## Cougaar Recent Release History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 September 2001</td>
<td>8.4</td>
</tr>
<tr>
<td>1 November 2001</td>
<td>8.6</td>
</tr>
<tr>
<td>11 January 2002</td>
<td>8.8</td>
</tr>
<tr>
<td>15 February 2002</td>
<td>9.0</td>
</tr>
<tr>
<td>1 May 2002</td>
<td>9.2</td>
</tr>
<tr>
<td>30 June 2002</td>
<td>9.4</td>
</tr>
<tr>
<td>16 December 2002</td>
<td>9.6</td>
</tr>
<tr>
<td>1 February 2003</td>
<td>10.0</td>
</tr>
<tr>
<td>19 May 2003</td>
<td>10.4</td>
</tr>
<tr>
<td>8 March 2004</td>
<td>11.0</td>
</tr>
<tr>
<td>5 July 2004</td>
<td>11.2</td>
</tr>
<tr>
<td>23 December 2004</td>
<td>11.4</td>
</tr>
</tbody>
</table>

## Recent Revisions

<table>
<thead>
<tr>
<th>Date</th>
<th>Section</th>
<th>Change</th>
</tr>
</thead>
<tbody>
<tr>
<td>23 December 2004</td>
<td>All Sections.</td>
<td>Minor edits.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Additional details for YP.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Detail regarding AssetData.</td>
</tr>
</tbody>
</table>
Contents

List of Figures ......................................................................................... iv

1 Overview ............................................................................................ 1
   1.1 Introduction to Cougaar ................................................................. 1
   1.2 Architecture “Quick Look” .............................................................. 2
      1.2.1 Full View: Cougaar Society ....................................................... 2
      1.2.2 Zoom In: Cougaar Community .................................................. 3
      1.2.3 Zoom in: Cougaar Node ............................................................ 4
      1.2.4 Zoom in: Cougaar Agent Internals ............................................. 5
      1.2.5 Zoom in: Cougaar Blackboard Contents .................................... 5
   1.3 Top Level Design Principles .......................................................... 7
      1.3.1 Top-Level Information Flow Concepts ......................................... 8
      1.3.2 Two-tier Communication Architecture ...................................... 9
   1.4 Document Overview ..................................................................... 10

2 Society and Community Concepts ....................................................... 11
   2.1 Community Interfaces and Relationships ....................................... 11
      2.1.1 Interfaces for Agents and Communities ..................................... 11
      2.1.2 Inter-Agent Relationships ....................................................... 13
      2.1.3 Tasks/Plugins for Managing Relationships .................................. 14
   2.2 Considerations for Constructing Societies and Communities ............ 15

3 Java VM and Cougaar Nodes ............................................................. 17
   3.1 VM Settings ................................................................................ 17
   3.2 Cougaar Nodes .......................................................................... 17
      3.2.1 Initialization ........................................................................ 18
   3.3 NodeAgent ................................................................................ 18

4 Cougaar Communications ............................................................... 20
   4.1 Agent Naming Services ................................................................ 20
      4.1.1 Name Generation .................................................................. 20
      4.1.2 White Pages ........................................................................ 20
      4.1.3 Yellow Pages ...................................................................... 22
      4.1.4 Service Discovery .................................................................. 23
      4.1.5 Local Discovery .................................................................... 23
      4.1.6 Peer-to-Peer Search ............................................................. 23
   4.2 MessageTransport ...................................................................... 24
   4.3 Inter-agent Communication .......................................................... 24
      4.3.1 Task Allocation ..................................................................... 24
      4.3.2 Relays ............................................................................... 24
      4.3.3 Attribute-based Addresses ..................................................... 25
   4.4 Blackboard ................................................................................ 25
   4.5 Cougaar Persistence Architecture ............................................... 25
      4.5.1 Persistence Goals .................................................................. 25
      4.5.2 Persistence Media Plugins .................................................... 26
      4.5.3 Persistence Modes ................................................................ 26

5 Cougaar Agents ................................................................................. 28
   5.1 Agent Internals .......................................................................... 29
7.2.9 Node Structure ........................................................................................................................................ 58
7.3 Example Cougaar Design Process ............................................................................................................. 58
  7.3.1 Methodology Example: Agent Enumeration ......................................................................................... 59
  7.3.2 Methodology Example: Role/Relationship Analysis ............................................................................. 60
  7.3.3 Methodology Example: Plugin Enumeration ......................................................................................... 60
  7.3.4 Methodology Example: Publish/Subscribe Analysis ........................................................................... 61
  7.3.5 Methodology Example: Task Grammar ............................................................................................... 63
  7.3.6 Methodology Example: Plan Element Map ......................................................................................... 63
  7.3.7 Methodology Example: Asset/Property Requirements Analysis ......................................................... 66
  7.3.8 Methodology Example: Execution Monitoring & Dynamic Replanning Analysis .......................... 67
  7.3.9 Methodology Example: Node Structure .............................................................................................. 68
List of Figures

Figure 1-1. Cougaar Society ................................................................. 3
Figure 1-2. Cougaar Community ..................................................... 4
Figure 1-3. Cougaar Node ................................................................. 4
Figure 1-4. Agent Internal Structure .............................................. 5
Figure 1-5. Agent Blackboard Contents ........................................ 7
Figure 2-1. A Simple Cougaar Society Consisting of Four Communities .............................................. 12
Figure 2-2. A Cougaar Community Consisting of Sub-communities ...................................................... 12
Figure 2-3. Various Relationships and Roles for a Single Agent ................................................................. 14
Figure 3-1. Classes at the Node Level .............................................. 18
Figure 5-1. Agent .............................................................................. 28
Figure 5-2. LDM/Blackboard ........................................................... 33
Figure 5-3. Typical Cougaar LDM Classes .................................... 39
Figure 5-4. Asset Prototypes and Property Prototypes ............... 40
Figure 5-5. Asset Instances .............................................................. 41
Figure 5-6. Specialized Asset Instance ............................................ 41
Figure 5-7. Cougaar UI Architecture .............................................. 44
Figure 6-1. A Hierarchy of (Bound) Components ......................... 46
Figure 6-2. Component Containment ............................................. 47
Figure 6-3. Component Binder .......................................................... 47
Figure 6-4. Complex Service Chain ............................................... 49
Figure 7-1 Cougaar Design Methodology Workflow .................. 54
Figure 7-2. Proposal Society ........................................................... 59
Figure 7-3. Deadline Scoring Function : “By June 10th” ............ 63
Figure 7-4. Plan Element Map for Management Agent ............. 64
Figure 7-5. Plan Element Map for Proposal Team Agent ........... 65
Figure 7-6. Plan Element Map for Production Team Agent ........ 66
1 Overview

1.1 Introduction to Cougaar

Cougaar (for Cognitive Agent Architecture) is an innovative software architecture that enables building distributed agent-based applications in a manner that is powerful, expressive, scalable and maintainable. In fact, Cougaar is a code baseline that has successfully demonstrated its utility at constructing dynamic, complex, distributed applications. Perhaps of even more significance than the software that implements its concepts, Cougaar represents a methodology, a tried and powerful approach towards designing and building distributed applications.

Cougaar was developed for DARPA (Defense Advanced Research Projects Agency) under the Advanced Logistics Program or ALP. The architecture was developed by ALPINE, a consortium currently composed entirely of BBN Technologies, over a period starting in 1996 until 2001.

Cougaar development and maintenance was then continued by BBN under a new DARPA program, Ultra*Log, over a period starting in 2001 until 2004.

The focus of the Advanced Logistics Project has been to develop techniques to better capture and solve the difficult problems of military logistics planning and execution. The scope and complexity of the problem of military logistics is tremendous:

- Millions of different object types to be managed
- Tens of thousands of different interleaved discrete business processes
- Thousands of different organizations with their own physical plants, constraints and user requirements
- Complex, continual interplay between planning and execution
- Over a thousand legacy databases and systems with different data models and protocols

The focus of Ultra*Log has been to make the Cougaar platform survivable, that is, to enhance the Cougaar platform with components offering:

- Robustness: A Cougaar application should survive the loss of any individual components and/or hardware substrate with minimal loss of functionality. This includes automatic recovery of lost agents, as well as various mechanisms to conserve resources and to use redundancies efficiently.
- Security: A Cougaar application should be capable of repelling various sorts of electronic attacks, should maintain information integrity, and should avoid exposing communications as much as possible.
- Scalability: The Cougaar infrastructure should not have any intrinsic scalability issues. It should be possible to implement Cougaar applications which scale to the degree that the application logic allows.

Standard software modeling techniques have proven inadequate to tackle a problem of this size and complexity. Standard software tools are unable to manage an ontology of so many distinct object types. The complexity of developing a standard top-down decomposition of the aggregate business processes involved defies monolithic modeling. Further, the interdependencies among different aspects of the model would tend to make such a model unmaintainable. Logistics systems have tended to treat planning and execution as entirely different realms, with different systems and architectures, due to the potentially chaotic feedback that execution can play on any plan.
As we will see, Cougaar was developed to tackle these challenges posed by the problems of military logistics head-on. But Cougaar is an approach to building software that transcends the domain of military logistics. Many complex domains can benefit from the approaches pioneered and embodied in Cougaar.

What kinds of problems are well suited to a Cougaar solution? While Cougaar was developed to handle a problem with all the complexities listed above, it is of potential value in any domain bearing any of the above complexities. For example, any of the following problem categories would benefit from being modeled in Cougaar:

- Problem domains that entail hierarchical decomposition and tracking of complex tasks
- Complex application domains involving integration of distributed separate applications and data sources
- Domains involving the generation and maintenance of dynamic plans in the face of execution
- Highly parallel applications with relatively loose-coupling and low-bandwidth communications between parallel streams
- Domains too complex to model monolithically, best modeled by emergent behavior of components

We should note that while Cougaar was designed to address the requirements of military logistics planning, we have seen applications of Cougaar to many different domains—some only tangentially related to military logistics, others completely separate. Nonetheless, this document will contain a flavor of military logistics in many of its examples and illustrations, as this is the domain to which Cougaar has been applied most broadly and successfully. It is important to keep in mind that the Cougaar technology is a domain independent architecture for large scale distributed agent systems.

1.2 Architecture “Quick Look”

In a nutshell, Cougaar is a large-scale workflow engine built on a component-based, distributed agent architecture. The agents communicate with one another by a built-in asynchronous message-passing protocol. Cougaar agents cooperate with one another to solve a particular problem, storing the shared solution in a distributed fashion across the agents. Cougaar agents are composed of related functional modules, which are expected to dynamically and continuously rework the solution as the problem parameters, constraints, or execution environment change.

The following section attempts to give a top-level description of many of the concepts of the Cougaar architecture that will be revisited in greater detail later in this document. We will take an approach to symbolically “zooming-in” from a high-level perspective of a Cougaar society, to illustrate and “drill-down” on particular concepts and how they interact.

1.2.1 Full View: Cougaar Society

A Cougaar Agent is an autonomous software entity that has been given behaviors to model a particular organization, business process or algorithm. A Cougaar Society is a collection of Agents that interact to collectively solve a particular problem or class of problems. The problems are typically associated with planning, where the plan objective and constraints may be continually changing and replanned in the face of execution. A Cougaar Community is a notional concept, referring to a group of Agents with some common functional purpose or organizational commonality. Thus a Cougaar society can be made of one or more logical communities, with some Agents associated with more than one community and other Agents not associated with any community. The society shares a DNS-like Namespace that allows all agents to resolve references to one another, and which may be monolithic or distributed/redundant. All Agents in a given society run the same Cougaar core software baseline, written in Java, though different Agents may contain additional “jars” to give them particular behaviors, model particular entities, or embody particular capabilities.
In Figure 1-1, we see a top-level view of an example Cougaar society. We see here a number of individual Agents and communities, containing different collections of Agents. *Enclave* is a term sometimes used to describe a Community defined by a common hardware or management base.

![Cougaar Society Diagram](image)

**Figure 1-1. Cougaar Society**

### 1.2.2 Zoom In: Cougaar Community

When we zoom in on a Cougaar community, things look similar to the view of a society: it is composed of other communities, perhaps sub-communities, and Agents. A community is not a software architecture concept, but a notional design concept that helps in designing and constructing the society out of a logical grouping of pieces. A community tends to speak a “common dialect,” meaning the Agents in a community may communicate about objects and issues that only this community knows about and using terms and activities only this community understands. There is typically a lot of traffic among Agents of a community, and fairly limited and constrained traffic between communities. A community typically has some notional interface of what it provides to the society: it performs the following services, it takes these inputs and produces these outputs, etc. Thus the concept of a community, during both design and implementation, provides an effective logical circumscription of a specific domain model and the suite of functionality that operates over that domain model.

Communities need not be distinct: communities are often hierarchical (e.g. as in departments in an org chart), or overlapping as a given Agent may be a member of any number of communities, each usually denoting a different semantic grouping.

In Figure 1-2, we see a Cougaar community containing a series of logically related Agents.
1.2.3 Zoom in: Cougaar Node

When we zoom in further, we reach a Cougaar node. A *Node* is a single Java Virtual Machine (JVM) instance that may contain and maintain multiple Agents. In most cases there is a 1:1 correspondence between the node and the hardware platform, but this is a configuration efficiency rather than a requirement. The grouping of Agents into a node is not necessarily domain related, but rather based on equitable sustainable sharing of computer resource requirements among all Agents. In some cases, Agents from different communities will share a single machine and node to be efficiently collocated near a shared data source, like a database. It is often convenient to think of a node as a special class of unnamed Cougaar communities where the logical grouping is by physical machine locality. As in other Cougaar communities, the membership can change over time as Agents are created, moved to other machines, and decommissioned.

All Agents on the same node share the same CPU, the same pool of memory and disk, and compete for incoming and outgoing bandwidth traffic. The node serves as a router of messages for the Agents it is hosting: messages to other Agents in the same node are passed directly within the same JVM, efficiently short-circuiting the message transport layer; messages to Agents in different nodes pass through a MessageTransport layer that passes the message through the network to the receiving node which then routes it to the appropriate Agent within that node.

In Figure 1-3, we see a Cougaar node containing several Agents, with message traffic passing among internal Agents as well as inter-node message traffic for external Agents.
1.2.4  **Zoom in: Cougaar Agent Internals**

As we zoom in yet further, we view the Agent and its internal components. An Agent consists of two major components: a partitioned distributed **Blackboard**, and **Plugins**. The Plugins are software components that provide behaviors and business logic to the Agent’s operations, and operate by publishing content and subscribing to objects on the Blackboard. The Plugins are encouraged to fit a computational model, i.e., usage pattern templates and their corresponding methodology, described by the Cognitive Agent Architecture (Cougaar) described in Section 7 of this document.

The Blackboard contents are segmented into sets of logically-related objects by **Domains**. A Domain is, in effect, a specification of the language used by plugins to communicate with each other and with related plugins in other Agents. Each Domain has a set of **LogicProviders**, which are plugin-like components that act as translators into other Domains’ business logic and/or Agent messaging.

The Agent is responsible for management of queues containing messages to and from other Agents, scheduling the execution of Plugins and managing the subscription mechanisms of Agents. The details of these operations are covered in the Cougaar Developers’ Guide.

In Figure 1-4, we see a Cougaar Agent with its internal structure exposed, showing the Blackboard and Plugins.

![Figure 1-4. Agent Internal Structure](image)

1.2.5  **Zoom in: Cougaar Blackboard Contents**

As we zoom in to our finest level of detail, we see the detailed contents of the blackboard. Plugins publish and subscribe objects to the blackboard. In principle, these could be any objects, as defined by the application’s business logic. For example, the Cougaar planning Domain implements the following sorts of objects:

- **Tasks** represent a requirement or request from one agent to another to perform or plan a particular operation.
- **Assets** represent resources to which tasks are allocated. A distinguished type of asset is an **OrganizationalAsset** or an **EntityAsset**, which represents a proxy to another Agent.
• PlanElements contain dispositions of tasks. Tasks may be allocated (through an Allocation PlanElement) to an asset resource, or expanded (through an Expansion PlanElement) into subtasks, or aggregated with other tasks (through an Aggregation PlanElement).

Other Blackboard interaction models are used, both within the Planning domain as well as in others. For instance, Cougaar has defined an interaction pattern (and reference implementation) called a Relay, which represents a range of abstractions of a communication channel between one or more Agents.

The blackboard of a Cougaar Agent is part of the distributed blackboard managed in a distributed fashion by the whole Cougaar society. Each Agent owns its blackboard and its contents are visible only to that Agent. All sharing of blackboard state is done by explicit push-and-pull of data through inter-Agent tasking and querying. In this way, Cougaar is able to maintain fine-grained state in individual Agents while sharing only high-level synopsis information around the society, making the management of information scalable and efficient.

A fundamental concept of the management of Assets within an Agent is “time-phased locality of reference.” By this we mean that a representation of a given object may be present in multiple Agents, but is only actively managed at one place at a time. In this way, different pieces of the plan of a given object can efficiently be performed in parallel without fear of conflict. An analogy of this model is that while a specific truck may be driven by different people on different days, it only has one driver at a time – this means that only the driver in control needs to worry about driving the truck, even if some other driver is, at the same time, planning out the next day’s deliveries.

Another fundamental concept of the distributed management of the blackboard is that of “managed inconsistency.” There is no overarching control within Cougaar that synchronizes operations, or imposes inter-Agent transaction boundaries. Each Agent works independently and asynchronously on messages passed from one another, and responds independently and asynchronously on responses received from Agents. No inter-Agent prioritization or ordering is imposed. All told, these features allow the Agents to work efficiently and independently: there are no delays waiting for another Agent to get its job done, or recover from a network problem, or a burst of requirements from yet another Agent. What this independence imposes, though, is that the state of the society-distributed blackboard, taken in the aggregate, is never guaranteed to be in a consistent state. The blackboard of any given Agent will be consistent, as interactions with the blackboard from Plugins is transaction-bound. But one cannot assume that the picture one sees based on multiple queries to multiple Agents will produce a consistent picture: there may be messages still in the incoming or outgoing queues that have yet to be processed, or Plugins not yet done completing their changes to the blackboard. Cougaar makes this inconsistency explicit and notes that any multi-Agent analysis needs to be sensitive to it and “sew up ragged edges” on its own.

In Figure 1-5, we see the contents of the blackboard of a given Agent to which Plugins publish and subscribe. The blackboard contains a dynamic chain of plan elements, tasks and assets.
1.3 Top Level Design Principles

There are several fundamental principles that underlie the Cougaar architecture that can be seen at many levels across the design. These principles serve as themes that will be revisited throughout this document.

Composability. One pattern that recurs throughout the Cougaar architecture is taking complex problems and decomposing them into smaller, maintainable components. These components are, in turn, composed into a larger entity whose behavior is the emergent behavior of its components. We see this pattern in many examples:

- **Agents** are composed from many **Plugin** components, each providing a small piece of business logic or functionality, allowing the Agent’s behavior to emerge from the composed pieces.
- **Assets** are composed from **PropertyGroups**, which contain a cohesive set of data slots, often from a single data source. A given Agent can determine what PropertyGroups are relevant to a given asset in a given Agent or modeled organization. Assets are further composed of **Prototypes**, which capture standard patterns of PropertyGroups that are invariant over a large set of instances.
- **Communities** are composed of smaller Communities and **Agents**. Societies are, in turn, composed of Communities. A community may present a simple interface to other Agents and Communities, hiding its internal structure and complexity.
- A **Blackboard** contains logically separate sets of objects grouped into Domains by their common application language. Each Domain specifies a series of **Logic Providers**, which allow the Domain’s logic to be translated into other co-resident languages, including Cougaar Messaging.

Information Hiding and Encapsulation. One theme of the Cougaar architecture is that a given component should have access to all the data it needs, and no more. All components are encouraged to provide a clean interface that allows composability, while intentional information hiding enhances scalability of a Cougaar society. Examples of these patterns in Cougaar include:

- Agents cannot see into the blackboards of other Agents, and must utilize standard interfaces to obtain the set tasks another Agent can perform and generate.
• Plugins cannot see into the state of other Plugins, and have clean “publish/subscribe” contracted behaviors.
• Communities typically have clean interfaces with one another, hiding their internal Agent configuration from other communities.

**Time-Phasing.** All aspects of the Cougaar plan are expected to vary over time, both as the plan changes, and as the plan is executed. Expected quantities, costs, values of different entities will change over the course of a planned operation. To this end, all information about physical entities within Cougaar, typically captured in Assets, are time-phased, meaning the time-wise history of values associated with that quantity for that object are maintained. One needs to be able to ask, “How much will we have on this or that date?”: the non-time-phased question, “How much is there?” simply doesn’t make sense in a time-phased Cougaar planning context.

**Dynamic Replanning and Execution Monitoring.** A key operating mode of Cougaar is planning, and storing that plan in a distributed fashion throughout all the Agent blackboards. The plan is built on a continual dynamic negotiation between Agents and Plugins to generate a feasible and ultimately optimized cooperative solution. The plan is based on real world requirements, situation information and asset availability: what do I need to do, what is the state of the environment and what can I use to accomplish my task? As these variables change, the solution becomes stale and Cougaar forces replanning to determine how to adjust the plan to compensate for those changes, if possible. Further, Cougaar continually monitors the plan as it is executed, and forces replanning as assumptions are modified in real-time.

**Security.** The Cougaar architecture is designed to contain a significant degree of commercial-grade security, ensuring that all inter-Agent communications are assured to be snoop-proof and tamper-proof. Further, the infrastructure core software, the Plugin modules and configuration information are all designed to be certifiably intact and secure. No Cougaar installation should be vulnerable to traffic interception, rogue agents or corrupted configuration baseline.

**Robustness.** Cougaar applications are designed to be long-lived, i.e., continuously running 24x7x365. To that end, many aspects of the Cougaar architecture are dedicated to allowing a society of agents to survive the temporary outage of a single Agent. Agents persist their internal state, which can be restored, as the Agent is restarted. Other Agents in the society are able to tolerate a long-term absence of a given Agent (due to Agent failure or network outage) from the society, reconnecting appropriately as it rejoins the society. In addition, Agent, community and society configurations can change without impacting other components of the society – providing both dynamic reconfiguration and a long term evolution of functionality.

**Scalability.** The Cougaar architecture is designed from the bottom up to support applications to a massive scale. By encouraging encapsulation, data hiding, and fine grained information management we limit the information passed between Agents to a bare minimum. By establishing a peer-to-peer inter-Agent communications, we avoid exponential growth in the interdependencies and interactions among different agents. The Plugin architecture allows for building large software systems with much more manageable maintenance and integration costs than traditional architectures. Our paradigm of PropertyGroups composing assets and Prototypes modeling instances allows for modeling extremely large object models. The intended scale of Cougaar architectures is very large: many of the features of Cougaar are best showcased on very large domains and applications.

### 1.3.1 Top-Level Information Flow Concepts

This section is intended to provide some sense of the “big picture” of how Cougaar does its work. In other sections, we will be stressing how Cougaar is a distributed architecture, but here we will illustrate how its operations allow for complex information flows and negotiations regardless of the topology of the society.

A Cougaar society is started at some point by hand or by an automated Application Server process. From there, Nodes can be manually killed and restarted, while new Nodes can join the society dynamically. The
Agents within the Nodes set up initial relationships with one another, so they can start to send tasks to one another. Agents invoke LDM Plugins to interact with databases, legacy systems and sensors to fill their Agent with Assets representing physical entities for whose management the Agent is responsible.

While Cougaar can be started, run and killed, its intended operating mode is to stay up permanently, continually handling an incoming stream of requirements, continually trying to find a solution to these requirements (and other pre-existing requirements) against available resources. Cougaar expects to be continually processing, continually trying to find a better solution to the given problem, continually reacting to changes in resources, in requirements, events monitored from execution.

How does the processing of Cougaar commence? There may be many stimuli. There may be LDM Plugins in some Agents who read real-world data from databases or sensors that trigger Cougaar processing. There may be a time-based Plugin who injects particular requirements on a time basis. Typically, a user interface will inject a set of requirements into Cougaar to trigger processing.

With the introduction of new requirements, Cougaar begins the work of dynamic planning and execution monitoring. Tasks are decomposed (by Expansion) and assigned (by Allocation) to other processing units, either in the same Agent or another Agent. Downstream processing results are passed back up to higher-level processing who then can aggregate or summarize this information back up the chain, or react to news by replanning.

Each task, then, creates a “channel” for information flowing through the society for requirements passing down, and responses flowing back up. At each point, the execution of the planned requirements is monitored, and replanning may occur if significant discrepancy is detected between the planned operations and the observed operations.

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 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. Customers and providers haggle between what is desirable on the part of the customer (in terms of “Preferences” on allocated tasks) and the Provider matches these preferences against what is available among its controlled assets (returning an AllocationResponse, indicating how well the task was satisfied relative to the preferences). 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.

1.3.2 Two-tier Communication Architecture

A Cougaar application has two equally-important levels of communication present and active at any given time:

- Agent-Agent, where agents communicate with each other as peers, hiding the internal business logic and allowing loosely-coupled, asynchronous, and widely distributed problem solving, perhaps with Agents collocated with their supporting external systems (data sources, humans, etc). Agents typically differ based on where they are and who they “represent”, so inter-agent communication is very much like messages between humans or autonomous systems.

- Plugin-Plugin, where components communicate with each other through the Agent’s Blackboard using tightly-coupled, transactionally-protected interactions. Plugins are often vastly different from each other, based on the task each one performs. Plugins more often represent detailed business logic and portions of decomposed problems than recognizable external entities, though they still cooperate more than compose to solve problems.
1.4 Document Overview

This document is organized as follows:

**Section 1, Overview**, presents the motivation for the Cougaar architecture, introduces the fundamental design concepts, and lays out the structure for the rest of the architecture document.

**Section 2, Society/Community Concepts**, describes how communities and societies are composed from Cougaar Agents.

**Section 3, Java VM and Cougaar Nodes**, describes some aspects of Cougaar which are inherently VM-related, and the mostly one-to-one relationship between VM and a Cougaar node.

**Section 4, Cougaar Communications**, describes the various communication mechanisms in Cougaar: Agent Naming Services; Message Transport; Inter-agent Communication via Relays; the Blackboard for Intra-agent Communications; and the Cougaar Persistence Architecture with respect to external storage media.

**Section 5, Cougaar Agents**, describes the structure of a Cougaar agent, its various internals, and several contained components like Blackboard, Plugins, and Servlets. This section also introduces the Logical Data Model (LDM) that is both a term for the Cougaar data representation language and the Cougaar infrastructure object used to implement objects in the LDM language.

**Section 6, Component Model**, describes how abstract elements of the Cougaar infrastructure like Components and Services relate to and interact with each other.

**Section 7, Cougaar Design Patterns**, addresses the Cougaar design methodology and the Cognitive Agent Architecture approach to the design of distributed composable agents.
2 Society and Community Concepts

This section of the Cougaar Architecture Document discusses Cougaar from the perspective of societies and communities. A Cougaar society consists of a number of communities, each containing Agents and, perhaps, sub-communities. A Cougaar society or community often reflects the structure and behavior of a real society or community in terms of its emergent behavior, its external interfaces, its internal structure, though this is a byproduct of the Cougaar methodology and not a strict requirement.

The construction of a society and communities within Cougaar has some standard approaches and patterns to it, much as the decomposition of an organization into Plugins has some standard approaches and patterns that we refer to as the “Cougaar Design Patterns” in Section 7 of this document. This section describes concepts associated with societies and communities, as well as a checklist of issues to consider in constructing a Cougaar society out of components.

2.1 Community Interfaces and Relationships

By the time of constructing a whole community or society, much work has already gone into the individual pieces: separate Plugins have been developed and tested, and these have been composed into Agents or parts of Agents that have tested.

The society that we wish to build consists of all the organizations we wish to model, all interacting in a coherent manner. When the number of organizations is large, the task can be daunting: we may have dozens or hundreds of different Agents in our society: how do we make some order out of this?

The key concepts for constructing and managing a society are external interfaces and inter-Agent relationships.

2.1.1 Interfaces for Agents and Communities

Each Agent presents an interface to external players: what kinds of messages do I receive, what kinds of messages do I send? As we will see below, one approach to defining a community is defining groups of Agents that present, together, a common interface to all other societies. In this way, one Agent may hide a rich hierarchy of Agents performing a complex interaction behind a simple interface.

Much as we construct an Agent from Plugins in such a way as to ensure completeness and consistency (all inputs are provided, all outputs are accepted), a society should provide this same sense of completeness and consistency of interfaces.

The following are examples of communities and their interfaces, drawn from the domain of military logistics planning.

Transportation is a key component of military logistics. There are complex organizations that take transportation requirements and plan how to satisfy those requirements by coordinated use of ground, sea, and air assets. To the rest of the logistics world, there is just “the transportation community,” who take transportation tasks and determines feasibility. Within the “transportation community,” there are over a hundred distinct organizations that perform the planning of individual transportation requirements against a wide pool of transportation assets. In Cougaar, the set of Agents that perform transportation requirements form a natural community. There is a set of Agents representing the external interface: “Give me ‘TRANSPORT’ tasks, and I’ll give you a feasibility answer.” There are many Agents hidden behind these gateway Agents who perform the implementation (the detailed planning) of this interface. Figure 2-1 reflects the internal and external view of a notional transportation community within a notional Cougaar society.
Even within the Transportation community, there is significant sub-structure that can be captured as sub-communities. From a high-level, one can see transportation planning as “Do the Ground leg, then the Sea or Air leg, then the other Ground Leg.” Each of these components, Ground, Sea and Air logistics planning have clean interfaces and complex sets of Agents underneath them. Figure 2-2 reflects the decomposition of the Transportation community into subcommunities.
In military logistics, there are many “customers” who require services for logistic planning: each Corps, Division, and lower unit requires logistics services. These organizations, each with their own structure are parallel, independent structures to one another and are undifferentiated to their providers. The structure of a given Corps or Division may be presented as a single interface (of required services) to the Transportation or Supply communities, for example, while the implementation of the structures of these organizations is contained in the hierarchical relationships among the constituent Agents.

2.1.2 Inter-Agent Relationships

If one were to take a set of Agents, compose them into nodes, point all these nodes at the same name server and start them up, surprisingly little would happen. Agents do not know of the existence of arbitrary other Agents in their society, nor do they know how to communicate or take advantage of them. For proper interactions within a society to be established, relationships among Agents must be established.

Relationships between Agents represent a role or capability that a given Agent can perform for another Agent. They do, however, tend to fall into several patterns and have distinct characteristics:

- Relationships are between two Agents: Cougaar does not explicitly support relationships among larger sets of Agents (I may have N siblings, but each sibling represents a distinct relationship).
- Relationships need not be unique: I may have many different Agents that provide the same service to me.
- Further, an Agent may provide multiple roles for the same Agent.
- Relationships are time-phased, in that Agent X can provide a particular service or role to Agent Y over a particular period of time.
- Relationships are dynamic. Typically a Cougaar society starts with an initial set of relationships, but these can be modified, enhanced, deleted, extended in the course of the processing of the society.

Roles and relationships tend to fall into particular patterns, of the following sorts:

- Superior/subordinate relationships, as described below.
- Customer/provider relationships, as described below.
- Note that most standard relationships are mutual. If A is related to B, then B is related to A (e.g., A is the superior of B, and B is the subordinate to A).
- Each relationship contains a role that describes the nature of the relationship. The relationship also contains two (“organizational”) assets—the asset performing the role and the asset for which the role is performed. The relationship that describes A as the superior of B is the same as the relationship that describes B as the subordinate of A (e.g., the role is superior, A is the asset acting as superior, B is the asset that has A as a superior).

The following figure represents the view of an Agent within the context of its relationships. Note that a given Agent can be in many different roles and relationships simultaneously.
Figure 2-3. Various Relationships and Roles for a Single Agent

Superior/subordinate relationships exist among Agents with long-standing orders. Typically, a superior gives high-level tasking to a superior of the form “Do Your Job according to these parameters, and report to me according to this format.” The subordinate will then do its job, be it, e.g., maintaining a set of assets, providing services (as a provider) to some other set of Agents, or managing some external interactions. It will report aggregate and trend information (“How am I doing on the task you gave me”) to its superior on a periodic basis. Typical Verbs of tasks flowing from superior to subordinate would be “ReportForDuty” or “ReportForService” as described below.

Customer/provider relationships exist among Agents who provide day-to-day, task-order services for one another. Unlike the superior/subordinate relationships where a single (or small set of tasks) pass between them, large streams of discrete tasks flow from customer to provider who returns responses and updates on each one. Typical Verbs of tasks flowing from customer to provider would be “Supply” or “Transport.”

2.1.3 Tasks/Plugins for Managing Relationships

Once an Agent has other Agents in its blackboard (represented as Organization or Entity Assets) related to it by roles, it may task that Agent to perform tasks in accordance to that role and time-phasing. But how do these Agents appear in the blackboard?

There are two ways by which Cougaar Agents develop relationships with each other. Relationships may be set-up by assignment or by service discovery. Currently, assignment can be used to set-up both superior/subordinate relationships and customer/provider relationships where the two Agents are specified in society initialization information. Service discovery can be used to set-up customer/provider relationships where the customer chooses the provider from a dynamic list. These two methods are compatible, so a Cougaar society can contain a mixture of relationships established through both mechanisms.

In the assignment process, there are two sets of two standard Cougaar Plugins that manage the static and dynamic flow of Agent roles and relationships to other Agents. The base version is in Planning, and deals
with generic “Entity” Assets. In GLM, the Plugins deal with Organizations. The latter is slightly more specific, and just an extension of the version in Planning. Here we refer primarily to the GLM version.

- The OrgDataPlugin or AssetDataPlugin (in GLM and Planning respectively) set up an initial set of inter-Agent relationships by parsing a set of static data. Where they get this data, depends on the version of the Plugin that you use. Whichever form the data may be in, it includes the list of roles and relationships that this Entity/Organization provides to other Agents. The Plugin creates the appropriate ReportForDuty and ReportForService tasks for the Agent’s Organization Asset.

- The OrgReportPlugin or AssetReportPlugin handles two different tasks to facilitate dynamic establishment of new relationships.
  - ReportForDuty is the verb of a task indicating that an Agent should establish a relationship to a new superior. One can have multiple superior relationships, both time-overlapping and non-time-overlapping. When receiving a ReportForDuty task, the OrgReportPlugin (AssetReportPlugin) will cause a new superior relationship to be added to the relationship schedules of both the local Agent’s Organization (Entity) Asset and the superior Agent’s Organization Asset. In addition, a copy of the Agent’s Organization Asset will appear in the blackboard of the new superior Agent.
  - ReportForService is the verb of a task indicating that an Agent should establish a provider relationship to a new customer. One can have multiple provider relationships, both time overlapping and non-time-overlapping. When receiving a ReportForService task, the OrgReportPlugin (AssetReportPlugin) will cause a new provider relationship to be added to the relationship schedules of both the local Agent’s Organization Asset and the customer Agent’s Organization Asset. In addition, a copy of the Agent’s Organization Asset will appear in the blackboard of the new customer Agent.

In the service discovery process, two Cougaar Plugins establish a ServiceContract. If the two Agents express compatible service needs and capabilities, LP processing creates the relationships between the Agents and updates each Agent’s Organization or Entity Assets.

- The SDClientPlugin submits queries for providers fulfilling the desired role and any other criteria. Based on the query results, it selects a provider (or providers) for negotiation. The Plugin contacts the selected provider(s) with a ServiceContract expressing the time period and service requested.

- The SDProviderPlugin responds with information about the time periods and services it is able to provide.

### 2.2 Considerations for Constructing Societies and Communities

The following section gives some suggestions and guidelines for constructing societies and communities from a set of Agents. It is important to note that “community” is not a Cougaar software concept: it is a society construction concept that makes a large society easier to conceptualize and manage.

In many ways, the problem of taking a large society and identifying communities parallels any Object Oriented decomposition effort: we seek to identify substructure that allows us to abstract external interfaces and hide complex internal structure.

The most significant characteristic of a community is that it presents a single, coherent interface to the rest of the society for a given capability. Ideally, we can identify such interfaces that encapsulate and hide the internal structure of how a given function is accomplished. Recursively, as that implementation breaks down into modules, it makes sense to think of these sub-components as sub-communities, each with its own interface.
When we build an Agent, one should be careful not to overload parallel independent threads into a single Agent unnecessarily: one can parallelize these threads into separate Agents, enhancing mobility and making resource consumption much more granular. The same argument applies to the construction of communities. It may make sense to break out parallel pieces of what is nominally the same organizational structure so that each works independently.

Again, mirroring a good design principle from Agent/Plugin development, the society/community designer is encouraged to seek out generic patterns. Where there are similar interactions between Agents or communities, new generic Plugins may be developed, or generic/parameterized Roles/Relationships may allow different parts of the society to be modeled in the same way.

In designing a community that provides an external interface to other communities, it is crucial to avoid the potential of bottlenecks. Perhaps a single Agent is the gateway to many Agents that do the work across a wide set of hardware. We can make multiple Agents as entry-points into this society providing the same interface, and have different gateway Agents report to different parts of the rest of the society.

The issue of bottlenecks introduces the critical general area of scalability. If a given community is required to satisfy all the needs of a particular sort for the entire society, it must be able to scale to meet the projected demand. Perhaps that requires the possibility of parallel whole community instances, or perhaps it requires an implementation behind the interface that allows for “throwing hardware” at the community to allow it to support greater loads. In any event, the design of Agents and communities must take the ultimate required loads into consideration and make sure that both the interface not be a bottleneck and the implementation society be able to handle the load.

The efficient interoperation of Agents and communities within a society depends on a reasonable and maintainable set of roles and relations. The set of roles should be as rich as the set of services actually provided, so that there is no ambiguity as to what Agent is providing what service. On the other hand, the proliferation of roles can be confusing and hard to maintain, so efforts should be made to use the same role to mean the same thing, and to encapsulate sets of roles that are internal to a given community from the rest of the society.

As we have seen, there is value in defining communities by identifying structures and patterns of function and relationships. Where possible, use tools or templates to allow for maintaining the same kind of community as an instance of a particular template, by parameterizing configuration files, or by creating the files automatically from a database. In this way, one can be assured of consistent behavior within a community, a consistent environment in which the Plugins can operate, and a consistent interface provided by the communities.

When we construct a node in Cougaar, we are usually considering hardware resources: we want to level-load CPU and memory requirements, and minimize bandwidth requirements over a low-bandwidth channel (so tightly coupled Agents are put in the same node or host). The same kinds of considerations are often important in designing communities. Communities tend to be Agents that communication primarily with one another (and much less so to external customers or providers). A community may be too big to all fit in a single node or host. However, there are advantages to placing a community on a LAN, so that the high-bandwidth communications between Agents of a community can take place efficiently. Further, it is often the case that Agents of a community will need to have shared access to some external resource (e.g., a database). In this case as well, it makes sense to try to design the physical layout of the society so that the community is co-located in the same LAN to allow efficient access to this shared resource.
3 Java VM and Cougaar Nodes

This and the following sections introduce the Cougaar core infrastructure components in order of highest to lowest level of abstraction. The more abstract components (VM and node) do not participate directly in Cougaar application problem solving process, merely serving as a lifeline for the rest of the system. The more specific components are much more tightly coupled with each other, both intra- and inter-layer, while maintaining as much encapsulation and isolation as possible to allow for maximum parallelism and maintainability.

3.1 VM Settings

The Java VM is usually not considered part of the Cougaar levels of abstraction. However, it is worthwhile noting that there are some aspects of the system which are inherently VM-related, even when there is a one-to-one relationship between VM and node.

A number of Cougaar components are controlled by Java system Properties (-D arguments). Typically these properties are use to specify alternate values for normally defaulted parameters, e.g., tuning the behavior of a specific Agent by adjusting thread use, timeouts, etc. Javadoc-generated documentation files describe most of the common options.

Non-constant static facilities are extremely rare in Cougaar infrastructure. Since the infrastructure allows Agents to coexist within a single Node, code cannot in general safely assume that there will only be one instance of a plugin (for instance) resident in a given VM. Use of non-constant statics is strongly discouraged as it breaks encapsulation and is rarely required.

3.2 Cougaar Nodes

A Node is effectively a container of Agents. It is the highest-level Cougaar object that participates directly in a Cougaar society. Nodes are used from outside the society to initialize and manage the node-local Agents. Nodes provide a number of Node/VM-level facilities to the contained Agents. The centralization of these facilities is due to convenience and efficiency, not to any particular design considerations. The Node instance itself does not generally participate in the application problem solving process—it is only a control and management point for a set of Agents.

Nodes are actually implemented as distinguished Agents, so may be addressed as message targets, may load additional management logic via plugins and may be probes by UIs. While most application developers need never pay attention to this, many of the robustness and security aspects of high-survivability Cougaar applications are implemented as NodeAgent components and plugins.
3.2.1 Initialization

A node is responsible for initialization of its Agents. This may be accomplished in either of two ways:

- An XML file is often used to describe both the set of Agents to be present in a given node and the set of Plugins to be loaded into each of the listed Agents. When this method is used, the Node instantiates the requested set of Agent instances, and then each Agent reads its section of the XML file to find the Plugins it should load.

- A Node may be run as a generic Agent server. That is, an empty node can be created, with either a Plugin or Servlet that will respond to external requests to create, modify, or remove agents on the local node. This approach is more dynamic than the static XML-based approach, and can be used to support either Agent-based or GUI-based management tools.

Developers often use the XML configuration approach, since it is often easier to statically configure the Agent society.

3.3 NodeAgent

As described above, Node is an abstraction of the services and structure required to support one or more Agents in a single memory space or VM. The Node abstraction is comprised both of the high level structures of root-level component containers and a set of "Node Level Services". These services are published by high-level components so that they available to be shared between all requestors in the Node, including all Agents.

Prior to Cougaar 9.2, the Node object had a rather complex and ad-hoc internal structure, with a set of hard-coded components implementing the services, and a separate set of objects implementing the root component and service models. Starting with Cougaar 9.2, most of the services and components are implemented as plugins and other subcomponents of a new object, NodeAgent.

A NodeAgent is a real Cougaar Agent that has the task of providing Node-level lifeline and management services to its Node. While it does not itself "contain" the root objects of the component hierarchy, it does have full control over those objects. Node level Services are implemented as child components (often Plugins!) of NodeAgent rather than actually implemented at the top level. NodeAgent subcomponents may
request access to a NodeControlService instance which may allow the component to control and to offer services to the entire node rather than just peers of the component and lower (as is typical of other components) - See CDG.

There is always exactly one NodeAgent per Node, and neither Nodes nor their NodeAgents may be moved under Cougaar control (cf. AgentMobility in the CDG). However, since NodeAgents are true Cougaar Agents, they do have a Blackboard which may be persisted, so may retain state across host failures.

Nodes are extensible by adding Components (e.g. Plugins, servlets) to the associated NodeAgent. All methods of configuring Agents are available for configuration of NodeAgents by using the Node's name as the agent configuration name. Components may be added to Node’s XML configuration file, just like any other Agent.
4 Cougaar Communications

4.1 Agent Naming Services

Cougaar includes support for distributed agent naming services. These naming services are used by the Cougaar message transport to route message over multiple network protocols to mobile agents. Application developers can also use the naming services to dynamically discover agents at runtime.

In this section we discuss the five different types of distributed naming capabilities have been identified:

- “Name Generation” constructs a globally unique agent name.
- The “White Pages” is a table that maps names to network addresses (e.g. DNS).
- The “Yellow Pages” is an attribute-based directory (e.g. a categorized phone book).
- “Local Discovery” uses LAN-based IP multicast to locate nearby agents.
- “Peer-to-Peer Search” allows an agent to search adjacent agents for resources.

4.1.1 Name Generation

All agents in a Cougaar society are required to have a unique name. An agent’s name is used to route messages to the agent and may hold special meaning in the developer’s application.

Most Cougaar developers select their own agent names to match the role of the agents. For example, an agent that models the Boston Department of Motor Vehicles may be named “dmv.boston.ma” by its developer. Note that the Cougaar infrastructure doesn’t assume that this name has any particular meaning, just like an Operating System doesn’t care what you name your files so long as they are unique.

A runtime name generator can be used to randomly select a name when the agent is created. This is typically accomplished by hashing the local host’s IP address with a large random number. A random name may make sense for an anonymous embedded sensor.

A simple random name generator is included in Sun’s RMI support (java.rmi.dgc.VMID).

4.1.2 White Pages

The white pages is a distributed table which maps agent names to network addresses. The primary function of the white pages is to support the Cougaar message transport and other network-aware components.

For example, a white pages lookup of “AgentX” may return a set of network entries such as the agent’s RMI message address (rmi://test.com:1234/xyz) and the agent’s servlet port (http://test.com:8800). This is similar to DNS name-to-address resolution.

The predecessor of the white pages is the Cougaar Naming Service, which has existed in different forms over the lifetime of the Cougaar project. The current white pages implementation is the third full redesign. Many important lessons were learned over the years, so it’s worthwhile to trace the history of the naming service:

The initial naming service implementation was developed in 1996. The naming server ran on a single JVM and communicated over RMI. The implementation was deeply tangled with the message transport, which
was its only client. There was no caching, change notification, security, or persistence. If the JVM hosting
the naming service was killed, the rest of the agents could no longer perform their necessary (name →
address) lookups. Despite these limitations, the initial naming service worked fine in societies with less
than 30 or so agents.

The second naming service implementation, developed in 2000, was primarily focused on separating the
naming service from the message transport and supporting new naming service clients. The front end was
changed to use JNDI, and the back end supported either RMI or LDAP. The RMI implementation added
change notification callbacks to help reduce client-side polling. JNDI allowed clients to bind arbitrary
objects and perform full JNDI attribute-based queries. Several client-side ad-hoc caching schemes were
implemented but never generalized to benefit all naming service clients. The naming service was still a
single point of failure, even with LDAP replication, and there was no JNDI federation or persistence
support in the RMI back end. This naming service barely supported societies of 200 agents.

Several scalability lessons were learned from the second naming service implementation. All of these
lessons are self-evident if you envision an agent society the size of the entire Internet, with millions of
individual agents. The lessons we learned include:

- It’s very tempting for developers to treat the naming service as a global database, which results in
  severe scalability issues.
- Simple requests like “list all agents” are a bad idea when there are hundreds of entries.
- A single-point name server is not robust or scalable. Replication alone will not fix this problem,
due to the quantity of data and cache synchronization overhead.
- Caching and leasing should be built into the design, since simple client-side polling can easily
  overwhelm the naming server.
- RMI and JNDI are blocking APIs that suffer from socket resource limitations and poor I/O
  handling. An asynchronous design that supported better message delivery control would have
  been beneficial.
- The real issue is “Where does the data live?” A highly distributed agent system must use a highly
distributed naming service.

The latest naming service implementation included several high-level goals:

- Must be **scalable**, to support thousands of agents, running on hundreds of hosts on a wide area
  network.
- Must be **robust**, with no single point of failure, multiple servers to survive overloads, and
  persistence to support restarts
- Must be **efficient**, utilizing an integrated caching and garbage collection scheme
- Must be cleanly **integrated** into Cougaar, leveraging the Cougaar message transport for message
  protocols, quality of service, and security

Reflection on how the naming service was used prompted us to split the naming service into two separate
services. A phone book analogy was adopted, where the two services are: A “white pages,” which maps
names to network addresses; and a “yellow pages,” which supports more complex attribute-based searches.

The white pages service has been modeled after DNS. Agent names now support Internet host name
semantics with the ‘.’ separator character. A hierarchical name space will support better cache control and
help distribute the data to multiple naming server agents.
In Cougaar 10.0 the initial WhitePagesService API was defined. An entry in the white pages contains an agent’s network address (URI) and an optional certificate to validate the entry. The white pages support an asynchronous callback API with methods to:

- `resolve(name)` → set of entries for that name
- `list(suffix)` → set of names
- `bind(entry)`
- `rebind(entry)`
- `unbind(entry)`

The Cougaar 10.0 implementation was built on top of the prior JNDI-based naming service.

Cougaar 10.2 will feature a white pages implementation that replaces the JDNI naming service. The white pages will use the message transport to send and receive messages, which will allow it to use alternate message transport protocols and quality of service guarantees. The white pages will be bootstrapped with pre-resolved message transport addresses for the “root” name servers. The Cougaar 10.2 implementation will support caching and a resolver hierarchy.

Cougaar 10.4 and later will support peer-based white pages replication and state reconciliation for restarted name servers. The white pages will effectively become an agent-based application that runs within Cougaar, whose job is to support the message transport and other Cougaar network-aware components.

### 4.1.3 Yellow Pages

The yellow pages is a directory service that supports attribute-based queries. This service allows agents to register themselves based upon their application’s capabilities, and allows agents to discover other agents based upon queries for these capabilities.

For example, an agent that models an inventory warehouse might register in the yellow pages with an attribute-value pair of ‘role=inventory’. At runtime another agent could query the yellow pages to list all agents where ‘role=inventory’. This query may be limited by geographic or other application-specific constraints, such as “I prefer geographically close entries.”

In prior versions of Cougaar this could be implemented by using the JDNI-based naming service, which supported both white-pages-style and yellow-pages-style queries. For Cougaar 10 and beyond these two services have been separated since they have significantly different runtime characteristics.

As noted above, the yellow pages supports complex attribute-based queries which are more complex than white pages name lookups. These arbitrary attribute queries are more difficult to cache. The yellow pages also supports entries with custom data structures with detailed information, as opposed to the white pages’ limited network-address entries. Lastly, the yellow pages often has multiple application-specific concepts of locality that don’t necessarily match the white pages’ network-based layout, such as geographic locality as opposed to LAN/WAN locality.

Although the yellow pages supports more complex queries than the white pages, it is more difficult to scale and risks developer abuse. Many of the lessons learned in the white pages service also apply to the yellow pages. A scalable solution cannot view the yellow pages as a global database!

In Cougaar 10.0 the initial YPService API was defined. The API uses UDDI4J, with an extended API to support asynchronous queries and callbacks. A prototype in-memory yellow pages implementation has been developed.
Cougaar 10.2 supports better integration with UDDI4J and additional database back-ends. The primary client of the yellow pages is the Cougaar service discovery framework, which will help guide the design of the yellow pages. No major changes to the YPService external API have been made since 10.2.

4.1.3.1 Distributed YPServers

Cougaar 11.0 supports distributed, hierarchical YPServers for better scalability. As used by the Cougaar service discovery framework to find Provider Agents to satisfy Customer Agent queries, the distributed Yellow Pages service helps prevent bottlenecks and balance workload. An Agent’s information may be contained in multiple YPServers.

It is important to structure the YPServer hierarchy to reflect logical Agent groupings. The CommunityService is used to control the order in which YPServer’s are searched. Each YPServer and its immediate set of Agents must be grouped into YPCommunity recognized by the CommunityService. The search for a matching Provider Agent will start at the YPServer for the requesting Agent’s YPCommunity. It will progress to successively distant YPServers using the YPCommunity chain if there is no query match found. Because the search relies on the YPCommunity groupings, they should be organized along some Agent characteristic that reflects a customer Agent’s preference for a provider Agent. For example, YPCommunities in a society representing military units might be organized to reflect the organizational structure or unit support command assignments.

4.1.4 Service Discovery

The Cougaar service discovery mechanism is an alternative and supplement to the mechanism of specifying Customer/Provider relationships by Agent name. Instead, the Customer Agent searches for and selects a Provider Agent to form a relationship with. The service discovery mechanism uses two services, the RegistrationService and the RegistryQueryService. For details, see the Cougaar Developers’ Guide.

4.1.5 Local Discovery

Local discovery allows an agent to discover other LAN-local agents without knowing their names or IP addresses. This typically uses LAN-local IP multicast with UDP.

For example, a new node with a couple agents may be started on a rebooted host. The agents can send a generic “Is anybody out there?” multicast to find the other agents on the LAN. Each agent on the LAN then replies to the multicast with its name and network-address information. The multicast query could be enhanced to support filters, such as a filter to find the local white pages resolver for a subsequent white pages bind.

This has not been implemented in Cougaar (yet). For Cougaar 10.0 a developer can write a component that uses a non-Cougaar network protocol, such as JXTA or JINI, to discover other Cougaar agents. A Cougaar IP multicast would likely be implemented as a new message transport link protocol. A future Cougaar release may feature built-in multicast discovery support if there is sufficient open-source interest.

4.1.6 Peer-to-Peer Search

Peer-to-peer search allows an agent to discover resources that reside on adjacent (peer) agents. This typically uses a hop-based search mechanism, analogous to JXTA search or the Gnutella protocol.

For example, an agent could send out a peer-based request for a specific file named ‘test.mp3’. This request must be scoped by either the “time-to-live,” a peer-based hop count, or some other limit. The result of the search is a list of matching agents and perhaps additional information pertaining to this search.
Cougaar views this as an application that can be developed on top of the existing Cougaar message transport. Another option is to embed a separate application within Cougaar, such as a JXTA component.

4.2 MessageTransport

When a node starts an Agent, it makes available to the Agent a MessageTransportServer instance that provides MessageTransport and NameServer functionality. The MessageTransportServer instance is usually constructed on behalf of the node by the static methods of the Communications class. Communications uses System Properties to determine the class of MessageTransport to construct, create an instance, and then start it. The choice of MessageTransport usually implies a single specific related NameServer class and instance which is, in turn, constructed and started.

The MessageTransport class provides an API for sending messages to arbitrary Agents by name (MessageAddress, usually a ClusterId) and for registering a MessageTransportClient (usually an Agent) with the Transport so that it can receive messages from other sources. MessageTransport implementations are usually fairly complex in order to achieve good throughput to multiple peers which may vary considerably in distance/latency, bandwidth, and connectedness (e.g., periodically connected). The default RMIMessageTransport uses multiple queues served by a pool of threads to guarantee proper, in-order delivery of messages. Messages are always one-way and asynchronous—any response will be in the form of another message. For details, see CDG.

4.3 Inter-agent Communication

Under most circumstance, developers do not need to be concerned with the communications needed to support their application. This is true because logic providers handle communications. Developers may be aware that allocating a task to another agent initiates communications so that the other agent receives the allocated task, but the interactions needed for a particular application may not map gracefully onto the use of tasks. Cougaar has two features that address this issue: Relays and AttributeBasedAddresses. Relays provide a general mechanism for blackboard objects of one agent to have manifestations on the blackboard of other agents. AttributeBaseAddresses allow messages to be sent to agents based on their attributes rather than their names. These mechanisms are independent but frequently used together.

4.3.1 Task Allocation

Applications using the Planning Domain or derivatives may communicate with other agents transparently merely by allocating tasks to an Asset associated with an Agent. This mechanism may be duplicated in other domains using similar methods – in particular, a Domain’s LogicProviders may specify an arbitrary mapping between blackboard objects and messaging [see section 5.3.1].

4.3.2 Relays

Relays consist of two interfaces that blackboard objects can implement so that data from a source blackboard can appear on target blackboards and responses from the objects on the target blackboards can appear within the objects on the source blackboard. Objects on the source blackboard implement the Relay.Source interface while those on the target blackboards implement the Relay.Target interface.

The essential features of a Relay.Source are that it has a list of target addresses to which its content should be sent and that it has content to send. The source must also furnish a factory that can be used at the target to construct an object to be published to the target’s blackboard from the content object. The source implementation may also retain responses from the targets, but that depends on whether responses are necessary. If the application does not require responses, then the implementations need only have empty implementations of the related methods.
The essential feature of a Relay.Target is that it represents the content in some way. Beyond that, the target must furnish a response and the source address if responses are to be used.

It is perfectly reasonable for a single class to implement both interfaces though care must be taken to insure that the behavior of the object as a source on a target blackboard is acceptable and that the behavior as a target on the source blackboard is likewise acceptable. It is also feasible for a single class to also be used to represent the content and the factory. Such an implementation is quite natural when a Relay is considered to be an object that is simply replicated from the source blackboard to the target blackboards. Such implementations often need special serialization considerations as indicated above.

4.3.3 Attribute-based Addresses

Attribute-based Addresses are extensions of MessageAddress objects meant to specify the recipient(s) of a multicast message based on the attributes of an agent within a community. Most commonly, the attribute is the agent’s role in that community, but other attributes can be used. Attribute-based addresses are commonly used with Relay objects especially when multiple agents might have a given role and want to receive the same message. The AttributeBasedAddress class is used for this form of Attribute-based addressing, delivering sensor data among agents in a community. Such an address is constructed from the name of the community within which the attribute has meaning, the name of the attribute, and its value. This Attribute information is stored in the NameServer, and must have a corresponding context for AttributeBasedAddresses mapped to Agent addresses.

AttributeBasedAddresses are ideal for sensor messaging among sensors with the same attribute in a community. The manager/bin/mnrtest contains an example of a AttributeBasedAddress usage to publish and deliver health reports to all interested sensors (those with the same ‘Role’ value in the NameServer). For an example, see the CDG.

4.4 Blackboard

Where Agents communicate with each other point-to-point, Plugins communicate with the Agent, and via the Agent with the rest of the society through a data structure called the Blackboard.

The Blackboard can be thought of as a partitioned distributed collection of Objects that may or may not be of interest to any particular Plugin or even the Agent itself. The Agent communicates with its Plugins by adding objects to the Blackboard and then watching for responses. Plugins may, in effect, communicate with each other via the same mechanism. By design, the Plugin has no direct interaction with other Plugins and for a given message does not know which Plugin will process it or if that Plugin is in the same Agent or in an Agent on the other side of the globe.

More details on the Blackboard below in section 5.2.

4.5 Cougaar Persistence Architecture

4.5.1 Persistence Goals

Persistence is the mechanism whereby Cougaar agents can recover from failures when they operate over long periods of time in the face of power failures, 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. In addition to saving the plan-related information, the state of communication with other Agents may need to be saved. We use the term “persist” as a verb to denote this process of saving the state.
When an agent is started, its state is restored from the persisted data. This process is called “rehydration” for historical reasons. The restored state consists only of the blackboard service and certain other services. Most components can continue executing as if there had been no interruption. This element of invisibility is crucial to the ease of developing plugin components. It means that, after rehydration, plugins can continue operation as if no interruption had occurred. In particular:

- All the objects that had been published still exist.
- All objects that had been removed no longer exist.
- All additions, changes, or removals that had not been seen by any Plugin will be seen in Subscription lists after restart.
- All additions, changes, or removals that had been seen by any Plugin will not be seen in Subscription lists again.

If a plugin maintains no internal state except that which is published to blackboard with each transaction, it can continue execution after rehydration without regard for whether rehydration occurred or not. If a plugin does autonomously publish objects to the blackboard then it must exercise care to not repeat such actions if the objects are already on the blackboard.

4.5.2 Persistence Media Plugins

Persistence data can be stored on any reliable medium. Most commonly ordinary files are used, but other options are possible. Persistence media plugins allow flexibility in choosing a medium by presenting a common API to the base persistence mechanism. Several media plugins can be in use at the same time to provide storage diversity (avoid putting all the eggs in one basket).

4.5.3 Persistence Modes

Two modes of persistence are provided with two levels of rigor regarding maintaining the synchrony of the persisted state with the state of other agents. A number of terms have been used over time to name these modes. The less rigorous and most often used mode is usually called lazy persistence. It has also been called “optimistic” because it works well when crashes are few and far between. The most rigorous mode is usually discussed namelessly by contrasting it with lazy persistence, but it has been called non-lazy, conservative, or pessimistic persistence. Here we will call it conservative persistence.

4.5.3.1 Conservative Persistence

With conservative persistence, objects are logically persisted as they are published. As each subscriber closes its transaction, the distributor distributes the outbox envelope to all subscribers and persists the object and the new subscriber states. For efficiency, conservative persistence takes advantage of the fact that if execution inside the agent is invisible to all external entities (other agents or other external systems), then the persisted information does not have to be completely up-to-date with respect to the execution of the agent. This allows a number of distribution cycles to be executed before all the distributed envelopes and subscriber states are saved. This accumulation of information to be persisted is allowed for as long as there is at least one subscriber having work left to do or until a sufficient period of time has elapsed.

The communication with other agents must be carefully integrated with persistence. Messages can be sent to other agents only after the state changes that produced the messages have been persisted. Conversely, if a state change has been persisted, then the consequent messages must be reliably delivered. The MessageManager retains these messages on a retransmission queue to achieve both these ends. First, the messages are not actually sent until the antecedents have been persisted. Second, the MessageManager is itself persisted complete with its retransmission queues so that the messages can be reliably delivered upon rehydration.
4.5.3.2 Lazy Persistence

A difficulty with conservative persistence is that a useful society requires cooperation between its agents. A consequence of this is that an agent cannot compute for very long before inter-agent communication is necessary. The rules of conservative persistence require that the agents state be persisted before such communication can occur leading to an agent spending much of its resources in maintaining its persisted state rather than doing useful work.

Lazy persistence takes advantage of two facts: an agent does not often need to be restarted and the state of an agent’s interaction with another agent can be stored redundantly in both agents. This leads to a strategy that delays persisting the state while allowing inter-agent communication to continue. Doing so virtually guarantees that agents will be out of sync with the persisted states of other agents, but if a restart occurs, the redundancy will allow the agent states to be resynchronized.

The reconciliation of inter-agent inconsistencies must account for all the data objects exchanged between agents. These objects vary with the domain of the application and so the details are left up to domain specific RestartLogicProviders. The form of these logic providers is that each agent independently informs the other of the state of the objects that have been sent to the other or received from the other. Typically, the objects that were sent to the other agent are sent again. If necessary, the resent objects are marked as being resent to avoid confusion with newly sent objects. The identities of the objects that were received are sent and the other agent repeats the rescind messages that it had sent for those objects that are no longer relevant.

An additional advantage of lazy persistence is that it permits reverting to older persisted data in case the most recent information has been damaged. This is particularly useful when a diverse set of media plugins has been configured since a particular problem is less likely to simultaneously damage the different media.
5 Cougaar Agents

Agents are the first-class members of a Cougaar society. Agents communicate with each other point-to-point by sending messages via the node-level MessageTransport facilities. Agents are themselves completely generic: all Agents in a society will, for most purposes, have an identical code base and be instances of the same class. It should be noted that this is not required, allowing very large, long lived societies to evolve over time – where some Agents are utilizing newer or experimental code bases.

![Agent Diagram](image)

*Figure 5-1. Agent*

Some received Messages are handled immediately by the Agent (e.g., LoadPlugin), whereas most are queued and eventually passed to a LogicProvider, which responds by making the appropriate changes to the blackboard or plan.
5.1 Agent Internals

Agents have a number of internal services that help manage the Plugins.

5.1.1 Thread Scheduler

Plugins were designed to run relatively independently from the Agent, e.g., with their own thread of execution, using event notifications and alarms to run only when needed. This allows Plugins to run in parallel to the degree that the problem(s) they are solving may make use of parallelism. Plugins use pooled Java Threads, allowing the architecture to support a large number of agents without the overhead a per-plugin Thread, and allowing Cougaar’s ThreadService to manage thread use.

5.1.2 Timers

Each Agent has two Timers for allowing Plugins to request rescheduling at specific times: a “real-time” system clock and an “execution-time” planning clock. Plugins can use real-time Alarms to manage computer resources (e.g., to run an expensive simulation no more than every five minutes) or even external resources. Execution-time Alarms are based on a separate clock that may run offset from the actual system clock, faster or slower than one second per second, may be stopped, or may be advanced in steps rather than continuously. Plugins have simple access to this functionality through the “wake” family of methods on the standard Plugin base classes. Timers and Alarms have a millisecond-level granularity, but have no specific variance – that is, a Plugin will be wakened as soon after the alarm instant as possible as constrained by load, other Plugin function, etc. Timers are completely local to the Agent and are not intended to implement any sort of inter-Agent synchronization functionality. The infrastructure implementation assumes that the host system clocks are kept relatively (smaller than network delays) synchronized, presumably with an NTP or NTP-like facility.

5.1.3 ConfigFinder

ConfigFinder implements a file-based search facility. In general, Plugins (and other components) may invoke ConfigFinder with a relative path to a file. The ConfigFinder will search for that file in a well-known and extensible set of locations and return an input stream. The ConfigFinder implementation is capable of searching using any Java-known URL protocol, including “file,” “http” and “ftp” which allows pulling configuration information, or other data, over simple networking protocols as well as looking in the local file system. This facility should be used for finding initialization-time and other one-shot information, not as a general-purpose means of finding relevant data. JDBC, other database, or other distributed data APIs are more appropriate and likely to be considerably more efficient for random-access and multiple-query data retrievals. However, it is appropriate to use ConfigFinder to retrieve the keys or other contact information for accessing such a database. Plugins may find their ConfigFinder instance via their PluginBindingSite.

5.2 Blackboard

A Blackboard is an agent-local memory store that supports publish/subscribe semantics. Components within the agent can add/change/remove objects from the blackboard and subscribe to local add/change/remove notification. Agent domains monitor the local blackboard and can send messages to other agents and alter the blackboard when the local agent receives messages.

The primary benefit of an agent blackboard is that it abstracts the message transport from the plugins. 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. JavaSpaces or JMS) is a single point of failure and a considerable performance bottleneck.
Cougaar blackboards also support transactions, persistence, rehydration, and dynamic reconciliation. Additional details can be found in the Cougaar Architecture Guide.

All access to the Blackboard is transaction-controlled. Blackboard transactions cover only membership of objects in the logical collection—Transaction safety is not guaranteed for sub-object changes, only for addition and removal of objects from the Blackboard. In addition, the mechanism that delivers packages of Add and Remove events also delivers Change events (with details) which may be used to track sub-object-level changes in-band with Add and Remove events.

Blackboard transactions are interpreted by a set of objects called LogicProviders which add additional blackboard-level business-logic behavior to the simple changes committed by plugin transactions. LogicProviders have an additional support structure in the form of XPlans. An XPlan is a concrete object which provides concrete views of Blackboard collections in support of associated LogicProviders. Transactional behavior is implemented with a private, internal object called the Distributor. The Distributor is responsible for coordinating the interactions between Plugins, the Agent (and other higher-level processes) and persistence.

5.2.1 Subscribers

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. The plugin’s Subscriber is the service implementation returned by BlackboardService.

Subscribers actually do the difficult work in management of the following for their associated Plugin:

- Transactions
- Subscriptions
- Queries
- Publication of Plan changes

All of the Plugin base classes provide methods that delegate and/or add functionality to the Subscriber methods.

When a Subscriber receives a new Envelope of changes from the Distributor, 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) by updating the Subscriptions as appropriate.

5.2.2 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” messages 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 or transaction-controlled features).

Transactions both protect the Blackboard from asynchronous modification and provide a framework for tracking individual changes made or requested. Exceptions are thrown if any Plugin (or other component)
attempts to modify the Blackboard, publish any changes to objects, create any new subscriptions, make any dynamic queries, or request any information from existing subscriptions which is meaningless outside of transaction boundaries.

Subscriber has a set of Transaction methods which are used to explicitly open, close or check the state of transaction. 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.

5.2.3 Subscriptions

All access to the Plan 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 (Java Collection API) 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 transaction-sized update Envelopes.

5.2.3.1 Predicates

A Predicate is an implementation of the utility class UnaryPredicate that has an execute(Object) method which returns true if and only if the object should be considered part of the set.

Predicates are run often—it is a good idea for predicates to exit as quickly as possible and be as inexpensive as possible.

Predicates are only executed once per object change. This implies that Predicates should only test for unchanging features of objects. If a Plugin would like to select for mutable features of an object, it should subscribe using a predicate that selects for the immutable aspects of the test and then it should refilter on the mutable aspects. The org.cougaar.util package includes a number of efficient filtering and selection utilities (e.g., org.cougaar.util.Filters, org.cougaar.util.Mappings) designed to facilitate this process.

An experimental variation is being added in version 6.6 which allows direct subscriptions using predicates which select for any aspects, mutable or not. Essentially, the Plugin implements DynamicUnaryPredicate instead of UnaryPredicate to select this more flexible but more expensive processing.

5.2.3.2 Delta Lists

Most Subscriptions (including all which extend IncrementalSubscription) support the concept of “delta” lists. This allows a Plugin to ask the Subscription which members were added, removed or marked as changed since the last transaction. Plugins may use the getAddedCollection(), getRemovedCollection() and getChangedCollection() to retrieve this information from the Subscription.

5.2.3.3 Collection

Most Subscriptions (including all which extend CollectionSubscription) support requesting the “current” set of objects from the Subscription. The Subscription itself may be downcast to a Java Collection or the getCollection() method can be called.
5.2.3.4 ChangeReports

Any Subscription derived from CollectionSubscription provides access to detailed set of ChangeReports on contained objects since the last Transaction with the getChangeReports() method. If an object was publishChanged and no ChangeReport was specified, then a default AnonymousChangeReport is used.

5.2.3.5 Subscription Classes

A Plugin may choose the class and/or certain aspects of the Subscription to use when subscribing. The choices are available by using different patterns of the subscribe() method in the PluginAdapter base classes or in the Subscriber. The default is IncrementalSubscription, but there are a number of other possibilities:

- CollectionSubscription just tracks the contents and the detailed ChangeReports. Also allows specification of what sort of Collection to use internally. The internal Collection is often specified to be a class which keeps elements sorted, hashed, or otherwise arranged for the convenience of the Plugin.
- IncrementalSubscription adds add/remove/change lists to CollectionSubscription.
- Subscription is an abstract base class which may be used to implement any other desired behavior.

5.2.4 Queries

PluginAdapter allows subscription-less queries of the Blackboard via the query() method. Query is functionally identical to a sequence of subscribe, retrieval of the subscription results followed by an unsubscribe, albeit potentially much more efficient. As with all Subscription activity, a Query may only be issued within an open transaction and all Plugin processing stops while the Query is being processed. There is no infrastructure facility for issuing a Query against another Agent’s Blackboard. BlackboardQueryService is an alternative to BlackboardService which allows (indeed, requires) the use of query() without transactions. **Note that in either case, the information resulting from a query is in an undefined transactional state** – in particular, there is no guarantee that a query with Predicate P and a Subscription with Predicate P will match at any given time (though they are likely to do so).

5.2.5 Distributor

The Distributor acts as a router between the Agent’s Message system, the Blackboard and the Agent’s Persistence mechanism. All changes to the Blackboard’s state pass through the Distributor. This allows the Distributor to coordinate Transaction management and to implement correct and predictable persistence (see Section 5.6 for more details).

The Distributor accepts Messages from the Agent’s MessageTransportClient service and forwards them to the Blackboard for LogicProvider invocation. Further, it accepts Messages to be sent from LogicProviders via the Blackboard and forwards them back to the MessageTransport.

On the Plugin side, the Distributor accepts packets of Transaction deltas (known as Envelopes) and carefully distributes each Envelope of deltas first to the Blackboard and then to all Subscribers that might be interested in the changes. The Subscribers are then free to act upon the changes, potentially in parallel. Much as a running node is a collection of Agents that share a common substrate, a running Agent is a collection of very different Plugins that share a significantly larger substrate. The major communication channel between an Agent and its Plugins is the Blackboard.

5.3 LDM Infrastructure

The term LDM (Logical Data Model) is used both as a descriptive term for the Cougaar design for a data representation language and as an infrastructure object which is used to manipulate instances of objects.
described by that language. This section will discuss the infrastructure while 5.4 will discuss the language details.

![Diagram of LDM/Blackboard](image)

**Figure 5-2. LDM/Blackboard**

The LDM is logically divided up into one or more problem Domains.

The ClusterServesPlugin interface and the standard Plugin base classes allow Plugin instances to get a reference to an LDMServesPlugin object.

LDMServesPlugin provides references to a number of other facilities:

- Access to a number of Factory objects for creating LDM objects.
- A UIDSer" which supports generation of globally-unique keys for distributed objects.
- An asset prototype cache for sharing commonly-used prototypical asset references.
- A mechanism for requesting that LDM Plugins provide Properties for a new asset.

### 5.3.1 Domain

A Domain is a description of a problem-specific space of objects and the specific processes for dealing with those objects. Specifically, a Domain includes:

- A unique name. Domains are referred to by String names. All known Domains are loaded at node startup time by looking for System parameters of the form `-Dorg.cougaar.domain.<name>=java.class.name.of.NameDomain.” An instance of each domain class is created which is used to get the other information. Plugins do not usually have access to
the Domain objects themselves. There is a special “root” Domain that is always present that
provides a basic factory and set of services.

- A Factory. Each Domain publishes a Factory for creating instances of the objects of its problem
  space. If it defines new assets and/or new PropertyGroups, it should usually register the
  appropriate Factories for those objects with the RootFactory (which is usually used to construct all
  assets and PropertyGroups) during its initialize() method.

- A set of LogicProviders. Each Domain will have a set of business rules which relates its problem
  space objects with each other and with those objects of other related Domains, usually at least the
  RootDomain.

- An extension of XPlan. Each Domain may have a problem-space specific XPlan to extend the
  abilities of the Blackboard in order to support its Domain-specific LogicProviders, most often by
  maintaining indexed subsets of Blackboard objects.

5.3.1.1 LogicProviders

Whenever the Blackboard is modified or Messages are received, LogicProviders are run to allow bridging
between the infrastructure data objects (e.g., Messages) and the problem domain data objects (e.g., LDM or
Blackboard objects). LogicProviders are small classes that provide the “glue” between the various levels of
data abstraction, usually between infrastructure data constructs and the problem domain specific
Blackboard objects.

MessageLogicProviders

The infrastructure and its data objects (Messages, for instance) are intentionally problem domain neutral.
When the Agent receives a Message it doesn’t know how to process directly, it forwards it to the
Blackboard, which then runs all the MessageLogicProviders to handle the Message. Usually, some
MessageLogic provider will recognize the Message and will inject some change into the Blackboard,
perhaps adding, removing or modifying some Blackboard object.

An example of this type of LogicProvider is the ReceiveTaskLP that responds to a TaskMessage by
inserting or updating the Task object enclosed in the TaskMessage in the Blackboard.

EnvelopeLogicProviders

EnvelopeLogicProviders are invoked when Blackboard objects change. The charter of these
LogicProviders is to react to Blackboard changes by either causing other Blackboard changes or by sending
Messages, usually to other Agents. EnvelopeLogicProviders are invoked on a single Envelope Tuple which
is the description of how a single member of the Blackboard has (or should be) changed.

One example is the RemoteAllocationLP that reacts to changes or additions to Allocations of assets
representing Agents in the Blackboard by sending TaskMessages to the Agent represented by the asset.

Another example is the RescindLP which responds to objects being removed from the Blackboard and
propagates the rescind thru the object’s tree of related Blackboard objects. If a task is removed, the logic
provider will rescind (publishRemove) that task’s planelement. If a planelement is removed, the resulting
action described by the planelement will be rescinded. For example, if an expansion planelement is
rescinded, all of the tasks of the referenced workflow will be rescinded. Note that the Plugins are allowed
to “turn off” this automatic infrastructure rescind propagation by setting the isPropagating flag on
workflows and compositions to false.
Other LogicProvider uses

The class of the LogicProvider determines which types of events will invoke it. It is possible to write LogicProviders that implement both of the above interfaces and emit both Blackboard modifications and Messages.

LogicProviders are invoked on the smallest changes possible. It is important to note that LogicProviders have no visibility into whole Transactions—that is, they are invoked as part of the process of reconciling pieces of incoming Transactions with the Blackboard.

5.3.1.2 XPlan

The XPlan is a domain specific extension of the Blackboard. For example, the LogPlan is an XPlan specific to the logistics planning domain. Other domains can implement their own XPlan. An XPlan usually provides services that are useful to the logic providers of the domain. For example, the LogPlan subscribes to all PlanElements and provides methods for its logic providers to find and process specific PlanElements of the logistics domain that require special handling. In reality, many of an Xplan’s methods are delegated to the Blackboard.

5.3.2 Factory

A Factory is an instance of a class which creates instances of other (Domain specific) classes, often making use of the Context of the current Agent. The class Factory specifies absolutely no API: it is entirely up to the Domain implementer what functionality is available to Factory consumers.

The “root” Factory is an instance of the RootFactory class which provides factory methods for all the core infrastructure LDM classes as well as a number of convenience methods for use by other factories, for example to aid in the construction of non-core asset and PropertyGroup classes.

Factories are retrieved by Domain name from the LDM instance. If no domain name is specified, the RootFactory is returned.

5.3.3 UIDServer

UIDServer is an Agent-level facility for assigning globally unique identification codes (instances of the class UID) to Plan objects.

There is a standard interface UniqueObject which is implemented by many infrastructure Blackboard objects and all asset classes which requires that the object be permanently associated with a UID.

Usually this is handled entirely within the Factory: However, Plugins will occasionally require generation of unique keys with features which are similar enough to UIDs to warrant reuse.

5.3.4 Assets

Instances of the class Asset may represent prototypical objects (e.g., “1999 Subaru Legacy Outback Station Wagon”) or an actual, identifiable instance of such a prototype (e.g., “1999 Subaru Legacy Outback Station Wagon” with “Plate Number MA RW12345”). Identifiable assets usually delegate most of their properties to a shared Prototypical asset of the right type. To facilitate the sharing of such assets, the LDM allows Plugins to “cache” any prototypes which they create so that others can find them later.

There is a method on both the RootFactory and on the LDM, getPrototype() which first checks the prototype asset cache for the appropriate asset, and then invokes PrototypeProvider LDMPlugins until one is able to supply the right one.
The LDM also has a set of methods which allow Plugins to create assets (especially Prototypes) without knowing all the properties which apply. These methods allow the creator of an asset to request that all PropertyProvider LDMPlugins be called to fill in whatever PropertyGroups they can.

It is worth noting that the usual construction sequence of a prototypical asset is something like:

- A database Plugin (plugin1) decides it needs to create an “actual” asset of type T.
- plugin1 calls LDM.getPrototype(T).
- LDM looks in the prototype cache and fails to find anything matching T.
- LDM invokes a PrototypeProviderPlugin (plugin2).
- plugin2 constructs a new asset (proto1) of the right class.
- plugin2 calls LDM.fillProperties(proto1).
- LDM invokes a PropertyProviderPlugin (plugin3).
- plugin3 adds a PropertyGroup to the asset (proto1) and returns.
- plugin2 calls LDM.cachePrototype(proto1).
- plugin1 calls RootFactory.createInstance(proto1).
- RootFactory creates an instance of asset (asset1) which delegates to proto1.
- plugin1 adds asset1 to the Blackboard.

5.4 LDM Language Representation

This section describes the motivation for and the key components of the Logical Data Model (LDM). It does not contain detail on the specific application of the LDM to the logistics domain. The view of Cougaar from this perspective is a language for expressing complex data types by composing them from groups of properties or building them from simpler data types.

5.4.1 Objective of the Cougaar Logical Data Model

The objective of the Cougaar Logical Data Model is to provide a general mechanism to describe complex object types in a very large problem space covering assets and organizations/entities (a distinguished type of asset):

- **Assets**: Millions of different object types to be managed, with many varied properties, relationships and activities in which they participate. Assets include equipment, materiel, facilities, and a distinguished type of asset: organizations
- **Organizations**: (Entity is the more general, Planning domain version.) Tens of thousands of different special type organizations with their own physical plants, constraints and user requirements. Organizations are of many varied types, from large, distributed organizations with many component organizations down to individual people, and have varied relationships with one another

The purpose is to provide enough richness in the description of these objects so that the Cougaar components (agents) might reason about these things. Consequently it is necessary to have a mechanism to represent all the properties of these assets required to reason about them. Consistent with the large space of assets, the space of properties that must be covered is also large and detailed. These properties must describe the forms and functions of each asset required for reasoning. These property types include:

- **Functional Capacity** properties: (e.g., types and amount of equipment, materiel, or other product that can be produced, contained, stored, handled, transported, etc.)
Relational properties such as component properties (e.g., an asset is composed of a set of other assets) and consumption properties (e.g., rates of consumption of fuel, ammunition, other consumable materiel, etc.)

Physical properties (e.g., dimensions, weight, etc.) as it relates to the storage or transportability of an asset

Environmental Requirement properties (e.g., storage and operating temperature, humidity, etc.)

Reliability properties (e.g., failure modes, failure rates, etc.), and Repairability properties (required repair parts, required facilities, required skills, etc.)

Geographic and spatial properties (e.g., location, physical extent)

Skill and Capability properties

Relational properties (superior-subordinate relations and provider-consumer relations, component relations).

Compounding the complexity of the problem space, some of these properties, particularly for organization assets, may be time-varying. Further, the set of assets and properties evolves continuously over time. New models and types of equipment and materiel are continuously introduced, expendable material is consumed, current equipment modified and upgraded, and older equipment is retired. At even a higher rate, new organizations and organization types may be composed, constituted, deployed, and dissolved.

Another key requirement of the Logical Data model is that it must support reasoning about objects over a range of granularities, since varied amounts of detail are available (or even desired) at different times, or in different situations. In some circumstances, perhaps in addressing future plans, coarse grained granularity is appropriate: e.g., determining how many assets of what general types are required or available. In contrast, in other situations, perhaps in addressing current operations, fine-grained granularity is appropriate: e.g., determining the precise state of the particular asset instances that are currently in use.

Complementing the different granularities required in different circumstances, for a highly distributed set of varied computations on the “same” objects, it is important to recognize that different portions of the system may require different sets of specialized knowledge. Consequently it can be advantageous to provide different computational processes with different information about different aspects of the same objects. This allows multiple Agents to have views of an object particular to their perspective, and dramatically reduce the overhead of object maintenance for those elements of the object not relevant to a particular perspective.

5.4.2 Problems with Traditional Class-Based Object Oriented Approaches

The first problem that arises in using the Cougaar technology in a large problem domain like logistics is that single inheritance imposes a strict hierarchy on the object classes. This hierarchy imposes unappealing restrictions on the representation of objects and on the re-use of classes. For example, when dealing with deployable electric generators, it is necessary to decide whether a Trailer Mounted Diesel Electric Generator is an electric generator that can be towed, or a trailer that can generate electricity. Either case is clearly problematic without multiple inheritance: either a subclass of generator must be created that can be towed, or a subclass of trailer must be created which can generate electricity. In either case, the new subclass is unlikely to be re-used.

Another serious problem is that representing all the different types of assets as classes, and all instances of those assets as instances requires far too many classes to be practical. Furthermore, in this approach, the creation of new types of items requires the creation of new classes, and this code must be compiled along with any code that refers to these new types of items.

Using a traditional approach of compiled classes to represent different object types also presents problems in handling time-varying properties of assets, different granularities in descriptions of assets, and different sets of specialized knowledge about assets.
5.4.3 Cougaar LDM based on Prototypes and Property Groups

In response to these concerns with traditional class-hierarchy based object oriented approaches to modeling assets, the Cougaar Logical Data Model has chosen an approach based on dynamically composable prototypes with no asset hierarchy and a fairly rich, use-motivated set of descriptive objects called property groups and behavior groups.

The history of using prototypical objects (in contrast to classes) to implement object oriented systems has an interesting history, dating back to the mid 1980s. Some significant discussions contrasting these approaches may be found in:

- Lieberman-1986: Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems
- Leiberman, Stein, Ungar-1987: Of Types and Prototypes: The Treaty of Orlando
- Taivalsaari-1997: Classes versus Prototypes: Some Philosophical and Historical Observations

5.4.3.1 Basic Principles of the Cougaar Logical Data Model

The design of the Cougaar Logical Data Model rests on several principles.

- **Primarily based on properties, not on class.** Things are primarily modeled based on their Properties rather than what they Are. It does not matter whether a towed electric generator is a Generator or Trailer, as long as it has the Properties of a Trailer, and the Properties of an ElectricGenerator. Similarly, it does not matter whether a Tank is a Vehicle or a Weapon, as long as it has the Properties of a Vehicle and the Properties of a Weapon.

- **Related Properties are collected in Property Groups.** Experience has shown that attributes often occur naturally in groups. For example, Physical Attributes such as Length, Width, Height, Mass, etc. are often found together in databases, are used together as when packing items inside a container, and change together, as when an asset is modified. For these reasons, it has been helpful to collect related individual attributes into descriptive classes called PropertyGroups.

- **Some Property Groups encapsulate Behavior.** Behavior Groups are specialized property groups which encapsulate behavior about their specific properties.

- **Actual Instances Delegate their Properties to Prototype Instances.** This greatly reduces the number of classes required to represent the domain. Instead of each asset type requiring its own compiled object class, each asset type only requires its own Prototype Instance. This greatly reduces the number of classes required to represent the domain. Furthermore, it allows new asset types to be defined and created dynamically by instantiating new Prototype Instances.

- **The Class of a Prototype Instance determines the Property Groups which must be present in each Prototype Instance.** This provides regularity in the normal properties of related things. For example, all Trucks have the CargoCarrying Property.

- **A Prototype Instance may include additional Property Groups.** This provides flexibility in extending the properties of special types of things. For example, a truck with a refrigerated cargo space can have a ControlledEnvironment Property.

- **An Actual Instance may refer to specialized Property Groups which differ from the Prototype.** This provides flexibility in specializing the properties of particular instances of actual things. For example, a particular truck may have a larger fuel tank than other trucks of the same type. Also, actual asset instances typically have property groups which describe time-varying instance properties which are inappropriate for prototypes. These may include such things as location, maintenance history, etc.

- **Quantitative Property Group Attributes are Usually specified by Measure Class Objects.** In order to handle various Units of Measure for quantitative property group attributes, a number of Measure classes have been defined. Each class defines a particular measure dimension, such as mass, distance, etc. and can return values in a variety of units of measure.
5.4.3.2 Description of Classes, Instances, Property Groups and Delegation in the Cougaar LDM

Figure 5-3 shows a set of typical Cougaar LDM Classes, illustrating how trucks would be represented.

- The **Asset** class is used to represent any asset. It always has an attribute TypeIdentificationPG which is always present for prototypes, and which refers to an instance of the TypeIdentificationPG class, or an ItemIdentificationPG, which is always present for asset instances and which refers to an instance of the ItemIdentificationPG class. It also has an attribute OtherPG, which contains a list of all other Property Groups and Behavior Groups which describe a particular asset prototype. For maximum generality, Plugin code written to deal with assets, should in general not test whether a particular asset instance or prototype is an instance of any class other than Asset. These other, specialized classes, such as Truck, are intended strictly to be for implementation optimization only.

- The **Truck** class is used to represent asset types which are trucks. It is a subclass of Asset, and it is guaranteed to have the additional attributes TypeIdentificationPG, PhysicalPG, GroundSelfPropulsionPG, and ContainPG.

- The **TypeIdentificationPG** class specifies the identity of the type of asset being represented. It has attributes typeIdentification which typically specifies an NSN (National Stock Number), altTypeIdentification which typically specifies some other domain-specific identifier, and nomenclature, which provides a human readable descriptive string.

- The **PhysicalPG** class specifies basic physical properties of assets, including length, width, height, footprintArea, volume, and mass.

- The **GroundSelfPropulsionPG** class specifies properties associated with self-propelled ground vehicles. Its attributes include tractionType (Wheeled, tracked, etc.), fuelUseRate, fuelType, engineType, maxSpeed, cruiseSpeed, etc. Note that fuelType actually is an association with another prototype representing a kind of fuel.

- The **ContainPG** class specifies properties associated with things which contain cargo. Its attributes include maxLength, maxWidth, maxHeight, maxVolume, maxMass, and maxPassengers.

Figure 5-3. Typical Cougaar LDM Classes

Figure 5-4 shows instances representing two Asset prototypes (a 5-Ton Truck and a 2-1/2 Ton truck) which are instances of the Truck class, and a set of instances of PropertyGroup classes which represent the properties of these prototypes. The object instances which describe the 5 Ton Truck prototype are:

- The **5-Ton Truck Prototype** is an instance of the Truck class. Its attributes refer to Property Group instances which have values which describe the properties of the 5 Ton Truck prototype.

- The **5TonTypeIdentificationPG** specifies the NSN and LIN of the Prototype, and specifies its nomenclature as “M813A1 5 Ton Truck.”

- The **5TonPhysicalPG** specifies the Physical characteristics of the 5 Ton Truck, including its length (307 inches) and unloaded mass (20982 Lbs).

- The **5TonGroundSelfPropulsionPG** specifies some self-propulsion properties of the 5 Ton Truck prototype, including its tractionType (wheeled), fuelType (JP8), and maxSpeed (65 MPH).
• The 5TonContainPG specifies the ability of the 5 Ton Truck prototype to contain cargo, including the cargo maxVolume (1394 CuFt), the cargo maxMass (10000 Lbs = 5 Tons), and the maxPassengers (30).

Figure 5-4. Asset Prototypes and Property Prototypes

Similarly, the object instances which describe the 2-12 Ton Truck prototype are:

• The 2-1/2TonTruck Prototype is another instance of the Truck class. Its attributes refer to Property Group instances which have values which describe the properties of the 2-1/2 Ton Truck prototype.

• The 2-1/2TonTypeIdentificationPG specifies the NSN and LIN of the Prototype, and specifies its nomenclature as “M35A2 2-1/2 Ton Truck.”

• The 2-1/2TonPhysicalPG specifies the Physical characteristics of the 5-Ton Truck, including its length (265 inches) and unloaded mass (13180 Lbs).

• The 2-1/2TonGroundSelfPropulsionPG specifies some self-propulsion properties of the 5-Ton Truck prototype, including its tractionType (wheeled), fuelType (JP8), and maxSpeed (70 MPH).

• The 2-1/2TonContainPG specifies the ability of the truck to contain cargo, including the cargo maxVolume (984 CuFt), the cargo maxMass (5000 Lbs = 2-1/2 Tons), and the maxPassengers (20).

Figure 5-5 shows instances representing several individual 5 Ton and 2-1/2 Ton trucks. Each truck instance has just two key attributes:

• A reference to its prototype. All the 5 Ton Trucks share the same 5TonTruck Prototype, and all the 2-1/2 Ton Trucks share the same 2-1/2TonTruck Prototype.

• A reference to an ItemIdentification Property Group. Each truck has its own ItemIdentification Property Group.

It is important that the Truck item instances and ItemIdentification PropertyGroups are very lightweight instances. Consequently, the creation of a large set of trucks requires one prototype instance and associated property group instances for each type of truck, and then just one lightweight item instance and ItemIdentification PropertyGroup instance for each individual truck.
To review the workings of the LDM, it is instructive to see how properties of an individual asset would be obtained by the software. Given an instance of an asset, that instance would be searched first for a specific property, rather than all the 5 Ton Trucks, it is associated with the T830 Truck instance, rather than the 5 Ton Truck prototype instance. Figure 5-6 shows a modified 5 Ton Truck (Truck T830). It has a hoist mounted on the back, and can lift a maximum of 7500 Lbs. To represent this, an instance of a new LDM class, LiftPG, is created, which specifies the maximum Lift capacity of the truck. Since this is a property of the modified truck, it is associated with the T830 Truck instance, rather than the 5 Ton Truck Prototype instance.

Figure 5-5. Asset Instances

Creation of new an Asset type would require a new prototype instance and a set of Property group instances. However, it is also possible to modify individual assets to reflect modifications or state changes. Figure 5-6 shows an instance of a modified 5 Ton Truck (Truck T830). It has a hoist mounted on the back, and can lift a maximum of 7500 Lbs. To represent this, an instance of a new LDM class, LiftPG, is created, which specifies the maximum Lift capacity of the truck. Since this is a property of the modified truck, rather than all the 5 Ton Trucks, it is associated with the T830 Truck instance, rather than the 5 Ton Truck Prototype instance.

Figure 5-6. Specialized Asset Instance

To review the workings of the LDM, it is instructive to see how properties of an individual asset would be obtained by the software. Given an instance of an asset, that instance would be searched first for a specific property.
property. If that property is not found, it would delegate (or default back) to the prototype. In general, properties usually associated with types of assets would be found associated with the prototype, while properties associated with an individual asset would be associated with the asset instance. Consequently, to determine the Maximum speed of Truck T830, one would retrieve the prototype from the truck instance, and then retrieve the GroundSelfPropulsionPG from the prototype, and then retrieve the maxSpeed value from the property group.

5.5 Cougaar Plugins

A Plugin is a modular software Component which is added to an Agent to contribute a specific piece of an Agent’s business logic. Each Plugin adds domain-specific behavior to the Agent. The interesting or emergent behavior of an Agent depends primarily on the set of Plugins that are loaded into it. It is the aggregate behavior of all the Plugins in the Agent that determine how the Agent responds to the Messages it receives from its peer Agents.

Once an Agent has created a Plugin instance (e.g., as the result of an AddPlugin Message), the Plugin is a conceptually independent entity from the Agent.

Plugins are the essential “compute engines” of an Agent. Each Plugin provides unique capabilities, knowledge, and behavior that allow the Plugin to specify how to complete a given task. This specification is often represented by a workflow which allows the Plugin to specify implied tasks and constraints that describe the order the tasks should be completed as well as other information about how they should be completed. Plugins are self-contained elements of software that can be loaded dynamically into Agents. Plugins communicate only with the Agent’s Blackboard, reacting to events and publishing results. The Agent infrastructure schedules Plugins to execute when a registered change to the Blackboard occurs or when the Plugin has requested to execute at a specific clock time via an Alarm. Registered changes to the Blackboard include additions, removals or changes to Blackboard objects that the Plugin has specified an interest in (registered for) through via a Subscription. Note that Plugins are generally unaware of other Plugins, and therefore cannot be dependent on the presence of other Plugins. Plugins may be specialized so that an Agent operating in a specific domain and context will use only those Plugins that are relevant and specific to its operation.

5.5.1 Base Classes

The core infrastructure provides a number of convenient base classes of Plugin which may be used by developers to drastically reduce the effort required to write and maintain the interface to the infrastructure. The most important examples are:

- ComponentPlugin: The base class which should be used by all new plugin development. Supports all of the critical basics of Plugin infrastructure functionality including lightweight thread scheduling, subscriptions and parameters. This class is also useful as the basis for more application-level plugin base classes.

- PluginAdapter: The (now deprecated) base of all other Plugin base classes. Automatically hooks up to the Agent, maintains its own Subscriber and provides protected delegation methods to LDM, ClusterServesPlugin and Subscriber. Also provides a number of high-level and general-purpose utility methods. This class, and all extending base classes are used solely to provide backwards-compatibility and support for legacy plugins.

- SimplePlugin: Extends PluginAdapter (now deprecated) with selection of Threading model, adds a simple Plugin invocation API and manages Transactions automatically. Extending classes of SimplePlugin need only define two methods to be instantiable (and useful): setupSubscriptions() and execute(). Most Plugin developers extended this class in Cougaar releases prior to 8.4.
Plugin developers are encouraged to use ComponentPlugin or domain-specific extensions whenever possible.

### 5.5.2 Publication of Changes

A Plugin may request that changes be made to the log plan using the `publishAdd()`, `publishRemove()` and `publishChange()` methods (which merely delegate to the same methods on the Plugin’s associated BlackboardService instance). These methods may only be called while inside an open Transaction. Modifications requested in this manner will not be visible inside any overlapping Subscriptions (in the same Plugin or not) until the current transaction is closed.

- `publishAdd(o)` requests that the object be added to the Plan.
- `publishRemove(o)` requests that the object be removed from the Plan.
- `publishChange(o)` requests that interested subscriptions be notified that the object has changed. An optional second argument may be used to specify details about any changes which were made. Any details so supplied, in addition to any details collected automatically, are made available to LogicProviders, and to Subscription with a `getChangeReports()` method.

As all Subscriptions are views into the Plan, all interested Subscriptions are notified of any relevant publish activity. For example a `publishAdd(o)` in one Plugin will result in an “add” event to all Subscriptions with a Predicate that matches the added object.

Changes to the Plan may result in the activation of applicable LogicProviders which, in turn, may either make additional changes to the Plan or send messages to effect the Plan of one or more other Agents.
5.6 Cougaar Servlets

Cougaar includes support for handling HTTP and HTTPS requests by invoking user-developed server-side request handlers, called “servlets”.

Developers can use servlets to generate HTML views for browsers, send binary data back to a remote client, interact with local or remote Swing-based UI clients through HTTP, and other applications.

Servlet-based UIs are preferred to local UIs (console, Swing, etc) for many reasons; the most significant is that local UIs are tied to the machine's display, which makes the use of those UIs awkward when running a distributed (multi-machine) society. HTTPS-based clients also allow for better security than simple popup-UIs.

Servlets are similar to plugins: each agent has a separate set of servlets that can use a ServiceBroker to access the Cougaar services within that agent. Servlets are bound within an agent to a unique URL path, such as “/test”. Agents themselves are registered with a globally-unique URL-based “/$name”, such as “/$TRANSCOM”, that matches the naming-services registration. Together these create a globally unique URL-path to that agent's servlet: “/$TRANSCOM/test”.

All nodes in the society create web-servers with a unique “scheme://host:port” address, such as “http://foo.com:8800”. A remote HTTP-based client can send the “/$TRANSCOM/test” path to any server in the society, such as “http://foo.com:8800/$TRANSCOM/test”, and the request will be redirected to the node that's running agent “TRANSCOM”.

Components have access to a “ServletService” through the ServiceBroker, which allows the component to register and unregister servlets. A component can have its servlet as an inner class or as a separate class. Additionally there are some helper classes included in Cougaar to simplify the design, such as a component that loads simple servlets.

Figure 5-7. Cougaar UI Architecture
6 Component Model

A Component Model describes how modular software Components relate to each other: How one Component may contain another, and how one Component may request services from another.

There have been many Component Models proposed in various contexts. Cougaar’s interface is closely modeled on the Java Beans BeanContext API. However, the BeanContext API was designed to construct (UI) applications, so imposes requirements and restrictions which do not make sense for an Agent system. Cougaar also requires a level of security and managability which is generally redundant in the world of application design, so the BeanContext API alone is not sufficient for our use.

Cougaar starts with the basic interaction model of the BeanContext API and retains most of the terminology for Class and Method names. We then implemented the API without the UI-centric restrictions required by BeanContext.

One problem with most simple component models (BeanContext included) is that components usually have actual references (pointers) to each other. Depending on the implementation language, this can often allow components to abuse each other, e.g. by downcasting a component reference to a less restrictive type or by using language introspection capabilities. This allows component developers an unacceptable level of freedom to break modularity (at best) and break or corrupt the system (at worst). Cougaar adds an additional layers of insulating objects in between each pair of interacting Components: Binders between parent and child Components, ServiceProxies between server and client objects. These insulating objects may be anything from simple forwarding objects, merely delegating calls to higher-level objects, or may implement arbitrarily complex adaptive behavior. Furthermore, all interaction between a Component and higher levels of the Component hierarchy are mediated at some point by the Binder, so arbitrary levels of security, auditing or adaptation may be implemented.

The Cougaar ComponentModel is explicitly designed to function only within a single Java VM: there is no direct support for remote component-component relationships. While such remote relationships are possible to implement via proxy objects, incorporating such support into the core API was deemed too expensive for most applications.

6.1 Containment Model

Cougaar specified that all Components be organized into a strict containment hierarchy. Components which can contain other Components are called Containers. Containers may hold any number of child components which may themselves be Containers, but no Component may have more than one parent container.
6.1.1 Component

A Component is a pluggable software entity which has an existence and identity separate from any other (Analogous to a Java Bean). A Component may be required to implement a specific interface (informally known as the ChildAPI) so that the parent container and/or the Binder can invoke the child’s methods.

6.1.2 Container

A Container is a Component which can contain other Components. Containers are required to implement the java.util.Collections API to support adding, removing, etc. of child components.

6.1.3 BindingSite

A BindingSite is the API used by a component to access the services available to it via its parent. A Container will usually not implement the BindingSite used by its children - instead, it will present BindingSite-implementing proxy objects to each child to protect it.

The Service model allows extending the BindingSite api freely to add an extra communication path between Container and Child components. Most current Cougaar applications, however, use the BindingSite interface directly and rely solely on custom Services to extend the parent-child dialog. This pattern simplifies writing Binders which may apply at multiple points in a given Component structure.
6.1.4 Binder

A Binder is an object which acts as a proxy for a Component’s parent container. The Binder is an object implementing a BindingSite on behalf of a parent Container for use by one of its child components. Typically the BindingSite offers services to the child component and it protects the parent from the child. The parent component will often delegate child management duties to the associated Binders.

Binder implementations will often have privileged access to facilities offered by the Container which are not directly available to the child.
6.1.5 BinderFactory

A BinderFactory is an interface to be implemented by a Component which can construct Binder instances for child Components on behalf of a particular parent. BinderFactories are prioritized and provisions are made so that multiple BinderFactories may, in effect, produce nested Binders to wrap a given child.

6.1.6 ComponentDescription

A ComponentDescription is an object which describes and names a particular Component, e.g. in order to construct it. Most Containers will accept ComponentDescription objects as the argument to their add method. The Description contains enough information to figure out where in the Component hierarchy the Component should be inserted, to find and instantiate the desired Component class and to start it running. ComponentDescription is intended to be immutable and stateless and may be used (for instance) as a hashtable key as well as a recipe for creating a Component instance.

6.2 Service Model

Where the ContainmentModel imposes a very simple, very rigid, and relatively static set of relationships between Components, there is also a need for Components to develop client/server relationships with other components - these relationships are often complex (there are many types of Services), flexible (Services levels and capabilities may change over time) and dynamic (client-server Relationships may come and go freely). The Service Model defines these relationships.

6.2.1 Service

Service is an API for a facility which may be requested by a Component. Services are always Java interface classes - that is, a Service is named by the class of its java interface object.

6.2.2 Service Implementation

A Service Implementation is an implementation of a Service class constructed by a ServiceProvider, usually for a specific client component.

6.2.3 ServiceProvider

A ServiceProvider is an object, usually part of a Component, which can construct ServiceImplementations for client components. Components offer a Service by registering their ServiceProvider(s) with their ServiceBroker.

6.2.4 ServiceBroker

A ServiceBroker is an object which maintains a registry of ServiceProviders. It forwards requests for Services to the appropriate ServiceProvider. Logically, at least, each container in the Component hierarchy has its own ServiceBroker. Each level may define its own rules about transitivity of ServiceBroker requests: for instance, most standard ServiceBrokers will usually attempt to satisfy requests first in the local service pool and then will (recursively) submit the request up to the Container’s parent.

The BeanContext equivalent is called “BeanContextServices”.

6.3 Adaptive Behavior

The Cougaar component not only adds insulation/firewall between components and services, but allows the implementation of adaptive behaviors at several levels.
6.3.1 Binders

Binder implementations may implement their BindingSite methods in whatever way required, including implementing behavior tuned specifically to the bound Component, even which changes over time.

Of particular interest is that a Binder can supply a proxy ServiceBroker to the real one when the broker is requested by the component. Since all service requests are required to be satisfied via this interface, the Binder may control the broker completely.

A Binder is responsible for making all management, monitoring and control decisions about the bound component. It may make use of any available information to make such decisions, including use of other services (such as authentication and policy services) and may even be able to arrange to alter previous arrangements due to changing conditions.

6.3.2 ServiceBroker Proxy

A ServiceBroker Proxy is rightly considered part of the Binder’s codebase, and is certainly under its control. Such a proxy can, in effect, allow a Binder to veto specific service requests (e.g. limiting the component to only certain services), alter service requests (e.g. by adapting one service to fill the request for a different one), or by wrapping the service in a Service Proxy.

Note especially that since ServiceBrokers often will recursively pass requests up the component hierarchy, a single service query may actually be examined, adapted and/or proxied at every intervening level, allowing for proper nesting of domains of security and control. This sort of nesting can be accomplished for both client-to-server interactions with a wrapper around the higher-level returned ServiceImplementation (called a ServiceProxy) and for server-to-client interactions (e.g. callbacks) by supplying a proxy for the client (called a ClientProxy) to the higher-level service request.

6.3.3 Service and Client Proxy

A Service Proxy is the lowest level of control that a binder can exert, effectively giving the binder per-method invocation visibility into calls to a service. Service Proxies may restrict access to certain methods, may audit calls (and return values), or may even alter or augment calls. It is also possible for a Service Proxy to present one API when it is actually a proxy around a different API (for instance, by removing methods or by implementing one Service in terms of another: a tunnel). A Client Proxy performs the same job for the Binder, but for calls from the server to the client. Figure 6-4 illustrates a complex example where some of the Binders between the Plugin and the Messaging system interpose both ClientProxies and ServiceProxies to mediate the service interactions. Note that in most Cougaar Nodes, the hierarchy of components is somewhat more complex than this figure might suggest.
7 Cougaar Design Patterns

The Cougaar architecture described in this document is fairly open-ended: one can publish arbitrary objects to the blackboard, model arbitrary business processes, generate Plugins that perform arbitrary operations. However, as noted above, our experience has taught us that the Cougaar architecture is best used against certain types of problems and representing certain kinds of objects and relationships. Further, we see significant benefits in a particular approach to designing Cougaar Plugins, Agents and societies. Our approach is to use a model of computation based upon an understanding of how humans do planning, what we call the Cognitive Agent Architecture, from which the Cougaar name was derived.

It is important to understand that there are many models of computation used across the agent community, and in many ways one model is as good as another. Some bodies of research do not utilize computational models with some success. What a computation model provides is some overarching design and construction concepts that help reduce the complexity and diversity in order to make human understanding more tractable. When properly designed and utilized, computation models can be very effective meta tools in constructing extremely complex distributed applications that are logically decomposable and intuitively understandable. This effect greatly reduces developmental errors, testing time and long term maintenance costs. The model provided here has been developed and proven in the logistics domain and has successfully served all the objectives of a computational model.

This section describes some of the “lessons learned” from the Cougaar team in terms of proper ways to decompose problems into Agents and Plugins, and in general, a high-level design methodology for using Cougaar to model arbitrary problems.

7.1 Agent / Plugin Template Patterns

When humans try to solve a problem, we invoke one of several strategies over and over (iteratively and recursively), namely:

- Decomposing: Breaking a problem into smaller sub-problems
- Delegating: Giving some problem to a resource to solve
- Consolidating: Taking a number of independent pieces and handle them as a single problem
- Monitoring: Continually checking to make sure things are proceeding as planned, and correcting/reacting accordingly
- Gathering: Getting information from outside world
- Reporting: Reporting back to outside world
- Acting: Performing some action that impacts with real entities in real-time

When attempting to use the Cougaar architecture to develop a solution to a complex problem, we must design Agents and Plugins to match the problem. We find that modeling Plugins to match some step in the above “human cognitive process” to be a simple and powerful approach. We introduce the following general usage patterns, or templates, for Plugins:

- Allocator (Delegating): Allocate tasks to appropriate resources for final handling or further disposition
- Expander (Decomposing): Break down task into a workflow of sub-tasks
- Aggregator (Consolidating): Join a set of tasks into a single super-task
- Assessor (Monitoring): Assess PLAN for internal consistency, and force replanning when necessary
- LDM Plugin (Gathering): Read new/changed information from external data sources
UI Plugin (Reporting): Provide external user interface
Execution (Acting): Interact with external entities, objects, systems

Note that there is no requirement to break down the problem precisely along these lines. Occasionally a “hybrid” among these different patterns is justified, or some entirely different pattern is most appropriate for some aspects of a problem. However, the above patterns are simple and focused, which enhances maintainability and troubleshooting: the Plugin does one thing well. The Cougaar approach is to decompose a problem into simple, testable building blocks from which an Agent operation can be composed from the emergent behavior of the pieces.

At this point, we will provide some more detail and some examples of each of the above Plugin templates.

7.1.1 Task Decomposition Pattern (“Planning Domain”)

7.1.1.1 Allocator Plugin Template

An Allocator is a Plugin whose essential mode of operations is to take tasks and allocate them to resources (assets) that are capable of handling the task.

The logic of an Allocator is typically focused in several areas:

- What kind of task does this Plugin subscribe to (for allocation)?
- What kind of assets does this Plugin subscribe to (and manage the schedules for)?
- What is the business logic determining what assets are appropriate for what task?
- What is the nature of the allocation details (when, how much, etc.)?
- What are the dynamics associated with the allocation? What do we do if the preferences on the task change? What if the asset changes? What if the asset or the task are removed from the blackboard?
- What are the expectations in execution represented by this allocation: how do we capture those expectations, how do we determine deviation from those expectations and what actions do we take in response?

An Allocator typically works closely with the RoleSchedule of an Agent to determine availability of a particular capability over a particular time period.

It is worth noting that for a distinguished set of assets (OrganizationAssets), the act of allocating a task to that asset causes a LogicProvider to copy that task into the blackboard of the Agent corresponding to that OrganizationAsset. In this way, the process of Allocation/Expansion/Aggregation can be extended hierarchically beyond the borders of a single Agent.

7.1.1.2 Expander Plugin Template

An Expander is a Plugin that takes a task and breaks it down into subtasks. These subtasks are typically then handled by other Plugins in the same Agent. An Expander creates a workflow as well as all of the subtasks to represent interdependencies among the subtasks.

The logic of an Expander is focused in several areas:

- What kind of task does this Plugin subscribe to (for expansion)?
- What business logic determines what decomposition into subtasks is appropriate?
- What are the constraints and interdependencies among the subtasks to be captured on the workflow?
• How are preferences to be allocated among subtask preferences?
• How are allocation results to be aggregated from subtasks to the parent task?
• What are the dynamics associated with the expansion? What do we do if the preferences on the parent task change? What if the parent is removed from the blackboard?
• What are the expectations in execution represented by this expansion: how do we capture those expectations, how do we determine deviation from those expectations (in subtasks or the parent task) and what actions do we take in response?

7.1.1.3 Aggregator Plugin Template

An Aggregator is a Plugin that takes a set of tasks and bundles them together into a single aggregate task. The aggregate task is typically handled by other Plugins in the same Agent. An Aggregator creates a composition as well as the parent task to represent the relationship between the parents and the child task.

The logic of an Aggregator is typically focused in several areas:
• What kind of tasks does this Plugin subscribe to (for aggregation)?
• What are the business rules that set criteria for grouping tasks into a single task?
• How are preferences aggregated from the parent task to the child?
• How are allocation results distributed from the child task back to the parent?
• How is time ordering handled for aggregations? When do we consider an Aggregation “closed,” and when can we add a new element to an existing aggregation?
• How do we handle the removal of a parent task from the blackboard? Do we remove the aggregation and start over, or keep the remainder aggregated.
• As above, how do we manage expectations in an execution context, and how do we react to deviations?

7.1.1.4 Assessor Plugin Template

An Assessor is a Plugin that performs independent analysis of the blackboard contents looking for issues or inconsistencies and taking response actions when these are found.

Assessors may be looking for inconsistencies within the internal state of an asset (between its internal PropertyGroups and its allocations, e.g.) or between different assets, or different PlanElements. Alternatively, an assessor may be looking for particular conditions or patterns in the plan and taking particular actions when such conditions are detected.

Assessors may attempt, themselves, to correct a problem or inconsistency found in the blackboard. Alternatively, an assessor may warn a human operator of a potential issue requiring attention. More typically, however, an assessor will make a small change to the Plan to “undo” some questionable state, and force the original Plugins to reinvocation their original logic in a slightly newer, perhaps disambiguated, context.

Cougaar provides a standard Assessor called the TriggerManagerPlugin, which manages Trigger entities. These are objects that can be published to the plan, and indicate that a particular action is to be taken when a particular condition on a particular set of objects is met.

7.1.2 LDM Plugin Template

An LDM Plugin maintains the state of assets relative to some external data source. The Plugin may query the data source for information and represent some aspect of that data as assets in the blackboard. Further, the Plugin may monitor the state of the entity in the data source and update the state of the Cougaar-internal
asset accordingly over time, or operate in the reverse, updating the external data source as the information of the blackboard is modified.

Cougaar provides standard JDBC, SQL and XML LDM Plugins to interface with standard data representations of objects and map these representations into assets and objects in the Cougaar blackboard.

Another pattern for LDM Plugins is that of a sensor. Instead of mapping the schema of some external data source into parallel objects within Cougaar, an LDM Plugin may read events or measurements from a sensor into the blackboard, or it may translate these events or measurements into changes to specific elements (assets, or PlanElements) within the blackboard.

It should be noted that a subcase of the generic LDM Plugin Template is that of Execution Monitoring. As a plan moves into execution, there are datasources and sensors that pass in data that is expected to correlate with elements in the blackboard during execution. It is the job of the LDM Plugin to update the blackboard with the state of execution, both the state of blackboard objects and the state of the environment on which the assets and activities rely, as it becomes available. Often, these discrepancies may be then noticed by an assessor causing replanning by an Allocator/Expander/Aggregator.

7.1.3 UI Plugin Template

Plugins can use the ServletService to provide HTTP-based interactions between Users and Agents. Remote applications can contact Cougaar Servlets to support interactive User Interfaces. For details see section 5.6.

Output UI Plugins can write information about Plugin state to any output device, ranging from standard out/error to web-pages or tailored graphical displays. Input UI Plugins can read inputs from standard in, but typically have some graphical widget under control of the Plugin to capture user input (and possibly display output). In response to the user input, the Plugin may, in the context of a transaction, publish new objects, or changes to or removal of existing objects to the blackboard, causing other Plugins to react appropriately.

7.1.4 Execution Plugin Template

An Execution Plugin is one that commits actions to the outside world as a result of the execution of the Cougaar plan over time. Executing a plan from within Cougaar may force real actions to take place with respect to entities to Cougaar: emails need to be sent, transactions need to be committed to a live database, messages need to be sent to some external server, orders or instructions issued to human personnel.

These Plugins need to be sensitive to the irreversibility of their actions: unlike many concepts in Cougaar, an action to the outside cannot be undone merely by deleting it from the blackboard. Further, these Plugins need to be sensitive towards their role in “multiple world” scenarios. While a “reality” plan needs to be actually executed, a “hypothetical” plan should not be executed with respect to real-world entities.

7.2 Cougaar Design Methodology

Out of several years of working with the Cougaar architecture, a top-level design methodology has emerged over time. This methodology can serve as a helpful roadmap towards successfully mapping a particular domain and set of business processes onto Cougaar concepts. The following section describes the steps of the methodology. In the subsequent section, an example application of the methodology will be presented.

The top-level steps of the Cougaar design methodology are as follows:

1. Agent Enumeration
2. Role/Relationship Analysis
3. Plugin Enumeration
4. Publish/Subscribe Analysis
5. Task Grammar
6. Plan Element Map
7. Asset/Property Requirements Analysis
8. Execution Monitoring and Dynamic Replanning Analysis
9. Node Structure

It should be noted that these steps should be taken as points of a process that may require many iterations, not a simple single-pass. Further, we present this methodology with hopes that it proves a helpful framework for approaching the design process, but with the caveat that every design process is, by nature, improvised and unique. The following figure illustrates the Cougaar methodology, nominally like a waterfall model of sequential steps, but with an iterative/feedback nature.

Figure 7-1 Cougaar Design Methodology Workflow

7.2.1 Agent Enumeration

The first step of the design process is the identification of the Agents within the design. The Agents are the fundamental actors whose interactions and behaviors we are modeling in the society. What determines what makes for a good Agent? This decision is critical to the success of the design. If the lines marking off Agent boundaries are drawn too broadly, the Agent will be bloated, inefficient and a likely bottleneck of processing and bandwidth. If the Agent is too narrowly defined, it may require too much network traffic to communicate between its natural partners, and too much state will be shared among Agents.

Some broad categories suggest themselves for Agent definition:

- An Agent should represent the processes of a particular organization. Alternatively, an Agent should represent a discrete functionality, business process or algorithm.
- It should be possible to describe the top-level operations of an Agent with a simple sentence.
- An Agent should not have several clear distinct pieces that seem to operate independently of one another: if there are such functions, they are candidates for divisions into different Agents.

Note that Agents communicate with one another by messages and Plugins communicate with one another by sharing objects through publish/subscribe to the blackboard. Therefore, functions that will be sharing
detailed information (and not reporting synopses) should be lumped together in an Agent and represented as Plugins within that Agent, rather than as separate Agents.

### 7.2.2 Role/Relationship Analysis

Once the Agents of a given society are enumerated, the interrelations among these Agents must be defined and analyzed.

First, for each Agent, we must determine the roles or services they provide to other Agents. Agents identify and relate to other Agents by their declared roles and relationships with respect to one another. They do not typically, for example, look up other Agents by name. The designer must declare what services/roles the Agent will provide to other Agents, what services it will require, and which Agents will be providing these services.

It is important in determining roles and relationships among Agents to assume the potential for dynamic relationships. There may be multiple subordinates to a superior, multiple providers to a customer, multiple customers to a provider. These relationships will vary over time, and further, may be declared up front as time-phased relationships (e.g., X supports Y in a given role only over this given time frame). No assumptions should be made within Plugin code about the identity of a particular organization/Agent playing a particular role to that Agent.

When the role and relationship requirements and identities are established, it is important to compute a closure analysis. Does every Agent have (at least) one Agent who will provide the services it needs? Are there Agents providing services that no one requires? Either of these two cases may or may not be indications of a fault in the conceptual design or development configuration.

### 7.2.3 Plugin Enumeration

Once the Agents and their roles and relationships are established, the process of decomposing the function of the Agent into Plugins begins. We seek to decompose the essential functionality of the Agent until all pieces play a simple well-defined role. Where possible, we seek to map these pieces onto the Plugin templates described in the previous section.

It is important to keep the top-level behavior of each Plugin relatively simple and well defined. As noted with Agent design above, it is advisable to identify Plugins whose operation can be described in a simple sentence. One should rely on the interactions of other Plugins to do other parts of a chain of operations. For example, when an Assessor Plugin detects some inconsistency, it should not fix the problem itself. Rather, it should merely note an inconsistency by removing or modifying an allocation result. The original Allocator and Expander Plugins will then be invoked automatically to replan. In this way, the rules of consistency are maintained in one Plugin, the rules for planning are maintained in others.

In identifying Plugins, it is very important to try to identify opportunities to use generic capabilities. Cougaar comes with a rich toolkit of generic Plugins that are described elsewhere in this document. Where possible, these should be leveraged as the “glue” to hold together pieces of more tailored business logic. Additionally, in enumerating new Plugins that need to be developed, the designer is encouraged to identify opportunities to create generic or parameterized Plugins to allow many Agents to take advantage of a single Plugin development effort.

### 7.2.4 Publish/Subscribe Analysis

This step of the Cougaar design methodology drills down into the interactions between the Plugins and the blackboard within each Agent. Every Plugin defines its interface to the blackboard in terms of the objects to which it subscribes, and the objects it publishes. The designer of an Agent must define what these interfaces are for each Agent.
Once these are defined, a closure analysis must be defined. Does someone subscribe to everything that some Plugin publishes? Does some Plugin publish something that fills each subscription of each Plugin? Do multiple Plugins have overlapping publishes (that is, two Plugins publish nominally the same type of object)? Each of these situations are potential problems in the design configuration.

As noted elsewhere in this document, Cougaar encourages all Plugins to declare their interfaces to the blackboard explicitly in XML contracts. In so doing, the society composer can invoke tools to perform the above closure analysis at configuration time. Further, Cougaar can invoke run-time checks to ensure that these contracts are being followed in execution.

### 7.2.5 Task Grammar

At this point in the design, it is important to enhance the details of the tasks that are published and subscribed to the blackboard. The tasks are the fundamental statements of requirements, and set up the “channel” for discussing the satisfaction of those requirements (through allocationResults).

For each task type that will pass through a given Agent, declare:

- The task verb, representing the fundamental operation and task type for the requirement
- The task direct object representing the fundamental asset upon which the task is to be performed
- The task prepositions and indirect objects, representing operational constraints and modifiers as well as other related auxiliary assets to be considered in disposing the task.
- The preferences on all relevant aspects for the task: what are the important dimensions about which this Plugin has preferences (or guidance) about how the task will be disposed. Note that only by adding a preference can downstream processing report the allocation results on that aspect
- For each preference, a ScoringFunction must be provided, specifying the desirability and acceptability of different values across the space of a given aspect. What is the best (most desirable) value for a given aspect? What are the criteria for acceptable values for a given aspect? How important is it to fall in one point in the “acceptable” range versus another?
- A weight must be provided for each ScoringFunction to allow representation of relative preferences when normalizing Scores across all aspects to allow for comparing different prospective allocations.
- The auxiliary queries indicating a desire for report information on the allocation without any sense of “preference.”

As above, it is important to perform an analysis to ensure that all information on a task that SOME Plugin will require is provided by the tasking Plugin. Further, an inter-Agent analysis very much like the Publish/Subscribe analysis is advisable: what tasks will a given Agent send and receive? Is someone going to send the kinds of tasks this Agent takes as input? Is there an Agent who takes the kinds of tasks this Agent sends as output?

### 7.2.6 Plan Element Map

At this point, the Plugins are enumerated in an Agent and the essential task grammar is established. It is now possible to visualize the structure of PlanElements (inter-related Blackboard Objects) that will be built in a given blackboard of a given Agent. Allocator Plugins will allocate tasks to assets, and record these allocations in an Allocation PlanElement. Aggregator Plugins will aggregate multiple tasks into a single task, and record these allocations in an Aggregation PlanElement. Expander Plugins will expand single tasks into multiple subtasks and record these relationships (along with workflow constraints) in an Expansion PlanElement. The inter-relation of incoming and outgoing tasks should be visually mapped out so that the designers of the Plugins understand the piece of the solution they are building, and maintainers and UI writers understand how the whole structure of the blackboard of an Agent is intended to be structured.
Once the map is laid out, one should follow the forward and backward flow of tasks, preferences and allocation results through the Agent to ensure that there are no “gaps” or “dead ends.”

### 7.2.7 Asset/Property Requirements Analysis

We shift focus now from the tasks and PlanElements to the assets to which the tasks will be allocated. Assets are typically intended to reflect real-world entities, either concrete (like ships and bridges) or abstract (like organizations). It is important for the assets to contain a sufficient set of data attributes to allow Plugins to adequately reason about the assets to determine their suitability or allocation. Different Plugins may require different data, but the asset as it is represented in a given Agent should contain all the data that ANY Plugin will require for its processing.

First, we must determine what assets are owned or managed by a given Agent. What type of objects “belong” to the Agent, that is, the Plugins are responsible for managing the state of and allocations to these objects. Only one Agent can manage the allocations to a given asset at a given time (though different Agents may control the same asset over non-overlapping time periods).

Next, we must drill down on the requirements on these assets. What properties do the Plugins who will be processing these assets need? Identify these properties as possible into PropertyGroups for natural management and maintenance, and efficient composition of sub-components. Further, where possible, build on existing Cougaar PropertyGroups.

Finally, once the detailed asset model requirements are established, we must establish a source of data. What databases, sensors or other sources of information supply these properties that are required for the asset? Note that we may compose multiple incoming data streams into a single PropertyGroup, and certainly can mix different PropertyGroups from different sources into a single asset. Where possible, try to make use of Cougaar generic JDBC, SQL or XML LDM Plugins to create PropertyGroups, Prototypes and asset instances.

### 7.2.8 Execution Monitoring and Dynamic Replanning Analysis

Cougaar Plugins are expected to continually plan and replan in the face of updates, feedback, execution monitoring, and self-assessment. A Plugin should always be looking to update the plan, to keep it up to date, to keep it consistent, to try to make it better. To this end, we must ensure that the Plugins we enumerated and notionally designed above are made sensitive to this model of continual dynamic replanning.

A Cougaar Plugin should subscribe to all of the objects it creates, and many of the objects on whose state it depended to make its decision. When these objects change, it should wake up and determine if the change warrants action of some sort.

A Plugin must determine how to react to “bad news,” that is unacceptable allocation results or “failed allocations.” In the former case, it must determine acceptability criteria: how good is good enough? In the face of bad news, the Plugin may take one of several approaches:

- It may remove the allocation and allocate the task to a different asset (if possible).
- It may change the task preferences in hopes that the task will be allocated more successfully with different preferences/constraints.
- It may report the failure back up to its own superior or customer, and expect them to replan, possibly removing the task from the Agent’s own blackboard.

The designer of the Agent must determine the execution and execution monitoring behavior of the Agent. Are the LDM Plugins that created the initial set of assets capable of providing real-time updates? Are there sensors that can provide real-time feeds? How do we translate these reports into updates to state on
PlanElements or assets? What actions need to be effected in the real world on the basis of execution of the plan as time moves on?

Finally, the Agent designer should define an assessment strategy. Whose job is it to maintain consistency of the plan on the blackboard? In some cases, it will be the creating Plugin. For more complicated kinds of consistency, typically involving relationships among multiple objects, Assessor Plugins are likely called for, to determine the problem and instigate some corrective actions among the forward-planning Plugins.

### 7.2.9 Node Structure

This final step of the Cougaar design methodology should probably be considered as a part of a later part of the process, not the Agent design process. The allocation of Agents to nodes is an important decision of system engineering. However, it is irrelevant to the internal design of Agents. Agents don’t know and can’t tell what node they’re in, and what Agents are co-resident with them in the same node.

That said, the successful composition of Agents into an efficiently running society depends on building nodes that are efficient with respect to network requirements, and balanced with respect to limited computational resources.

Messages are sent between Agents when various actions are taken by Plugins within the Agent. For example, the allocation of a task to an OrganizationalAsset, the generation of an AssetTransfer, the setting of an AllocationResult on a downstream allocation all create messages between Agents. The node infrastructure ensures that these messages are passed efficiently as function calls for Agents in the same node, and using the MessageTransport network layer for Agents in other nodes. This implies that Agents that will be in frequent or high-bandwidth communications should be candidates for co-locating in the same node.

To illustrate the significance of co-locating (in the same machine or the same LAN) Agents that communicate frequently, consider the absolute physical constraints imposed by the speed of light. Even in the most optimal network environment, messages can never travel at greater than 186,000 miles per second (and usually, of course, travel at speeds well below this rate). Given this, it is never possible for more than 30 round trip messages to go between, say, Los Angeles and New York in a second. To that end, the quantity of message traffic between distant sites must be designed to be small and infrequent. Conversely, Agents that need to send large or frequent packets of data between them need to be in a physical environment that will support such traffic.

Another factor in the decision about node allocations is management of computation resources. Agents in the same node share the same memory and CPU resources. One should compose Agents into a node in such a way as to be confident that they will not overly tax the available memory and CPU resources. There is often an iterative process of allocating Agents to nodes, measuring loads and identifying bottlenecks and reallocating Agents to try to effect a more balanced load over the society.

Yet another factor in the decision to allocate Agents to nodes is the locality of physical resources or proximity to users. If a given Agent will be using a local database or file system that is only available from a given platform, it must run on a node on that machine. Further, consideration of the bandwidth requirements of communications with users should be considered. While Cougaar UIs are expected to be primarily web-based and build on the Servlet architecture, users who will be querying particular Agents with frequent or heavy queries may want to ensure that the Agents are on the same LAN as the UI clients.

### 7.3 Example Cougaar Design Process

We will now illustrate the use of the Plugin Template concepts and the Cougaar design methodology described above. Consider the problem of planning a proposal effort. We will use the Cougaar design methodology to design a Cougaar society to model this process.
Let us lay out the problem in more detail.

We have a company that writes proposals from time to time at the whim of management. There is a proposal team of tech writers, graphic designers and engineers that grinds them out, a review team among management and a production team of printers and shippers. The process tends to work like this:

- Management determines that there is a need to write a proposal to a particular customer for a particular piece of work by a particular date, say, June 10.
- Management tasks the proposal team to write the proposal and have a draft ready by June 1st.
- The proposal team breaks the proposal up into sections, assigns the sections and then groups them all together.
- The management team takes the proposal, reviews and edits it by June 7th.
- The proposal is passed to the production team who prints it by June 9th.

7.3.1 Methodology Example: Agent Enumeration

In this problem, we have a number of choices for breaking the problem down into Agents. There is one company: we could model everything in one Agent. There are, in fact, several players: managers, tech-writers, graphic designers, engineers, reviewers, printers, shippers, etc. We could represent each of these types as a different Agent.

A middle ground is probably best here. It seems that there are three fundamental groups who play a role here: management, the proposal team and the production team. They have their own assets (people, printers, etc.) and their own business processes. We would lose this natural decomposition and parallelism by joining them all into one Agent; by breaking them down further, we might incur too much network traffic by modeling the cooperation among the teams.

So let’s assume the existence of three Agents, as noted in Figure 7-2 below:

- Management: Generates the requirements for a proposal, delegates it to the proposal team, reviews it and releases it for production
- ProposalTeam: Takes the requirements, breaks them down into assignments and schedules
- ProductionTeam: Prints the proposal and ships it

Figure 7-2. Proposal Society
7.3.2 Methodology Example: Role/Relationship Analysis

In this case, the relationships are fairly straight forward. Management is a customer of two services, ProposalWriter and ProposalProducer, and there is a provider of each of these services to Management, namely, ProposalTeam and ProductionTeam. Note that we could have expressed these relationships in terms of Superior and Subordinates, with different Subordinate roles. Such an implementation would have been equivalent to what we represent here. The relationships are summarized in the following table:

<table>
<thead>
<tr>
<th>Agent</th>
<th>Roles/Relationships</th>
</tr>
</thead>
<tbody>
<tr>
<td>Management</td>
<td>(No explicit services provided: customer of ProposalWriter and ProposalProducer services)</td>
</tr>
<tr>
<td>ProposalTeam</td>
<td>ProposalWriter for Management</td>
</tr>
<tr>
<td>ProductionTeam</td>
<td>ProposalProducer for Management</td>
</tr>
</tbody>
</table>

7.3.3 Methodology Example: Plugin Enumeration

To perform the Plugin Enumeration for these Agents, let us take each Agent and decompose its functions in turn.

Management
1. Generates requirements for a proposal
2. Sets up a plan and schedule for the proposal: Write, Review, and Produce
3. Assign the Write, Review and Produce jobs to appropriate teams

ProposalTeam
1. Sets up a detailed plan for writing the proposal, breaking into sections
2. Assigns each section to an appropriate contributor

ProductionTeam
1. Sets up a plan for production: Print and Ship
2. Allocates the Print and Ship tasks to appropriate assets

Based on this decomposition, we see that these functions map neatly on to Cougaar Plugin templates as follows:

Management
1. Generates Requirements for a Proposal. This is a form of LDM Plugin. From some stimulus (a human UI gesture, a sensor that reads the web for RFPs, a database of leads, a random number generator…), the Plugin determines the need to write a proposal. It creates a GENERATE_PROPOSAL task into the blackboard of the Agent.

2. Sets up a plan and schedule for the proposal: Write, Review, and Produce. This is a form of Expander Plugin. We have taken he GENERATE_PROPOSAL task and broken it into subtasks (WRITE, REVIEW, PRODUCE) with time-preferences and constraints (we can’t review until the writing is complete, we can’t produce until the review is complete) to create a process workflow.

3. Assign the Write, Review and Produce jobs to appropriate teams. This is a form of Allocator Plugin. We have Organizations in our blackboard that provide particular services, (in this case ProposalWriter and ProposalProducer) and we have local assets (Reviewers) in our blackboard. We allocate the subtasks to these as appropriate. There is a Cougaar table-based Plugin that can perform these allocations by providing an XML script.

ProposalTeam
1. Sets up a detailed plan for writing the proposal, breaking into sections. This is a form of Expander Plugin. We need to take the WRITE task and generate WRITE_EXECUTIVE_SUMMARY,
WRITE_TECHNICAL_CONTENT, WRITE_COST_SECTION, etc. These are assumed (for simplicity) to be parallel efforts without interdependencies, but sharing the same deadline.

2. Assigns each section to an appropriate contributor. This is a form of Allocator Plugin. We will have assets to write these sections in our blackboard. As noted above, a Cougaar generic Plugin may handle this allocation.

ProductionTeam
1. Sets up a plan for production: Print and Ship. This is a form of Expander Plugin. We need to take the PRODUCE task and break it into subtasks of type PRINT and SHIP. In this case as above, there are constraints between these subtasks (we can’t ship before it is printed).

2. Allocates the Print and Ship tasks to appropriate assets. This is a form of Allocator Plugin. We will have assets to assign for printing and shipping, and we can use the Cougaar generic to handle this operation.

We have, then, the following table of Plugins required in our society:

<table>
<thead>
<tr>
<th>Agent</th>
<th>Plugin</th>
<th>Function</th>
<th>Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>Management</td>
<td>ProposalInitiationPlugin</td>
<td>Create initial GENERATE task</td>
<td>LDM</td>
</tr>
<tr>
<td></td>
<td>GenerateExpanderPlugin</td>
<td>Expand GENERATE into WRITE, REVIEW, PRODUCE</td>
<td>Expander</td>
</tr>
<tr>
<td></td>
<td>GenericTablePlugin</td>
<td>Allocate WRITE to ProposalWriter provider, PRODUCE to ProposalProducer provider, REVIEW to local asset</td>
<td>Allocator</td>
</tr>
<tr>
<td>ProposalTeam</td>
<td>WriteExpanderPlugin</td>
<td>Expand WRITE to WRITE_XXX sections</td>
<td>Expander</td>
</tr>
<tr>
<td></td>
<td>GenericTablePlugin</td>
<td>Allocate sections to local assets</td>
<td>Allocator</td>
</tr>
<tr>
<td>ProductionTeam</td>
<td>ProduceExpanderPlugin</td>
<td>Expand PRODUCE to PRINT and SHIP</td>
<td>Expander</td>
</tr>
<tr>
<td></td>
<td>GenericTablePlugin</td>
<td>Allocate sections to local assets</td>
<td>Allocator</td>
</tr>
</tbody>
</table>

Note that this description of the functionality of the Agents is to generate a PLAN to write a proposal. If we wanted to generate Agents to actually GENERATE the proposal, we would add Execution Plugins to perform the various tasks (WRITE, REVIEW, PRODUCE) at the appropriate times as time moves forward through the plan.

Further, note that the Expanders have a similar flavor, and there is an opportunity to implement all three from a common generic base tailored by some script.

7.3.4 Methodology Example: Publish/Subscribe Analysis

We need to ensure that the Plugins within the various Agents subscribe and publish consistent sets of objects. To that end, we produce the following table indicating what each Plugin for each Agent will publish and subscribe.
<table>
<thead>
<tr>
<th>Agent</th>
<th>Plugin</th>
<th>Publishes</th>
<th>Subscribes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Management</td>
<td>ProposalInitiationPlugin</td>
<td>GENERATE task</td>
<td></td>
</tr>
<tr>
<td></td>
<td>GenerateExpanderPlugin</td>
<td>Expansion(GENERATE), WRITE, REVIEW, PRODUCE subtasks</td>
<td>GENERATE task Expansion(GENERATE)</td>
</tr>
<tr>
<td></td>
<td>GenericTablePlugin</td>
<td>Allocation(WRITE), Allocation(REVIEW) Allocation(PRODUCE)</td>
<td>WRITE task REVIEW task PRODUCE task Allocation(WRITE) Allocation(REVIEW) Allocation(PRODUCE)</td>
</tr>
<tr>
<td>ProposalTeam</td>
<td>WriteExpanderPlugin</td>
<td>Expansion(WRITE), WRITE XXX subtasks</td>
<td>Expansion(WRITE) WRITE task</td>
</tr>
<tr>
<td></td>
<td>GenericTablePlugin</td>
<td>Allocation(WRITE XXX)</td>
<td>WRITE XXX tasks Allocation(WRITE XXX)</td>
</tr>
<tr>
<td>ProductionTeam</td>
<td>ProduceExpanderPlugin</td>
<td>Expansion(PRODUCE), PRINT, SHIP subtasks</td>
<td>Expansion(PRODUCE) PRODUCE task</td>
</tr>
<tr>
<td></td>
<td>GenericTablePlugin</td>
<td>Allocation(PRINT) Allocation(SHIP)</td>
<td>PRINT task SHIP task Allocation(PRINT) Allocation(SHIP)</td>
</tr>
</tbody>
</table>

Note that every Plugin that publishes a PlanElement (Expansion or Allocation) subscribes to that PlanElement so that it can monitor results returned from downstream processing or execution monitoring, and replan as necessary.

By means of this table, it is clear that we have no gaps or overlaps in the coverage of publishes and subscribes between Plugins in any Agent.
7.3.5 Methodology Example: Task Grammar

In this step of the methodology, we need to detail the contents of the tasks produced and consumed by the various Plugins. The following table summarizes the different tasks, their preferences, aspects, direct/indirect objects that flow through the society:

<table>
<thead>
<tr>
<th>Verb</th>
<th>Direct Object</th>
<th>Prepositions</th>
<th>Aspects/Preferences</th>
</tr>
</thead>
<tbody>
<tr>
<td>GENERATE</td>
<td>Customer</td>
<td>FOR_WORK(RFP Spec)</td>
<td>END_DATE (by June 10th)</td>
</tr>
<tr>
<td>WRITE</td>
<td>RFP Spec</td>
<td></td>
<td>END_DATE (by June 1st)</td>
</tr>
<tr>
<td>WRITE XXX</td>
<td>RFP Spec Section</td>
<td></td>
<td>END_DATE (by June 1st)</td>
</tr>
<tr>
<td>REVIEW</td>
<td>Proposal</td>
<td></td>
<td>END_DATE (by June 7th)</td>
</tr>
<tr>
<td>PRODUCE</td>
<td>RFP</td>
<td>TO (Customer)</td>
<td>END_DATE (by June 9th)</td>
</tr>
<tr>
<td>PRINT</td>
<td>Proposal</td>
<td></td>
<td>END_DATE (by June 8th)</td>
</tr>
<tr>
<td>SHIP</td>
<td>Proposal</td>
<td>TO (Customer)</td>
<td>END_DATE (by June 9th)</td>
</tr>
</tbody>
</table>

Note that all preferences have an END_DATE aspect. The ScoringFunction associated with each of these will represent a hard-deadline as illustrated in Figure 7-3.

![Figure 7-3. Deadline Scoring Function : “By June 10th”](image)

7.3.6 Methodology Example: Plan Element Map

In this stage of the methodology, we lay out a map of the different PlanElements in the various Agents, to see how the tasks coming into an Agent are ultimately handled (directly or indirectly). In effect, we are zooming in on the backboard to see the anticipated product of the various Plugins cooperating on their assignments.

Figure 7-4 reflects the decomposition of the incoming PREPARE task within the Management Agent. We have an expansion into WRITE, REVIEW and PRODUCE subtasks. The first and third of these are allocated to appropriate organization assets, and the second is allocated to local assets.
Figure 7-4. Plan Element Map for Management Agent

Figure 7-5 reflects the handling of the incoming WRITE task within the ProposalTeam Agent. We have an expansion into WRITE_XXX subtasks, each of which is allocated to local assets.
Figure 7-5. Plan Element Map for Proposal Team Agent

Figure 7-6 reflects the handling of the incoming PRODUCE task within the ProductionTeam Agent. We have an expansion into PRINT and SHIP subtasks, each of which is allocated to local assets.
7.3.7 Methodology Example: Asset/Property Requirements Analysis

In the course of this example, we have made assumptions about different kinds of assets being represented and available in the Agents. Let’s solidify these definitions and see what properties are required for these assets.

First, we have Organization Assets, which are proxies for remote Agents. The Management Agent has Organization Assets in its Blackboard that represent the ProposalTeam and ProductionTeam Agents. When a task is allocated to these special assets, Cougaar automatically copies the task to the plan of the associated Agent for further processing. The relevant property of these assets is the RelationshipSchedule, indicating what relationship the organization has to the Management Agent. In this case, the Management Agent needs to know what Agent supports it as a ProposalWriter or a ProposalProducer.

Further, we have human assets. In the Management Agent, we have reviewers. In the ProposalTeam Agent, we have different people with different skills for writing different sections. In the ProductionTeam Agent, we have different people who print and ship the product. With a little more effort, we could have easily modeled the actual printers as assets, and modeling the operations of the mail room, but modeling people with skills will suffice for this example.

Assume a new PropertyGroup that holds some training or skill specification, for simplicity, a string. (In our simple model, a person only has one skill). We can then use a Cougaar XML generic Plugin to fill the
blackboard with a set of people with specific skills. Then we could use a Cougaar table-based generic Allocation Plugin to indicate which assets with which skills are acceptable for allocating particular tasks.

This is a simple example of the use of assets. We could have used many different properties, structuring them into property groups. We could have used our LDM Plugin to create Prototypes of different people of different skill types, and used the LDM XML Plugin to create different instances of the prototype.

7.3.8 Methodology Example: Execution Monitoring & Dynamic Replanning Analysis

The planning we’ve been discussing in this example so far has been pretty static. If no requirements change and everything gets done on time then there’s no need to replan. We can imagine adding some dynamics into this problem and discuss how processing should handle these variations.

Changes to the incoming GENERATE TASK:

- If the task is removed from the blackboard of the Management Agent, the entire hierarchical plan will be automatically removed from all the blackboards by Cougaar, unless special provisions are made by the Expander Plugins to handle “rescinds” manually.
- If the task direct object or indirect object change, this is probably best considered a fundamentally new task, and the response should be to remove the old task (removing all traces of it through the society) and adding a new task with the new details.
- If the preferences for the END_DATE requirements change, we can propagate these changes down the Expansion/Allocation hierarchy. The Expander’s should wake up to the change and change the subtask preferences, recursively down the tree.

Failures to Allocate:

- If one of the Agents cannot find an asset to satisfy some incoming task, it must return failure information back up the Expansion/Allocation hierarchy. Ultimately, the failure of the entire plan should be reported to the top-level management, who can accept the failure, or change the preferences, or cause new assets to be transferred to the failing Agent.
- If the failure is not the lack of an asset but the lack of an AVAILABLE asset, the Agent may respond that it succeeded with a different date (assuming the ScoringFunction tolerated some time-slip). In this case, the tasking Agent could decide that this solution is good enough, or replan. Alternatively, the Agent could return a failure, but fill the failure with details of what it COULD have done if the preferences had allowed it, in which case the tasking Agent could change preferences to allow that alternative allocation.
- If a task is allocated to an asset but the allocation result is changed to “failure” due to some assessment function, the Allocator should invoke replanning, trying to find another suitable asset, or another acceptable time frame.

Changes to Assets:

- Imagine that we have the plan completely and successfully built. Then, some LDM Plugin may note that a person has declared vacation for the first week in June, and will not be able to complete their task in time. In this case, an Assessor could be written and invoked to check the consistency between vacation schedules and work schedules. In the case of a contradiction, the Assessor could change the status of an Allocation to that asset at that time to “failed.” In this case, the rest of the society should react much as above, and try to replan a successful allocation for this task.
7.3.9 Methodology Example: Node Structure

There is really not much to say for this simple example. It is likely that this problem is small enough to fit on a single node. The comments noted above still hold, however: there may be reasons to distribute these Agents among different nodes. Perhaps some Plugin needs to access a local data source, or perhaps one Agent contains a Plugin that uses up nearly all of the CPU. These are things that should be identified as the Plugins are developed and should be published as conditions for using the Plugins, making an informed node allocation easier. In any event, as Agents are composed and Societies are run, one needs to monitor the use of system resources and continually reallocate the hardware configuration as need arise.