a completed query.

The TimeSync Coordination Artifact implements a simple barrier synchronization. The TimeSync CA specializes the generic Query/Response in several ways. First, it must determine in advance of posting a query how many responses it expects. Second, it waits until that many responses have come in before the query is considered to be finished. Finally, the content of the query and response are empty, since the point of the query is simply synchronization (ordinarily, queries and responses would consist of domain-specific data).

The TrafficMatrix Coordination Artifact merges locally gathered inter-Agent message statistics into a global matrix. In this case, the CA specializes the generic

Query/Response primarily by means of content: the query content is empty, since the CA itself implies the query. The response content is domain-specific: the traffic data from the responding Agent. Unlike the TimeSync CA, in which a query finishes when enough responses have been received, a TrafficMatrix query finishes after a predetermined length of time; late responses are ignored. This implies the use of a supporting Alarm artifact.

As an example of an Agent coordinating three Coordination Artifacts, we implemented a prototype that uses the TimeSync and TrafficMatrix Coordination Artifacts together with a supporting Alarm. In this example, the domain logic includes four Roles and two Coordination Artifacts, each with its own Cougaar Community. One Agent simultaneously plays two of those Roles, coordinating between them (see Figure 7). This Agent plays the *responder* Role on a TimeSync Artifact and the *requestor* Role in a TrafficMatrix Artifact; whenever it receives a TimeSync query it generates a TrafficMatrix query. Jess rules for this coordination appear below in Figure 8. The support logic includes an Alarm and its associated Role.

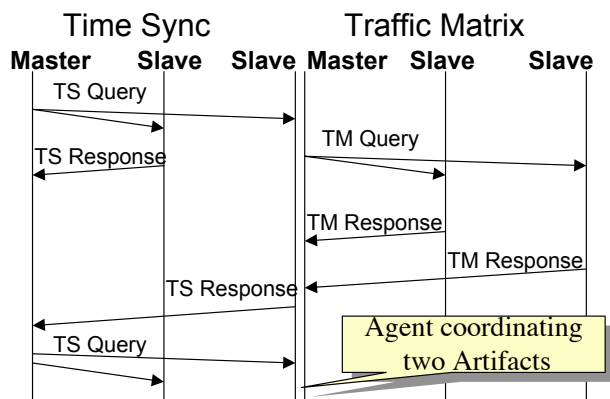


Figure 7: Sequence Diagram of an Agent as Coordination

There are several interesting points to note:

Fact bases: Earlier we described the partitioning of collections of facts into Fact Bases. Such a partition is represented in Jess as a fact-base Fact. The existence of such a Fact implies not only that the Agent running this code plays a role in some CA, but also that the connection to the CA has been successfully established. This provides a very convenient mechanism for dividing the Rule space into logical parts and running the Rules for each part only when appropriate.

Call-outs for domain processing: At two points in the rule logic, once on the query side and once on the response side, the Agent must do some more-involved computation. Rather than implement this computation awkwardly in a rule language, we rely on tight integration between Rules and Java.

Alarms: The example in Figure 8 also shows the use of Alarm artifacts to link the TimeSync and TrafficMatrix. When an Agent playing the Responder role in a TimeSync CA receives a TimeSync Query, it will

generate a TrafficMatrix Query, swap the rate matrix, and set an alarm for 10 seconds (first Rule). As the TrafficMatrix Responses arrive, they are processed (third Rule). When the alarm expires, the TrafficMatrix Query is considered done and a TimeSync Response is sent back to the TimeSync master (second Rule)

10. Overhead Analysis

Adding another layer to the Cougaar infrastructure to support Coordination Artifacts increases the overhead for inter-Agent interactions. This overhead occurs mainly in interfacing the Cougaar Blackboard with the new Agent rule engine and running the rule engine itself. Other processing needed to perform the Coordination is implemented in the same way that the *ad hoc* Coordination was performed without Coordination Artifacts. Note that binding to the CA does not have a big impact on performance, because binding is done once for the duration that an Agent is acting in a Role.

```
(defrule traffic-matrix-process-timesync-query
  "Assert a TrafficMatrixQuery in a response to the receipt of a TimeSyncQuery"
  (declare (auto-focus TRUE))
  (TIMESYNC::TimeSyncQuery (fact-base ?ts-fb) (tick ?tick))
  ?ts-fb <- (CA::fact-base (kind ?*TimeSyncCA*) (role ?*ResponseRole*))
  ?tm-fb <- (CA::fact-base (kind ?*TrafficMatrixCA*) (role ?*QueryRole*))
  ?alarm-fb <- (CA::fact-base (kind ?*AlarmCA*) (role ?*SleeperRole*))
  =>
  (swap-rate-matrix)
  (assert (TMATRIX::TrafficMatrixQuery (fact-base ?tm-fb) (query-id ?tick)))
  (assert (ALARM::alarm (id ?tick) (fact-base ?alarm-fb) (setter ?tm-fb)
    (millis 10000))))

(defrule traffic-matrix-alarm-ring
  "Traffic Matrix alarm went off, respond to the (only?) TimeSyncQuery"
  (declare (auto-focus TRUE))
  ?alarm <- (ALARM::alarm (id ?tick) (fact-base ?alarm-fb) (setter ?tm-fb)
    (expired TRUE))
  ?old-tm-query <- (TMATRIX::TrafficMatrixQuery (fact-base ?tm-fb))
  ?ts-query <- (TIMESYNC::TimeSyncQuery (fact-base ?ts-fb) (tick ?tick))
  ?ts-fb <- (CA::fact-base (kind ?*TimeSyncCA*) (role ?*ResponseRole*))
  ?tm-fb <- (CA::fact-base (kind ?*TrafficMatrixCA*) (role ?*QueryRole*))
  ?alarm-fb <- (CA::fact-base (kind ?*AlarmCA*) (role ?*SleeperRole*))
  =>
  (retract ?alarm)
  (retract ?old-tm-query)
  (retract ?ts-query)
  (assert (TIMESYNC::TimeSyncResponse (fact-base ?ts-fb) (tick ?tick))))

(defrule process-traffic-matrix-response
  "Handle a response to a TrafficMatrixQuery"
  (declare (auto-focus TRUE))
  ?tm-response <- (TMATRIX::TrafficMatrixResponse (fact-base ?fb)
    (source ?source)
    (response ?response))
  ?fb <- (CA::fact-base (kind ?*TrafficMatrixCA*) (role ?*QueryRole*))
  =>
  (receive-traffic-matrix ?source ?response)
  (retract ?tm-response))
```

Figure 8: Rules for TrafficMatrix Master

Table 1 shows the results of running a simple query/response coordination using various society configurations. These measurements were performed on the prototype implementation of Coordination Artifacts, which has not been tuned for performance. The hosts were 2.6GHz, single processor Pentium 4's with 2 GB of memory, running Linux. The network connection was 100Mbps Ethernet. The Cougaar version was B12_0 alpha.

The columns represent various types of Coordinations. *No CA* is the case where the Coordination was

implemented in the traditional *ad hoc* manner, *i.e.*, without the new CA layer. *With CA* is the case where the new CA layer was used to implement coordination. *Multicast CA* is the case where the Cougar Community service was used to send the query to a Community of responders, though for this experiment only one responder was used.

The rows in Table 1 represent various computing environment configurations. *1-host 1-node* is the case where both Agents were on the same host and in the same Java VM. This removes the biggest source of overhead, which is serializing inter-Agent messages. *1-host 2-node* is the case where the Agents are on the same host but in different Java VMs. *2-host 2-node* is where the Agents are on different hosts. Notice that the 2-hosts case has a higher query throughput, because some of the query processing can be done in parallel. The *No CA* column is consistent with previous Cougar performance measurements[13].

	No CA	With CA	Multicast CA
1-host 1-node	1007	530	432
1-host 2 nodes	155	108	104
2-hosts 2-nodes	198	137	130

Table 1: Inter-Agent Query/Responses for Various Society Configurations (Queries/Second)

As expected the CA overhead affects the 1-node case (50% decrease) more than the 2-node case (30% decrease). This is because the CA overhead is large relative to other Cougar overhead. But as the 2-node case shows, the CA overhead is small relative to the overhead of serializing inter-Agent messages. We have not made measurements to determine the source of the CA overhead, but we do expect the overhead to lessen if we implement new logic providers that directly support Coordination Artifacts.

11. Related Work in Coordination Artifacts

Other projects have also used the notion of Coordination Artifacts to implement Agent-to-Agent communications, but their implementations fall short of full support of QoS-adaptation. TuCSoN's Coordination Artifact system is a proposed extension to the FIPA inter-Agent communication standard[14]. TuSCoN [15, 16] is based on the use of *tuple-centers*[17]. TuSCoN is a descendant of Linda, which made popular the distributed *Tuple-spaces* paradigm[18]. Tuples are typed structures with a vector of typed data fields. Many distributed clients can add, request, or remove tuples from a logically centralized tuple space. TuSCoN extends this model into a *tuple-center*, which adds Prolog-like Rules to process tuples as they are added and removed from a center. The Rules become the behavior for the center.

Some research is available on how to add QoS-adaptation to Linda-like tuple-spaces. Limbo[19] adds two extensions to Linda to help QoS-Adaptation for mobile applications: (i) tuples and tuple requests are

augmented with QoS attributes, such as time-outs, (ii) the global tuple-space is partitioned into multiple-named tuple-spaces that are restricted to a limited type of tuple. TSpaces[20] implements a standard centralized tuple-space, but it uses robust message-based middleware as its implementation infrastructure. Lime[21] adds a new operation called *reactor* that is triggered asynchronously based on the state of the tuple-space.

Coordination Artifacts should include all these features and more. To better support QoS-adaptation, we have extended Coordination Artifacts to include:

- *Addition of facts and Roles at design time*, which limits the kind of data that can be asserted to a Coordination Artifact and the relationships between the Roles being coordinated.
- *High-level services*, including data structure translation (future work).

Addition of facts and Roles at design time: We introduced two extensions at the interface between Coordination Artifacts and their clients. *Roles* are a typed portal into an artifact that assigns its operations to a specific part of the coordination. *Facts* add types and named slots to tuples to define the QoS requirements for handling the tuple. In this design, the artifact's tuple center is further organized and divided into a *fact-base* per Role, providing control over data flows between Roles played by Agents. Figure 9 illustrates this partitioning.

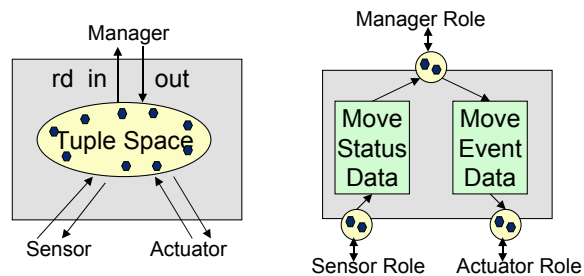


Figure 9: Tuple-Space vs. Fact-Role Implementations

The advantage of partitioning an Artifact's tuple space into fact-bases for each Role is threefold. First, the complexity of instructions and operations are reduced to focus on a specific type of Agent interaction with the Coordination Artifact. This interaction can have explicit QoS requirements associated with both the Role and the facts. Second, a Role's fact-base is local to the client Agent playing that Role, so the direct interactions with the Coordination Artifact are not burdened with typical distributed system issues. In some sense, the local fact-base is a cache or proxy for the Coordination Artifact. Third, the Coordination Artifact can implement the coordination behavior by controlling the data flow among fact-bases. Since these data flows are distributed, and hence susceptible to physical constraints, they can have custom implementations using standard QoS-adaptive middleware [22, 23].

The main difference between a fact-Role implementation and the TuCSoN tuple-center approach is

in how the data is moved between Agents. Both use a high-level language to define the behavior of the coordination, but in the fact-Role approach, the actions are specified as the composition of pre-existing components implemented in the Agent middleware. The underlying components can have robust implementations that include pragmatic handling of error conditions. Note that both schemes could have the same external interface to Roles, *i.e.* the restricted fact-bases. The main difference is in the implementation of the plumbing between fact-bases.

High-level services, including data structure translation[24]: This feature stems from the communication function of an artifact, which is to move data from one Role to another. A role can use various knowledge representations as appropriate for that Role's function, because the Artifact acts as a translating mediator. An Artifact can do these translations based on the properties of the communication path. Moving ontology-based data structures will require further extensions to the Coordination Artifact approach in order to support an ontology standard such as OWL.

12. Conclusions

The structure of Agent interaction in a Multi-Agent Society can be understood as a set of relationships among Roles played by each Agent. By organizing coordinated sets of Roles into first-class systemic objects, *i.e.*, into Coordination Artifacts, the design of both Agents and the communication between them, as represented by those Artifacts, can be simplified. The resulting Artifacts provide an ideal place for handling systemic issues, including the dynamic adaptation to changes in Quality of Service. Finally, Coordination Artifacts can readily be implemented using existing Agent infrastructure.

13. Acknowledgements

This paper builds on ideas presented at the KIMAS conference [25] and in a companion paper on scaling Coordination Artifacts[26]. The work described here was sponsored in part by DARPA UltraLog contract number #MDA972-01-C-0025.

14. References

[1] A. Omicini, A. Ricci, M. Viroli, G. Rimassa. "Integrating Objective & Subjective Coordination in Multi-Agent Systems." *AAMAS'04*.
 [2] A. Omicini. "Towards a Notion of Agent Coordination Context." *Process Coordination and Ubiquitous Computing*, Chapter 12. CRC Press, October 2002.
 [3] A. Omicini, A. Ricci, M. Viroli. "Coordination Artifacts: Environment-based Coordination for Intelligent Agents." *AAMAS'04*.
 [4] Cognitive Agent Architecture. <http://www.Cougaar.org/>
 [5] A. Omicini, S. Ossowski, A. Ricci. *Methodologies and Software Engineering for Agent Systems: The Agent-Oriented Software Engineering Handbook*, Chapter 14. Kluwer Academic Publishers, June 2004. "Coordination Infrastructures in the Engineering of MultiAgent Systems."

[6] J. Zinky, R. Shapiro, and S. Siracuse. "Complementary Methods for QoS Adaptation in Component-based Multi-Agent Systems." *The 2004 IEEE First Symposium on Multi-Agent Security and Survivability*.
 [7] M. Schumacher. "Objective Coordination in Multi-Agent System Engineering – Design and Implementation." Volume 2039 of *LNAI*. Springer-Verlag, Apr. 2001.
 [8] M. J. Wooldridge and N. R. Jennings. "Intelligent Agents: Theory and Practice." *Knowledge Engineering Review*, 10(2), pages 115–152 (1995).
 [9] "Cougaar Developer's Guide" v. 11.4. http://Cougaar.org/docman/view.php/17/133/CDG_11_4Final.pdf
 [10] Ultralog Home page: <http://ultralog.net>
 [11] SUN Java Beans Home Page: <http://java.sun.com/products/javabeans/index.jsp>.
 [12] A. Helsing, K. Kleinmann, and M. Brinn, "A Framework to Control Emergent Survivability of Multi Agent Systems", *AAMAS, 2004*.
 [13] J. Zinky, R. Shapiro, S. Siracuse, S. Ford, and D. Wells. "Case Studies of Node-level QoS Adaptation in Cougaar", *OpenCougaar Conference, 2004*.
 [14] A. Omicini, A. Ricci, G. Rimassa, and M. Viroli, "Integrating Objective & Subjective Coordination in FIPA: A Roadmap to TuCSoN." *AI*IA/TABOO Joint Workshop (WOA 2003)*, Italy, September, 2003.
 [15] TuCSoN page: <http://lia.deis.unibo.it/research/TuCSoN/>
 [16] A. Omicini and F. Zambonelli. "TuCSoN: a Coordination Model for Mobile Information Agents." *1st International Workshop on Innovative Internet Information Systems (IIS'98)*, Pisa, Italy, June 1998.
 [17] A. Omicini and E. Denti. "From tuple spaces to tuple centres." *Science of Computer Programming*, 41(3):277–294, Nov. 2001.
 [18] N. Carriero and D. Gelernter. "Linda in Context." *Communcation of the ACM*, April, 1989.
 [19] N. Davies, S. Wade, A. Friday, and G. Blair "Limbo: A tuple space based platform for adaptive mobile applications." *Proc. Int'l Conf. on Open Distributed Processing/ Distributed Platforms, 1997*.
 [20] T. Lehman, et al. "Hitting the distributed computing sweet spot with Tspaces." *Computer Networks (35) 2001*.
 [21] A. Murphy, G. Picco, and Gruia-Catalin Roman. "Lime: A Coordination Middleware Supporting Mobility of Hosts and Agents." *Submitted for publication*.
 [22] Quality Objects (QuO) home page: <http://quo.bbn.com>
 [23] G. Heineman, J. Loyall, and R. Schantz. "Component Technology and QoS Management." *International Symposium on Component-based Software Engineering (CBSE7)*, May 24-25, 2004.
 [24] J. Lee and T. W. Malone, "Partially Shared Views: A Scheme for Communicating among Groups that Use Different Type Hierarchies." *ACM Transactions on Information Systems*, January 1990, pp 1-26.
 [25] J. Zinky, S. Siracuse, and R. Shapiro, "Using QoS-Adaptive Coordination Artifacts to Increase Scalability of Communication in Distributed Multi-Agent Systems", *KIMAS*, March 2005.
 [26] S. Siracuse, J. Zinky, R. Shapiro, and T. Wright. "Scalable MAS-Based Control Systems Using QoS-Adaptive Coordination Artifacts." "2nd Workshop On Challenges In The Coordination Of Large Scale Multi-Agent Systems" *AAMAS Workshop*. July 2005.