Cougaar Developers' Guide

A BBN Technologies Document

Version for Cougaar 11.4

23 December 2004

Please e-mail comments to: cougaar-developers@cougaar.org

A copy of this document is available online at: http://www.cougaar.org

This document is designed to be printed double sided.

Cougaar Recent Release History

1 September 2001	8.4.
1 November 2001	8.6
11 January 2002	8.8
15 February 2002	9.0
1 May 2002	9.2
30 June 2002	9.4
16 December 2002	9.6
1 February 2003	10.0
19 May 2003	10.4
1 March 2004	11.0
5 July 2004	11.2
23 December 2004	11.4

Revisions in this Manual

Date	***Section***	***Change***
December 2004	3.2.5 Message Transport Service	Updated.
	3.2.7 White Pages Service	Updated.
	3.2.8 Yellow Pages	Updated.
	3.2.9 Service Discovery	Updated.
	3.2.12 Metrics Service	Updated.
	3.2.13 Alarm Service	Updated.
	3.2.15 Thread Services	Updated.
	3.2.20 Quality Objects (QuO) Support	New section.
	3.2.21 View Service	New section.
	5.2 Built-in Servlets	Updated.
	9.1 Cougaar Quick Start Guide	Updated.

Contents

Figures

1 Introduction

1.1 The Cougaar Project

The Cognitive Agent Architecture (Cougaar) project focuses on developing and maintaining an *open source* agent architecture for the construction of large-scale distributed agent-based applications. It is the product of multi-year DARPA research projects involving large-scale agent systems, and includes not only the core architecture but a variety of demonstration, visualization and management components to simplify the development of complex, distributed applications. Cougaar development was initiated under the Advanced Logistics Project (ALP) and continues under the Ultra*Log program, both sponsored by the Defense Advanced Research Projects Agency (DARPA). In ALP and Ultra*Log, DARPA is realizing the vision of a survivable, automated logistics system that incorporates leading-edge technologies from a variety of research initiatives and from evolving industry standards.

1.2 Cougaar Development Activities

One can categorize the Cougaar development activities into the following areas:

- The effort devoted to the creation of an agent infrastructure (managing components and providing services).
- The effort devoted to the creation of components that specialize the behavior of an agent to provide *systemic* capabilities such as survivability in terms of security and adaptivity.
- The effort devoted to the creation of Plugins that specialize the behavior of an agent to provide *domain* capabilities such as managing inventory or scheduling resources.

1.3 Scope of this Document

The intent of this document is to set forth the guidelines and specifications to which Cougaar Component developers shall adhere. All interactions between Plugins, Binders, Services, and other parts of the larger Cougaar system shall be through the defined Application-Program Interfaces (APIs) specified herein.

This document assumes that the reader is familiar with the basic concepts of the Cougaar Agent Architecture, which are introduced in the Cougaar Architecture Document (CAD).

- As the ALP and Ultra*Log programs have been concerned with the logistics domain, many of the examples and concepts described in this document draw from the military logistics domain. For further documentation visit the section 'References' at the end of this document.

There is a significant community of Cougaar developers, so there are a number of additional resources that developers should consult:

- New developers are encouraged to see the Cougaar tutorials, which include slides and exercises that introduce key Cougaar concepts: http://tutorials.cougaar.org/. New with Cougaar 11.4 comes th Pizza Party Planner application which is a simple 8 agent society that exercises many key Cougaar modules, while staying relatively simple. It demonstrates a little of what Cougaar can do, and key features.
- Frequently Asked Questions (FAQ). See http://cougaar.org/faq.html
- Basic familiarity with the Cougaar architecture is assumed, such as the concept of agents, blackboards, and plugins. See the Cougaar tutorial slides (http://tutorials.cougaar.org) or Cougaar Architecture Document (http://docs.cougaar.org).

- Community support mailing lists. See http://www.cougaar.org/mail/ for information on subscribing and archives. cougaar-developers@cougaar.org s a community support mailing list. Many developers of Cougaar applications and infrastructure read and post here. cougaar-information@cougaar.org is a forum for announcing software updates, workshops, training sessions, and conferences.

- Conferences and Workshops. Post a request to cougaar-information@cougaar.org if you are interested in attending a Cougaar conference, workshop or training. Watch for course or workshop announcements on the Cougaar web site.

- Bugs and Requests For Enhancements can be reported on the Cougaar Bugzilla web site at http://bugs.cougaar.org/

- The javadocs are available at http://cougaar.cougaar.org/software/latest/javadoc/

1.4 Document Overview

This document is organized as follows:

Section 2, Plugin Development, describes *"How to write a Cougaar plugin"*. The typical code structure of a plugin and its API are presented, as well as an example for typical plugin operations.

Section 3, Service Development and Reference Implementations, explains both *"How to use one of the existing Service implementations"* that come with the Cougaar release, and *"How to write a new Service"* that encompasses defining a Service API, writing a Service Provider, and adding the Service to a Service Broker.

Section 4, Binders and Binding Sites, addresses *"How to write a Binder"* that encapsulates a Cougaar component. It explains how components should interact with their parent components through Binders, discusses the implications for the development of child components, and presents a cookbook on how to write new binders.

Section 5, Web-based User Interfaces, explains *"How to write a UI for Cougaar"*. Based on examples, this sections illustrates how to build User Interfaces using Servlets, and how to install a Servlet Server for Cougaar.

Section 6, LDM (Logical Data Model) Structure and Usage, describes *"How to use the Cougaar Data Model"*. It explains how Cougaar Agents/Plugins can access and interact with data, and how assets and property groups are modeled.

Section 7, Notes on Persistence, explains *"How to use the Cougaar persistence mechanisms"*. The various persistence modes available in Cougaar and their impact on the application are discussed, and the necessary APIs are presented with some examples.

Section 8, Adaptivity Engine, explains *"How to instrument Cougaar components for adaptivity"* and *"How to control Cougaar components by an Adaptivity engine"*.

Section 9, Cougaar Administration and Programming Techniques, addresses a number of typical tasks of Cougaar developers like *"How to configure Cougaar"*, *"How to manage a Cougaar society"*, or *"How to debug in Cougaar"*.

Section 10, References, contains some useful links to background information and other documents related to Cougaar development.

Section 11, Glossary, explains fundamental Cougaar terminology and provides references to related terms.

2 Plugin Development

Plugins are the essential "compute engines" of each Agent. This section presents the information necessary to develop Plugins that interact via local and inter-agent blackboard operations. Subsequent sections present the architecture services that support Plugins, and developer control and configuration of the underlying architecture.

2.1 Definition and Function

Plugins are self-contained elements of software that can be loaded dynamically into Agents. Plugins interact with the Agent infrastructure according to a set of rules and guidelines and through a defined API. Plugins communicate only with the Agent infrastructure, reacting to Blackboard events and publishing results to the Blackboard. Plugins are unaware of other Plugins, and therefore cannot be dependent on the presence of other Plugins. Plugins may be specialized by domain and echelon so that an Agent operating in a specific domain and echelon context will use only those Plugins that are relevant and specific to its operation. In this way Plugins bring functionality to the Agent, while the society of Agents provides structure and order.

All Plugins must implement PluginBase and extend the GenericStateModelAdaptor. Most, if not all Plugins should implement the BlackboardClient interface as well in order to use the Blackboard. We expect that Plugin developers will deliver their Plugins in "jar" files following standard Java conventions. Plugins can "wrap" legacy software and applications developed in non-Java languages to conform to Cougaar interface specifications. However, we encourage all new development to use Java to promote portability.

A set of Plugin convenience classes is available for Plugin developers. These should be used whenever a new Plugin is being developed. The ComponentPlugin has been developed for Plugins being written under the new Cougaar Component Model. ComponentPlugin is a base class for writing small Plugins with a very simple initialization and runtime model. Subclasses need only implement setupSubscriptions() and execute(). Transaction handling for these methods is built-in. The former is used to set up any initial subscriptions, leaving subscription changes intact for the first call to execute(). Then it loops over standard in-transaction calls to execute() on any subscription activity. Subscription activity triggered outside the execute() method, such as GUI-injected tasks, must still be explicitly wrapped by transactions. Plugins can be developed using their own base class, but they must implement PluginBase.

2.2 Typical Code Structure of a Plugin

The following section describes the basic code sections needed to build a plugin that interacts with the Cougaar blackboard.

First, the plugin should extend ComponentPlugin, which is an abstract class that extends BlackboardClientComponent. These two classes provide basic services and APIs needed by plugins that will use the blackboard.

```
import org.cougaar.core.plugin.ComponentPlugin;
public class MyFirstCougaarPlugin extends ComponentPlugin {
```

Now the plugin should determine what services it needs. Some services are already provided by the BlackboardClientComponent base class, such as the BlackboardService, AlarmService, AgentIdentificationService, and SchedulerService. However, other services must be requested. Services can be requested directly from the plugin's service broker or the plugin can rely on load-time introspection to set the services for them.

To get the BlackboardService from your super class (BlackboardClientComponent):

```
BlackboardService myBBService = getBlackboardService();
```

To request a service from the service broker, place the following code in your load or setupSubscriptions method:

```
LoggingService loggingService = (LoggingService)
     getServiceBroker().getService(this, LoggingService.class, null);
```

To rely on load-time introspection to set up the service, which looks for public methods with the form “void setX(X)”, place the following methods in your plugin:

```
protected LoggingService loggingService;
public final void setLoggingService(LoggingService ls) {
   loggingService = ls;
}
```

Then the plugin should override the state methods provided by the infrastructure where necessary. These methods include initialize, load, start, suspend, resume, stop, halt and unload. Note that it is not necessary to override these methods if the plugin does not have specific work that needs to be done during these states.

```
public void load() {
    super.load();
    //initialize my special utility class
    myUtils = new PluginUtils(this);
    //initialize any collections
    myHash = new HashMap();
}
//release the plugins services when it is unloaded
public void unload() {
    super.unload();
    if (metricsService != null) {
      getServiceBroker().releaseService(this, MetricsService.class,
metricsService);
    }
}
```

Now the plugin is ready to decide what kinds of objects it is interested in or needs to complete its job. In order to collect a view of interesting objects, the plugin must define a predicate defining the object(s) of interest. The predicate will allow the infrastructure to fill in the plugin’s subscription with blackboard objects that pass the predicate restrictions. Plugins can specify multiple subscriptions that are each defined by a predicate.

```
// A simple predicate that specifies the plugin's interest in all Foo objects
// that contain a color slot that's value is blue
private static final UnaryPredicate MY_FOO_PREDICATE = new UnaryPredicate() {
  public boolean execute(Object o) {
     if (o instanceof Foo) {
       return ((Foo) o).isBlue();
     }
     return false;
  }
};
```

Subscriptions are requested in the setupSubscriptions method of a plugin.

```
private IncrementalSubscription myFooSubscription;
protected void setupSubscriptions() {
   myFooSubscription = (IncrementalSubscription) myBBService.subscribe(
      MY_FOO_PREDICATE);
}
```

Now the plugin is ready to use its logic and perform some work on the objects that it has subscribed to. All plugins run via their "execute" method. Plugins are usually scheduled to run (execute) when their subscriptions determine there has been a change. A change in an incremental subscription is defined as a new object to be added, an object that has been removed from the blackboard that was previously in the plugin's subscription, or an object that has been changed that is contained in the plugin's subscription. Plugins can also be scheduled to run via the AlarmService – for more details see section 3.2.

In the example code snippet below the plugin is responding to subscription changes.

```
protected void execute() {
  if (myFooSubscription.hasChanged()) {
     processNewFoos(myFooSubscription.getAddedCollection());
     updateChangedFoos(myFooSubscription.getChangedCollection());
     handleRemovedFoos (myFooSubscription.getRemovedCollection());
  }
}
```

At this point, the plugin methods will contain very specific Java code that completes their work. These methods may reference other utility methods or services. When the work is finished, the methods should publish any new objects, change notifications, and removal notifications to the blackboard using the Blackboard service. Plugins should be careful that the methods called from execute do not cause the plugin to hold their thread of execution for longer then necessary. Below is an example of a partially stubbed *processNewFoos* method.

```
private void processNewFoos(Collection addedFoos) {
   if (loggingService.isDebugEnabled()) {
       loggingService.debug("Processing new Foos");
   }
   // unpack the new Foo collection and process each one
   // for this example processing the Foos result in publishing a Bar
   Bar newBar = new Bar(x, y, z);
   MyBBService.publishAdd(newBar);
}
```

2.3 Discussion of “the Plugin API”

The basic APIs provided by the base plugin classes are listed below. Note that the code boxes below are not copies of the complete classes. Instead, they are code snippets taken from the actual classes to convey methods available to plugins that extend from these classes. For complete classes please look at the source code for org.cougaar.core.plugin.PluginBase, org.cougaar.core.plugin.ComponentPlugin and org.cougaar.core.blackboard.BlackboardClientComponent.

The PluginBase interface describes the minimum expected capability of all Plugins.

```
public interface PluginBase
  extends Component
{
  /** The insertion point for Plugins, defined relative to their parent,
PluginManager. **/
  public static final String INSERTION_POINT = PluginManager.INSERTION_POINT +
".Plugin";
}
```

The ComponentPlugin is the standard base class for Plugins.

```
public abstract class ComponentPlugin
  extends BlackboardClientComponent
  implements PluginBase
{
  public ComponentPlugin() { }
    /** Called once after initialization, as a "pre-execute()".*/
  protected abstract void setupSubscriptions();

  /**Called every time this component is scheduled to run.*/
  protected abstract void execute();

  // misc utility methods:
  protected ConfigFinder getConfigFinder() {
    return ConfigFinder.getInstance();
  }
}
```

The BlackboardClientComponent is the standard base class for Components that watch the Blackboard for activity and use the shared thread-scheduler.

```
public abstract class BlackboardClientComponent
  extends org.cougaar.util.GenericStateModelAdapter
  implements Component, BlackboardClient {

  /**@return the parameter set by {@link #setParameter} **/
  public Object getParameter() { }

  /** Get any Component parameters passed by the instantiator.
   * @return The parameter specified
   * if it was a collection, a collection with one element (the parameter) if
   * it wasn't a collection, or an empty collection if the parameter wasn't
   * specified.*/
  public Collection getParameters() {}

   /**Get the binding site, for subclass use.*/
  protected BindingSite getBindingSite() {}

  /** Get the ServiceBroker, for subclass use.*/
  protected ServiceBroker getServiceBroker() {}
  /**Get the blackboard service, for subclass use.*/
  protected BlackboardService getBlackboardService() {}

  /**Get the alarm service, for subclass use.*/
  protected AlarmService getAlarmService() {}

  /**Get the local agent's address.*/
  protected MessageAddress getAgentIdentifier() {}

  public long currentTimeMillis() {}
}
```

2.4 Exemplar View of Plugin Operations

This section provides a high level view of Plugin operations within an Agent taken from the Planning domain where the society processes tasks and workflows in order to generate a logistics plan.

Figure 2-1 is a linear presentation of Agent and Plugin activity as the Agent processes a single Task. Plugin developers should focus on the Plugins that are shaded in the figure.

1. The Agent receives a Task. An Expander Plugin that has a Blackboard subscription matching the Task is notified.

2. The Expander Plugin processes the incoming Task and generates a Workflow (a set of subtasks that must be accomplished in order to accomplish the original Task).

3. The Expander Plugin creates a PlanElement that contains the incoming Task and the Workflow it created. The Expander Plugin publishes the PlanElement and each new subtask to the Blackboard.

4. The Agent notifies all Plugins that have subscribed to any of the new Blackboard elements (Tasks). In this example, the Allocator Plugin has a subscription matching the new Blackboard element and is notified.

5. The Allocator Plugin allocates assets among *all* the tasks that have been expanded, both those from the newly created Workflow and the tasks from any previous Workflows.

6. The Allocator Plugin creates Allocation PlanElements, each of which contain a Task and the Asset assigned to perform the Task. An allocation may assign physical assets to accomplish the Task, or it may assign the Task to another Agent. The Allocator Plugin publishes the Plan Elements in the Blackboard.

7. The Agent infrastructure sends Tasks allocated to other Agents to those Agents.

8. The Assessor Plugin reviews the Allocations made by the Allocator Plugin.

9. The Assessor can send Directives to the Agent that originated the Task, if the Allocation score exceeds a specified threshold and action external to this Agent is required.

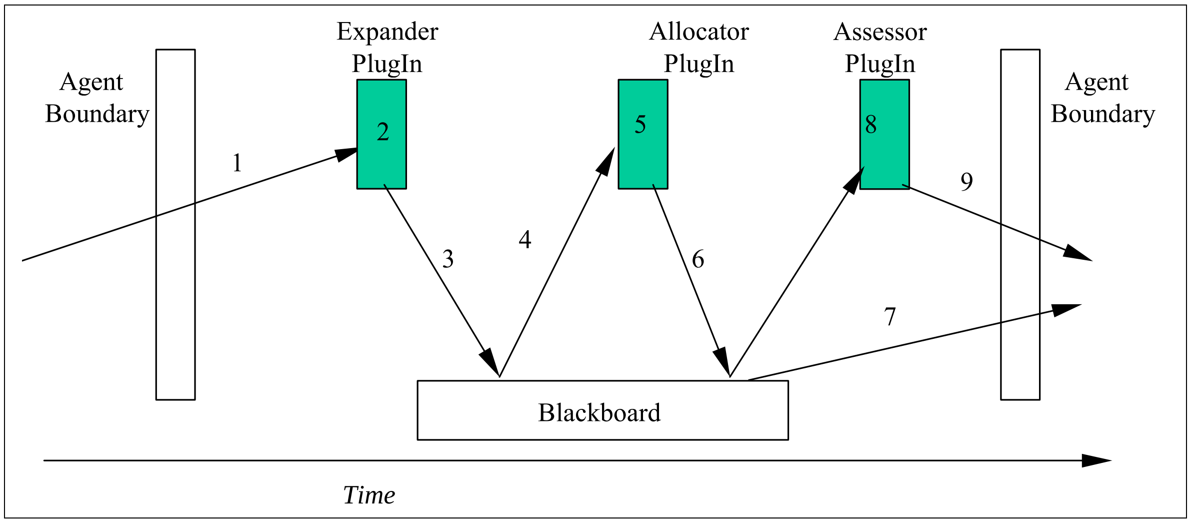


Figure 2-1: Plugin Operation Flow

2.5 Subscribers, Subscriptions and Persistence

Subscribers and subscriptions are used to access to the Blackboard. Plugins should request the BlackboardService in their load() method. Note that a plugin must implement BlackboardClient in order to successfully receive and use the BlackboardService. The ComponentPlugin base class implements BlackboardClient and has a BlackboardService that may be used by its subclasses. Alternatively, each Plugin may request its own BlackboardService and override the BlackboardClient implementation.

Plugins should not create their subscriptions until they are started. In terms of the state model, BlackboardServices are created as a consequence of the load() method being called. If the Plugin is derived from ComponentPlugin (most Plugins), subscriptions should be created in the setupSubscription() method.

When a subscription is created, it is immediately filled with objects matching the subscription's predicate. Consequently, already existing objects can immediately be enumerated with the elements() method of the subscription. In addition, some of these objects may be on the added list of an incremental subscription. There are two cases: If the BlackboardService has been marked as "should not be persisted," using the BlackboardService.setShouldBePersisted(false) method, then all elements of the subscription are on the added list. If the Subscriber/BlackboardService has not been marked as "should not be persisted," either by default or using the BlackboardService.setShouldBePersisted(true) method, then only those elements that have not been on the added list in some earlier incarnation of the agent will be on the added list.

A Plugin can find out if rehydration has occurred by calling BlackboardService().didRehydrate(). This method returns true if and only if the first transaction of the BlackboardService was closed and persisted. This means that any objects that were published during the first transaction of the BlackboardService are already in the Blackboard and should not be added again. Plugins that publish any initial objects (private state, assets, etc.) should not do so if didRehydrate() returns true. Instead, the Plugin should examine the elements of the subscriptions to initialize any cached state that the Plugin might maintain.

Some Plugins may keep track of information that is not otherwise available from the Blackboard. This "private state" will disappear if the agent crashes and restarts. Every instance variable of a Plugin class is potentially private state and may therefore vanish if the agent is restarted. In many cases, this private state can be recomputed from the contents of the Plugin's subscriptions and this is exactly what should be done. In other cases, the state is completely redundant with the subscription contents. Beware of code like this:

```
For (enumeration e = subscription.getAddedList(); e.hasMoreElements(); ) {
  tasksVector.add(e.nextElement());
}
```

This code is completely redundant because the tasksVector has exactly the same contents as the subscription.elements(). Variations such as storing the objects in a hash table are similarly pointless; the collection of the subscription can be chosen when the subscription is created to implement the desired functionality.

If, after considering the above, it still seems necessary to have private state, then by all means, use one or more private state objects. These objects should be created and published when the Plugin starts for the first time (see didRehydrate() above). If the Plugin is restarting, then a subscription that selects the state objects should be used to get the state objects from the Blackboard. Alternatively, a Blackboard query could be executed to find the objects. Be sure you can distinguish your private state object from other objects that might be in the Blackboard. Generally, a static inner class of the Plugin should be used. The class should be marked with the particular instance of the Plugin in the agent. The getBlackboardClientName() method is useful for this purpose since it incorporates the Plugin parameters. Each time the private state is modified publishChange() must be called. If the private state object is big and changed frequently, you will impose a huge burden on the agent. Find some other way to do what you need to do.

2.6 Predicate Language Extensions

Most Plugin developers will implement predicates in Java, as show in the prior examples. This section introduces an alternate predicate representation language, based upon s-expressions:

The design is a subset of Java, represented in a decision-tree style format. There are several built-in operators in the language:

- Logical (and, or, not, exists, any)
- Type checking (is:*, is:not:*, is:null, is:not:null)
- Constants (Strings, Java-primitives)
- Method/Field reference (heavily based on reflection!)

This is perhaps easiest to introduce by example. Suppose one wants to find all Tasks in an Agent that have a Verb of "Transport". The typical "org.cougaar.util.UnaryPredicate" implementation would be:

```
new UnaryPredicate() {
      public boolean execute(Object o) {
        if (o instanceof Task) {
          Verb v = ((Task)o).getVerb();
           if (v != null) {
              return v.equals("Transport");
          }
        }
      }
  }
```

The Predicate language representation will use "and"s, type casts, and reflection-based method calls to express an equivalent predicate as a tree. Graphically this would look like:

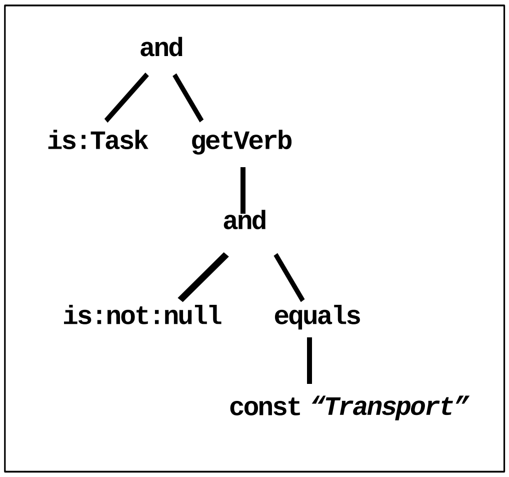


Figure 2-2: Predicate Represented as a Tree

Note that the Object enters the "and" at the top and is then cast to a "Task" on the left branch. If the Object is not a Task then the predicate returns "false". After the cast the method "Task:getVerb" is called upon the Task, which returns a Verb. Below the "getVerb" the Object is the Verb of the Task, which is subsequently checked to not be null and "Verb:equals(String)" is called with the constant String "Transport".

This predicate must be represented in either XML:

```
<and>
    <is:Task/>
    <getVerb>
        <and>
            <is:not:null/>
            <equals>
                <const>Transport</const>
            </equals>
        </and>
    </getVerb>
</and>
```

or in "s-expression" (parenthesis) format:

```
 (and
    (is:Task)
    (getVerb
        (and
            (is:not:null)
            (equals "Transport"))))
```

The servlet "/tasks" has an "Advanced Search" page that allows the user to type in these predicates and search the Blackboard. Over two dozen examples are included in the "/tasks" search page, which forms a tutorial of the language and points the user to the full BNF language specification.

The implementation uses a class type "org.cougaar.core.util.Operator", which is a sub-interface of "org.cougaar.core.util.UnaryPredicate". The Operator interface defines several extensions to UnaryPredicate, including XML generation and a "boolean implies(Operator)" method for analysis.

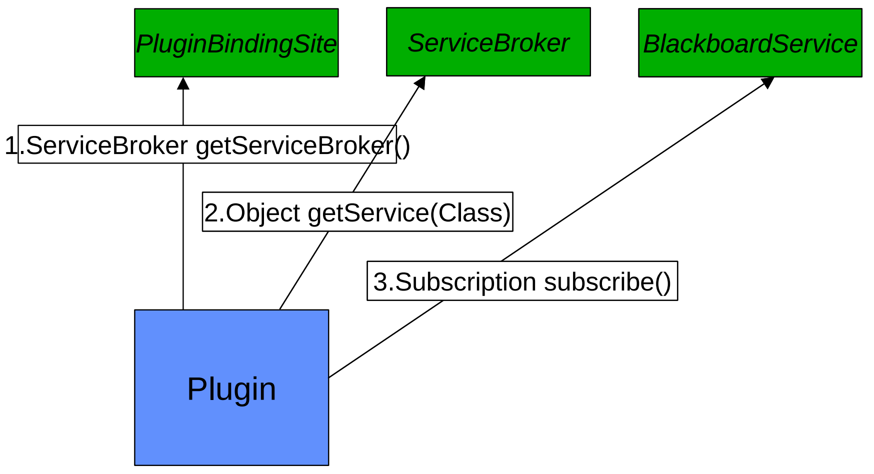
3 Service Development and Reference Implementations

3.1 What is a Service?

A Service is an API that provides facilities that are requested by a Component through a ServiceBroker. ServiceBrokers maintain registries of ServiceProviders which can construct specific service implementations for a client component. Services allow components to interact with other components and the infrastructure. Two examples of a Service are the BlackboardService and the MessageTransportService.

Note: as of version 10.0, all core Cougaar code use the BindingSite interface directly rather than extending it for each Container/Child relationship (e.g. PluginBindingSite). The following discussion has been left using the original terms since the current implementation represents a less complete example of the flexibility of the Component/Service model. The infrastructure's use of the Component model was simplified to make it easier to write Binders and Components that may be inserted at different locations. In situations where this is not required, custom BindingSites may still be used freely as discussed below.

The diagrams below depict the relationships among Components, ServiceBrokers and Services as well as example code for a Component requesting a Service. It is important to note that a Service request can be denied or proxied at almost any level of the component hierarchy.


PluginBindingSite
ServiceBroker
BlackboardService
1.ServiceBroker getServiceBroker()
2.Object getService(Class)
3.Subscription subscribe()
Plugin

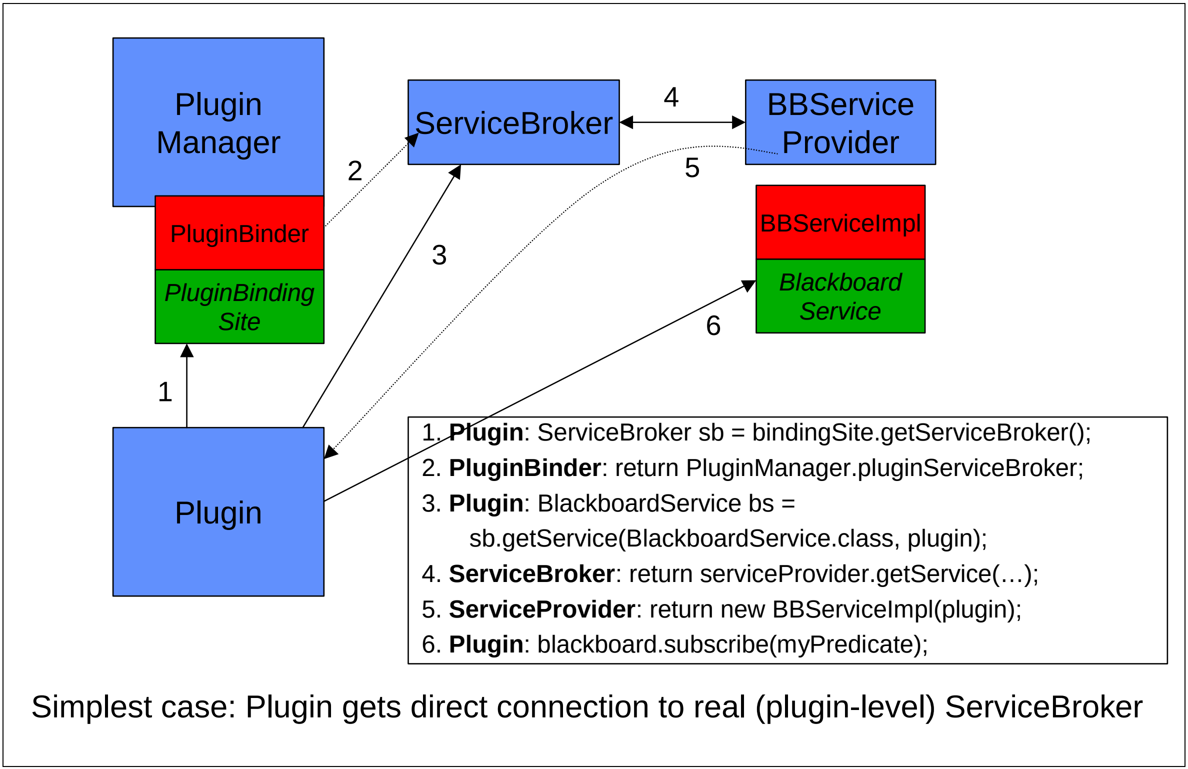
Plugin:

1. ServiceBroker sb = bindingSite.getServiceBroker();
2. BlackboardService bbs = (BlackboardService) sb.getService(BlackboardService.class);
3. Subscription mystuff = bbs.subscribe(myPredicate);

Figure 3-1: Service Model: Component View

In the example above, a plugin requests the ServiceBroker from its (parent) BindingSite. Once the plugin has a reference to the ServiceBroker, it requests access to a particular service, as named by the service class. The getService request is answered (unless service access is denied) by returning a service object for which the plugin can invoke the service methods.

Note that the binder need not respond with the parent's actual ServiceBroker - it may return a proxy, a more complex implementation or even may veto the request (forcing the child to rely on other methods of service discovery). The ServiceBroker implementation may respond directly or delegate the request to another broker and the service implementation may also act on the request directly or delegate to another object.

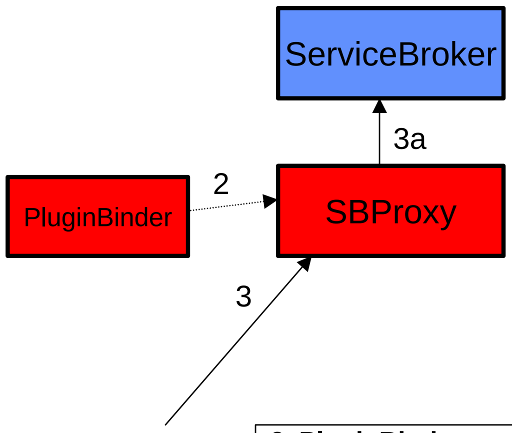


1. **Plugin**: ServiceBroker sb = bindingSite.getServiceBroker();
2. **PluginBinder**: return PluginManager.pluginServiceBroker;
3. **Plugin**: BlackboardService bs =
   sb.getService(BlackboardService.class, plugin);
4. **ServiceBroker**: return serviceProvider.getService(…);
5. **ServiceProvider**: return new BBServiceImpl(plugin);
6. **Plugin**: blackboard.subscribe(myPredicate);

Figure 3-2: Direct Connection to Service

The diagram above describes further details about the steps between the ServiceBroker receiving the Service request and the request being answered. Once a ServiceBroker receives a Service request, it finds the registered ServiceProvider that provides the requested Service and passed along the request. The ServiceProvider then decides what to return to the client. In this case it creates a new instance of BBServiceImpl and returns this object in the form of the BlackboardServiceAPI.

The next two diagrams depict some of the service request regulation points available in the Component Model.



2. **PluginBinder**: return myServiceBrokerProxy;
3. **Plugin**: BlackboardService bs =
       sb.getService(BlackboardService.class, plugin);
3a. **SBProxy**:
   if (isAuthorized(BlackboardService.class))
     return realServiceBroker.getService(BlackboardService.class,
plugin);

- Plugin gets proxy ServiceBroker which can veto requests.
- Proxy ServiceBroker can also transform requests or proxy the *client.*

Figure 3-3: Connect-Time Regulation

A ServiceWrapper is an object which implements the Service API by delegating calls to another implementation of the service. Essentially 3a becomes something like:

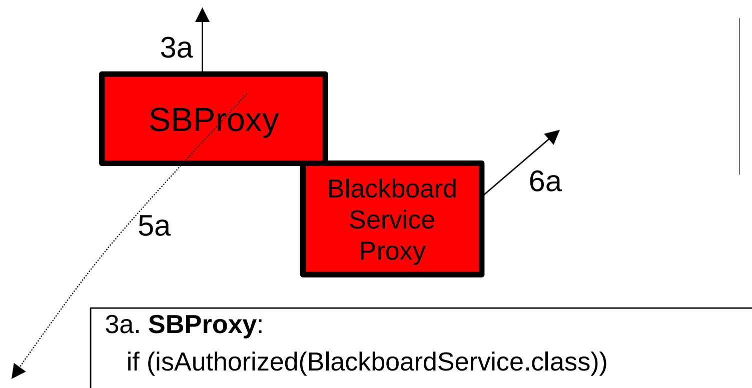
```
    service = realServiceBroker.getService(BlackboardService.class, plugin);
    return  new ServiceProxy(service);
```

A ClientProxy is an object which wraps the *client* for service request, essentially allowing callbacks to the client to regulated. 3a becomes:

```
   return realServiceBroker.getService(BlackboardService.class,
      new PluginProxy(plugin));
```

A ServiceAdapter transforms a request for one service into another request. In practice, adapters will often also require the use of a ServiceWrapper and ClientProxy. 3a becomes:

```
   otherService =
realServiceBroker.getService(AlternateBlackboardService.class,
     new AlternatePluginProxy(plugin));
  return new AlternateBlackboardServiceAdapter(otherService);
```



```
3a. SBProxy:
   if (isAuthorized(BlackboardService.class))
      return wrapService(realServiceBroker.getService(…));
5a. SBProxy.wrapService:
    return new BlackboardServiceProxy(realService);
6a. BlackboardServiceProxy.subscribe():
    if (isPredicateAllowed(predicate)) return realService.subscribe();
```

Service Proxy case: Binder's ServiceBrokerProxy wraps the service.

Figure 3-4: Invocation-Time Regulation

3.2 Existing Services

All Services must be registered with a ServiceBroker. Each Component may have their own ServiceBroker and define their own rules regarding the transitivity of ServiceBroker requests. The current Cougaar architecture contains Services that are registered at the top-level “root” ServiceBroker and per-agent ServiceBrokers. These ServiceBrokers are designed to look locally for qualifying ServiceProviders and if none are found, the broker will request the service from the parent Container.

In this section, we present the services that are included with the Cougaar architecture and loaded by default. Figure 3-5 presents an overview:

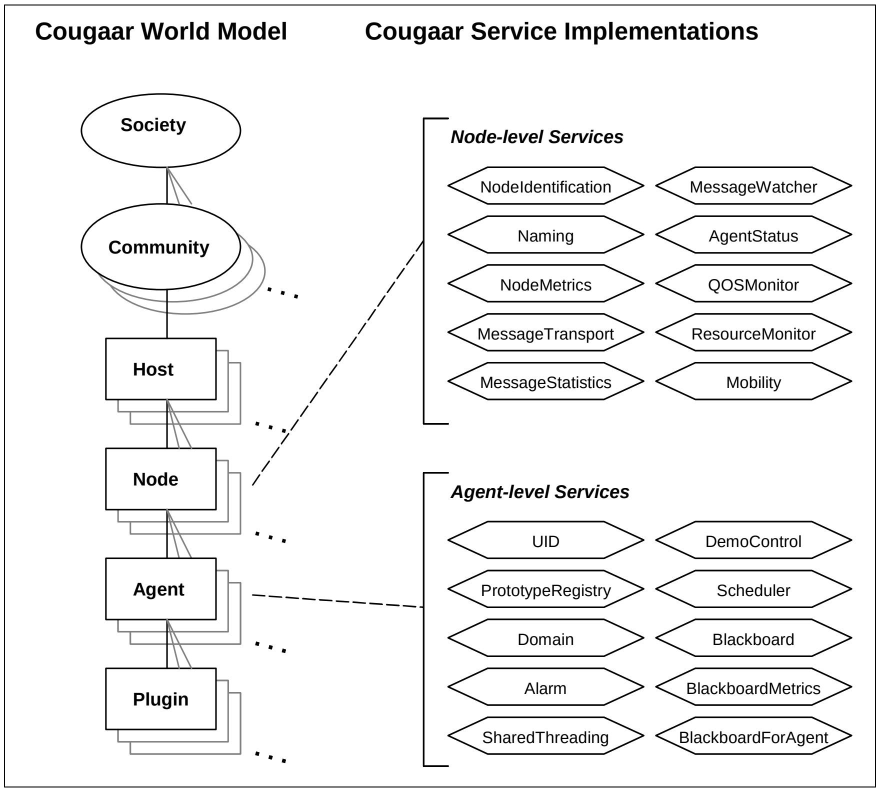


Figure 3-5: Service Implementations Overview

All of these services are specified in XSL node and agent template files that can be modified to add/remove core services. For details, see section 9.1.17.

3.2.1 AgentIdentificationService

AgentIdentificationService allows all components in an Agent to discover of which Agent they are a subcomponent.

```
package org.cougaar.core.service;
import org.cougaar.core.component.Service;
public interface AgentIdentificationService extends Service {
  // return the MessageAddress associated with the Agent.
  MessageAddress getMessageAddress();
  // return the (human readable) name of the Agent.
  String getName();
}
```

3.2.2 NodeIdentificationService

This service provides the MessageAddress of the Node that the calling Component is a member of.

```
package org.cougaar.core.node;

import org.cougaar.core.component.Service;

public interface NodeIdentificationService extends Service {
  public MessageAddress getNodeAddress();
}
```

3.2.3 Logging Service

Cougaar provides a standardized logging service to all components. This shared logging service is used extensively by Cougaar plugins, components, and service providers. The level of logging can be controlled at start-time and at run-time. Third-party tools can be used to view and filter the logs.

Each log entry is tagged with the component's classname, timestamp, a description message, and a logging level. There are 7 logging levels:

```
DETAIL < DEBUG < INFO < WARN < ERROR < SHOUT < FATAL
```

Application code (plugins) can obtain a LoggingService and record their separate log messages at these levels.

The user can specify the minimal log level for their debugging needs. The default level is "WARN", which discards all "DEBUG" and "INFO" logging statements. The logging level can be configured for all components or on a per-package or per-class basis.

3.2.3.1 End-user Configuration of the Logger

By default Cougaar will write all logging calls at WARN or higher to system-out, in a form like:

```
2002-03-08 22:26:34,898 WARN [class] - message
```

The output format, destination, and logging threshold can be configured at node creation time by setting "-D" system properties. Currently these properties use log4j-style statements, and the underlying LoggingService implementation uses log4j. Logging configuration can assume that log4j is in use. To see more information on log4j and configuration properties see the documentation: http://jakarta.apache.org/log4j/docs/documentation.html

The system properties are:

- "-Dorg.cougaar.core.logging.config.filename=FILE"
  Specify a log4j property filename, which is found using the ConfigFinder.

- "-Dorg.cougaar.core.logging.NAME=VALUE"
  Pass the given "NAME=VALUE" to override the above FILE contents.

For example, to enable INFO logging for the Cougaar agent-level loading sequence:

```
-Dorg.cougaar.core.loggin\
g.log4j.category.org.cougaar.core.agent.BeginLogger=INFO
```

Note that if you use any of the above "-Dorg.cougaar.core.logging.*" properties then you must fully specify the logging configuration.

Here is a minimal log4j property file that matches the Cougaar default logging configuration:

```
log4j.rootCategory=WARN,A1
log4j.appender.A1=org.apache.log4j.ConsoleAppender
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern==%d{ABSOLUTE} %-5p - %c{1} - %m%n
```

Change the "WARN,A1" to a "DEBUG,A1" and you'll get lots of debugging output from all the components. Change the "WARN,A1" to "FATAL,A1" and you'll see virtually zero logging output.

The logging level is set to a default level with the "log4j.rootCategory=.." line. The level for individual packages can changed by adding "log4j.category.PACKAGE=LEVEL" lines, such as:

```
...
# set the level to "info" for all "org.cougaar.core.*" subpackages:
log4j.category.org.cougaar.core=INFO
```

This can be used to turn the level up or down from the root level. As noted above, logging levels are "inherited" down the package hierarchy. The level can also be set for a specific class, which must be the "service-requestor" that obtained the logging service:

```
...
# debug your "com.foo.MyPlugin":
log4j.category.com.foo.MyPlugin=DEBUG
```

In some cases you may need to specify the category name:

```
log4j.category.org.cougaar.core=INFO,A1
```

The two Cougaar-specific logging levels are “DETAIL” and “SHOUT”, which must additionally specify the classname of the level implementation:

```
log4j.category.com.foo.MyPlugin=SHOUT#org.cougaar.util.log.log4j.ShoutPriority
log4j.category.blah=DETAIL#org.cougaar.util.log.log4j.DetailPriority
```

Typically one only filters at ERROR or FATAL, so a reference to SHOUT in a properties file is rarely used.

Additional documentation on log4j configuration is provided in the log4j manual.

Log4j supports many open-source log appenders and log visualizers. For example, log4j can be configured to stream logging statements over sockets. Log visualizers include the lumbermill UI.

Here is an example lumbermill ".props" file:

```
log4j.rootCategory=DEBUG,LogServer
log4j.appender.LogServer=org.apache.log4j.net.SocketAppender
log4j.appender.LogServer.Port=4445
log4j.appender.LogServer.RemoteHost=127.0.0.1
```

The Cougaar ACME automation tool uses a custom socket-based logger class, "org.cougaar.util.log.log4j.SocketAppender", to send EventService "EVENT.*" logs to the ACME controller. For details on this appender, see the ACME documentation.

3.2.3.2 Developer Use of the Logging Services

The logging support includes a "LoggingService", which is used by components to append to the log, and configuration utilities to customize the logger: set logging levels, set logging destinations, etc.

The LoggingService is the API used by component developers. This service is obtained through the ServiceBroker, just like any other service -- see the Cougaar Developer Guide section on use of the ServiceBroker for further details.

Cougaar components should **NOT** directly access the "org.cougaar.util.log" classes, such as the "LoggerFactory" or "Logger". The standard "LoggingService" should be used instead. Third-party tools or client UI's may use the LoggerFactory for their non-Cougaar logging.

For example, this is an easy way for a plugin to obtain the logger during load-time:

```
   import org.cougaar.core.service.LoggingService;
   ...
     private LoggingService log;
     public void setLoggingService(LoggingService log) {
       this.log = log;
     }
```

This is equivalent:

```
   import org.cougaar.core.service.LoggingService;
   ...
     private LoggingService log;
     public void load() {
       super.load();
       this.log = (LoggingService)
         serviceBroker.getService(
           this, LoggingService.class, null);
       ...
     }
```

Typically "this" is passed as the ServiceBroker "getService" requestor, in which case the classname of "this" is used. A component can instead pass a String to specify an alternate log4j category name.

For third party software such as client UI's that are applications that exist outside the COUGAAR infrastructure, Cougaar also has a non-service logging API, "org.cougaar.util.log"

An example of non-service logging, using a custom properties file stored on the configuration path is:

```
Private LoggerFactory lf = LoggerFactory.getInstance();
// Define some default values in case the custom properties cannot be found.
Properties defaults = new Properties();
defaults.setProperty("log4j.rootCategory", "DEBUG, A1");
defaults.setProperty("log4j.appender.A1", "org.apache.log4j.ConsoleAppender");
defaults.setProperty("log4j.appender.A1.Target", "System.out");
defaults.setProperty("log4j.appender.A1.layout",
"org.apache.log4j.PatternLayout");
defaults.setProperty("log4j.appender.A1.layout.ConversionPattern",
"%d{ABSOLUTE} %-5p [ %t] - %m%n");
Properties props = new Properties(defaults);
 // Get the debug file stored on the config path.
```

```
ConfigFinder cf = ConfigFinder.getInstance();
try {
  props.load(new FileInputStream(cf.locateFile("debug.properties")));
} catch(Exception e) {
System.err.println("Could not read debug properties file, using defaults");
lf.configure(props);
```

As previously noted, the LoggingService defines 7 logging levels:

```
   DETAIL < DEBUG < INFO < WARN < ERROR < SHOUT < FATAL
```

and three generic methods:

```
   /** @return true if the logger is enabled at the given level */
   public boolean isEnabledFor(int level);

   /** Append the given message to the log at the given level.  */
   public void log(int level, String message);

   /** Append both the given message and exception at the given level.  */
   public void log(int level, String message, Throwable t);
```

plus there are short-cut methods for all the levels:

```
   /** Equivalent to "isEnabledFor(DEBUG)" */
   public boolean isDebugEnabled();

   /** Equivalent to "log(DEBUG, message)" */
   public void debug(String message);

   /** Equivalent to "log(DEBUG, message, t)" */
   public void debug(String message, Throwable t);

   // ditto for all the other levels (isInfoEnabled(), etc)
```

Here's an example use of the log:

```
 ..
    if (log.isInfoEnabled()) {
      log.info("this is my message");
    }
```

As noted above, all logging statements specify a logging level, and if the logger is configured at a level >= to the specified level then the logging message is appended to the log.

A general guide to logging levels is:

- use DETAIL for very verbose developer “trace” debugging
- use DEBUG for code-level developer details that are just for code debugging
- use INFO for user-level "verbose" details that may be of interest but can be ignored

- use WARN for user-level warnings that are important but ignorable. This is the default level for cougaar logging.
- use ERROR for significant problems (exceptions, etc) that can't be recovered from. Log statements at ERROR or higher are rarely filtered out.
- use SHOUT for significant information messages that one would otherwise sent to standard-out, such as "Started node". This level should only be used for occasional non-ignorable information messages. Most developers should consider using INFO instead.
- only use FATAL for violations of major assumptions

Developers are **strongly** encouraged to wrap all "log(..)" calls with the related "is*Enabled(..)" check. For example, instead of just using:

```
log.debug("cons together "+this+" plus other args "+i+", etc");
```

use:

```
if (log.isDebugEnabled()) {
  log.debug("cons together "+this+" plus other args "+i+", etc");
}
```

This is most important when the log message requires the creation of a new String, but should be followed as a general principle even if a constant string is passed. Checking the logging level before creating the string, if DEBUG is not enabled, will avoid the runtime memory and performance cost, which (in practice) has been found to be a significant performance penalty.

To check for correct "is*Enabled" usage, a system property can be specified:

```
-Dorg.cougaar.util.log.checkwrappers=true
```

This adds a minor performance penalty, since all checked LoggingService calls will be checked twice, by the client and the checking code, i.e. the equivalent of:

```
if (log.isDebugEnabled()) {
  String s = "cons together "+this+" plus other args "+i+", etc";
  if (log.isDebugEnabled()) {
    log.debug(s);
  } else {
    generate_warning();
  }
}
```

The LoggingService API is essentially a subset of the log4j API, but the LoggingService API itself is not compile-time dependent upon log4j. In the future the underlying implemention may be modified to use Sun's JSR47, so LoggingService clients shouldn't make log4j assumptions when using the LoggingService.

3.2.3.3 Developer Notes on Static Logging

The LoggingService is only provided to Components (e.g. Plugins), which must have a reference to the agent's ServiceBroker. Plugins themselves are never static, so each Plugin instance can hold onto its own LoggingService instance that was provided by its ServiceBroker.

The non-static nature of the LoggingService allows the infrastructure to control logging on a per-plugin or per-agent basis. For example, if two agents are on the same node, the logger could have Binders to disable all logging for one agent and enable debug-level logging for the other agent.

A static Logger API is provided in the "org.cougaar.util.log" package. The LoggerFactory can be used to construct a new Logger instance. See the javadocs for details.

3.2.4 Blackboard Service

The objective of the BlackboardService API is to provide Plugins with the ability to specify and interact with objects (data) of specific interest to that Plugin. Plugins can specify specific Blackboard objects and LDM assets of interest (through predicates) and the types of containers in which to store these objects. The implementation of the BlackboardService API ensures transactional safety of Containers allowing multiple Plugins to concurrently interact with the Blackboard. The Blackboard was previously referred to as the LogPlan, which is really a Logistics-specific Cougaar Blackboard. The previous API was commonly known as the Subscriber and Plan API.

```
package org.cougaar.core.blackboard;

import org.cougaar.core.component.Service;
import org.cougaar.core.blackboard.Subscriber;
import org.cougaar.core.blackboard.Subscription;
import org.cougaar.core.blackboard.SubscriberException;
import org.cougaar.core.blackboard.SubscriptionWatcher;
import org.cougaar.core.persist.PersistenceNotEnabledException;
import org.cougaar.util.UnaryPredicate;
import java.util.*;

/** A BlackboardService is an API which may be supplied by a
 * ServiceProvider registered in a ServiceBroker that provides basic
 * blackboard publish, subscription and transaction services.
 **/
public interface BlackboardService extends Service {
  //
  //Subscriber/ Subscription stuff
  //
  Subscriber getSubscriber();

  /** Request a subscription to all objects for which
   * isMember.execute(object) is true.  The returned Collection
   * is a transactionally-safe set of these objects which is
   * guaranteed not to change out from under you during run()
   * execution.
   *
   * subscribe() may be called any time after
   * load() completes.
   **/
  Subscription subscribe(UnaryPredicate isMember);

  /** like subscribe(UnaryPredicate), but allows specification of
   * some other type of Collection object as the internal representation
   * of the collection.
   * Alias for getSubscriber().subscribe(UnaryPredicate, Collection);
   **/
  Subscription subscribe(UnaryPredicate isMember, Collection realCollection);

  /**
   * Alias for getSubscriber().subscribe(UnaryPredicate, boolean);
   **/
  Subscription subscribe(UnaryPredicate isMember, boolean isIncremental);

  /**
   * Alias for <code>getSubscriber().subscribe(UnaryPredicate, Collection,
boolean);</code>
   * @param isMember a predicate to execute to ascertain
   * membership in the collection of the subscription.
   * @param realCollection a container to hold the contents of the
subscription.
   * @param isIncremental should be true if an incremental subscription is
desired.
```

```
   * An incremental subscription provides access to the incremental changes to
the subscription.
   * @return the Subsciption.
   * @see org.cougaar.core.blackboard.Subscriber#subscribe
   * @see org.cougaar.core.blackboard.Subscription
   **/
  Subscription subscribe(UnaryPredicate isMember, Collection realCollection,
boolean isIncremental);

  /** Issue a query against the logplan.  Similar in function to
   * opening a new subscription, getting the results and immediately
   * closing the subscription, but can be implemented much more efficiently.
   * Note: the initial implementation actually does exactly this.
   **/
  Collection query(UnaryPredicate isMember);

  /**
   * Cancels the given Subscription which must have been returned by a
   * previous invocation of subscribe().  Alias for
   * <code> getSubscriber().unsubscribe(Subscription)</code>.
   * @param subscription the subscription to cancel
   * @see org.cougaar.core.blackboard.Subscriber#unsubscribe
   **/
  void unsubscribe(Subscription subscription);

  int getSubscriptionCount();

  int getSubscriptionSize();

  int getPublishAddedCount();

  int getPublishChangedCount();

  int getPublishRemovedCount();

  /** @return true iff collection contents have changed since the last
   * transaction.
   **/
  boolean haveCollectionsChanged();

  //
  // LogPlan changes publishing
  //

  boolean publishAdd(Object o);

  boolean publishRemove(Object o);

  boolean publishChange(Object o);

  /** mark an element of the Plan as changed.
   * Behavior is not defined if the object is not a member of the plan.
   * There is no need to call this if the object was added or removed,
   * only if the contents of the object itself has been changed.
   * The changes parameter describes a set of changes made to the
   * object beyond those tracked automatically by the object class
   * (see the object class documentation for a description of which
   * types of changes are tracked).  Any additional changes are
   * merged in <em>after</em> automatically collected reports.
   * @param changes a set of ChangeReport instances or null.
   **/
  boolean publishChange(Object o, Collection changes);

  //
  // aliases for Transaction handling
  //

  void openTransaction();

  boolean tryOpenTransaction();
  void closeTransaction() throws SubscriberException;

  void closeTransaction(boolean resetp) throws SubscriberException;


  //
  // ScheduleablePlugin API
  //

  /** called when the client (Plugin) requests that it be waked again.
   * by default, just calls wakeSubscriptionWatchers, but subclasses
   * may be more circumspect.
   **/
  void signalClientActivity();

  /** register a watcher of subscription activity **/
  SubscriptionWatcher registerInterest(SubscriptionWatcher w);

  /** register a watcher of subscription activity **/
  SubscriptionWatcher registerInterest();

  /** stop watching subscription activity **/
  void unregisterInterest(SubscriptionWatcher w) throws SubscriberException;

  //
  // persistence hooks
  //

  /** indicate that this blackboard service information should (or should not)
   * be persisted.
   **/
  void setShouldBePersisted(boolean value);
  /** @return the current value of the persistence setting **/
  boolean shouldBePersisted();


  /** is this BlackboardService the result of a rehydration of a persistence
   * snapshot?
   **/
  boolean didRehydrate();

  /**
   * Take a persistence snapshot now. If called from inside a
   * transaction (the usual case for a plugin), the transaction will
   * be closed and reopened. This means that a plugin must first
   * process all of its existing envelopes before calling
   * <code>persistNow()</code> and then process a potential new set of
   * envelopes after re-opening the transaction. Otherwise, the
   * changes will be lost.
   * @exception PersistenceNotEnabledException
   **/
  void persistNow() throws PersistenceNotEnabledException;
}
```

3.2.4.1 BlackboardService API Concepts

For the user of the BlackboardService API, most notably the Plugin developer, the most important concepts are the Subscriptions. Using the BlackboardService API, the Plugin may establish one or more subscriptions with the BlackboardService, and as a result receive objects of interest.

The BlackboardService API provides the user with a flexible interface that allows the Plugin to maintain its own collections of items. Using the BlackboardService API, the user specifies its objects of interest via a UnaryPredicate, and receives the objects of interest in an extension of a Collection, called a Subscription. For instance, in order to maintain a collection of expandable Tasks, a Plugin could use the following code:

```
// Sample ExpanderPlugin code
private IncrementalSubscription expandableTasks;

// Predicate to select the objects of interest
private static final UnaryPredicate expTaskPred = new UnaryPredicate() {
  public boolean execute(Object o) {
    if (o instanceof Task) { // is o a Task?
      Task t = (Task o);
      if ( t.getWorkflow() == null ) { // is it a "root" task?
        return true;
      }
    }
    return false;
  }
};

protected void setupSubscriptions() {
  expandableTasks = (IncrementalSubscription)
    myBlackboardService.subscribe(expTaskPred);
}

protected void execute() {
  for (Iterator it = expandableTasks.iterator(); it.hasNext(); ){
    Task t = (Task) it.next();
    if (t.getPlanElement() == null) {  // is it unexpanded?
      // expand Task t
    }
  }
}
```

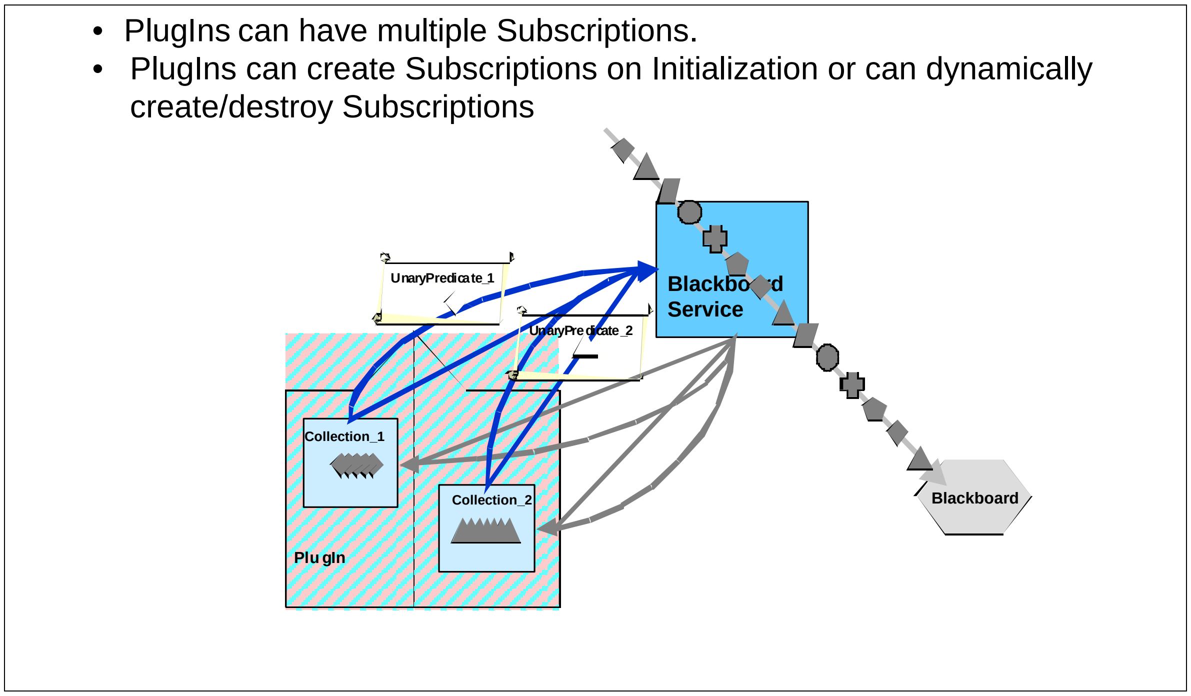
3.2.4.2 Plugin BlackboardService API Usage Model

Plugins will usually override the setupSubscriptions() method which is the first point at which Subscriptions can be safely created.

Plugins desiring to use the BlackboardService API mechanism should implement the execute() method. The relationship of "execute()" to the BlackboardService API is defined as:

- When your Subscriptions receive updates, your "awaken" - execute() is called.
- You do something in execute() - based on updated Subscriptions.
- When execute() is finished, you go back to sleep and the BlackboardService forwards any changes you have made to the Subscriptions to the rest of the Agent.
- (repeat)

If execute() never relinquishes control (e.g., non-terminating loop), the cycle is broken, and no other entity in the Agent will ever be updated with any changes the Plugin has made. From a Plugin developer's perspective, you need to decide how much "work" each execute cycle can perform relative to how frequently you want to be "notified" of updates (execute() called). "Do I stuff in code that takes 3 days to calculate ... during that time I won't be updated of new inputs?" In cases like this, where you have a lot of work or work that needs to be de-coupled from the "Container update event cycle" ... you may want to look at pushing that into a background private thread



- PlugIns can have multiple Subscriptions.
- PlugIns can create Subscriptions on Initialization or can dynamically create/destroy Subscriptions

Figure 3-6: The Plugin has two subscriptions registered with the BlackboardService.

Subscription_1 causes objects that are selected by UnaryPredicate_1 to be placed in Container_1; Subscription_2 causes objects that are selected by UnaryPredicate_2 to be placed in Container_2.

Changes that are made to the Subscriptions that are returned by subscribe will eventually be propagated to other Plugins in the agent, as well as changing the local copies of the Subscriptions.

Objects that are added to Subscriptions are transmitted to other BlackboardService clients.

The user can add objects to a Subscription. As these new objects are added to a Subscription, they become known to the Blackboard and are sent to the Subscription of any other BlackboardService client whose UnaryPredicate matches the object.

Objects that are removed from Subscriptions are transmitted to other BlackboardService client.

The user can also remove objects from a Subscription. As these new objects are removed from a Subscription, that change becomes known to the BlackboardService and is removed from the Subscription of any other BlackboardService client whose UnaryPredicate matches the object.

Objects that are explicitly changed are transmitted to other BlackboardService clients.

Any Plugin may also explicitly mark an object in a Subscription as “Changed.”

BlackboardService clients are notified when any Subscription is Changed.

When any Subscription is changed by the BlackboardService, the client/subscriber is notified. A method named execute() , is called in the subscribing Plugin's interface. Subscribing Plugins must implement the execute() method.

A Subscription can specify the actual Container to use.

In order to allow more efficient searching of Subscriptions, a second argument to subscribe can specify the Container instance to use for the real storage of the collection inside the returned Subscription. This Container can be an instance of any reasonable subclass of the JGL Container class, such as OrderedSet, HashMap, etc.

The inner Container can impose an ordering or more efficient representation of the collection so that the Plugin has better (read only) access. Examples of this would be a List instance that kept the elements sorted in some useful way, or a Set that provided a method to index the elements by some accessor value.

Container Updates

A Plugin can create a subscription at any time via the BlackboardService API:

```
      mySubscription = myBlackboardService.subscribe(myPredicate).
```

This call may block for some time. Upon return, the returned Subscription will be filled with all objects known to the agent that pass the specified predicate. Each time the execute() method is called, the Subscription (and the inner Container) will be updated with matching changes. All such matches will be propogated to mySubscription. In this way, the "ad hoc subscription" mechanism can be used to "dynamically" query for Blackboard objects of interest.

Unlike previous infrastructure versions, the returned subscription is always "filled" as much as possible upon return. This obviously does not, however, preclude the subscription from changing in the future (e.g. new Assets being given to or taken from an Agent, etc).

The BlackboardService and Transactions

Generally, the "transaction" mechanism for Plugins is defined by the execute() method. Plugins have "Container level transaction safety" during the scope of the execute() method, thus, within the execute() method, it is sufficient to simply invoke the subscribe method. In some circumstances the Plugin may need to provide "Container level transaction safety" outside of execute() cycles.

To safely invoke the subscribe method outside of the execute() method, use the following code sample, which provides container level transaction safety. Note that openTransaction and closeTransaction are methods provided in the BlackboardService API.

```
myBlackboardService.openTransaction();
mySubscription = myBlackboardService.subscribe(myPredicate);
myBlackboardService.closeTransaction();
```

No object level transaction safety is provided. These "Container level transactions" only lock the Plugin's subscriptions: other Plugins in the agent are free to run even while your Plugin has an open transaction, as they each have their own separate collections.

A Plugin may have multiple active threads of execution which do not rely on the execute() method being called. To allow other threads safe read and write access to their Subscriptions, the Plugin must explicitly call openTransaction et al.

Other Plugin base classes may offer different levels of automatic transaction handling.

3.2.4.3 Subscribing to Changing Objects

Subscription Predicates are only evaluated when objects are added, changed or removed. That is, a subscription considers adding an object to its collection by invoking the associated predicate exactly once when the object is added to the Blackboard: only Add events can result in additions to a subscription, only Remove events can result in removed objects and only Change events can result in objects marked as changed in a subscription.

This means that Predicates should, in general, only select objects based on immutable features. Violating this principle may result in extremely confusing results, especially when using the underlying contents of a subscription rather than "delta" lists. For example, if the above predicate were modified to select for unexpanded tasks, such a task might be added to the subscription and then the Plugin would run and expand the task. What is surprising is that the task would remain in the collection forever - even if "removed" from the plan! The reason is that any future Change or Remove events on that task would be deemed irrelevant to the subscription because the predicate is no longer true.

Some simple Plugins have successfully avoided problems in this situation by only watching the delta lists of the subscription. This works by, in effect, completely ignoring the long-term state of the subscription and only ever watching the events.

The recommended method for avoiding this situation is to subscribe using a predicate which selects only based on the unchanging subset of features and then filter for the currently interesting dynamic features in your execute method.

The Cougaar infrastructure allows for the definition of predicates which extend DynamicUnaryPredicate rather than UnaryPredicate. This instructs the Subscriptions to expend the additional effort needed to support selection of objects based on changing features. There is considerable overhead for this feature and it may only be used with Subscription classes that support it (including anything which is instanceof CollectionSubscription).

CollectionSubscriptions actually themselves implement the full Java Collections API. This allows treating Subscriptions as any other Collection in any context which requires one. Of particular interest to Cougaar users is the org.cougaar.core.util package which contains a number of utility classes for filtering, selecting and mapping over elements of collections (see org.cougaar.core.util.Filters and org.cougaar.core.util.Mappings).

3.2.4.4 ChangeReport

When a Blackboard object has been reported as changed, a ChangeReport facility allows detailed information to be collected and published along with the "change fact." Most Publishable plan objects have one or more implementations of ChangeReport which are collected implicitly as a result of making changes to those objects. For instance, whenever an estimated result of a PlanElement is modified, the infrastructure automatically constructs an instance of the class PlanElement.EstimatedResultChangeReport and makes it available to any Plugins which are interested in that PlanElement.

Only top-level Blackboard objects which implement Publishable can do automatic changereport construction. Exactly which changes are tracked on which objects may be found in the javadocs. This set and the level of detail collected will continue to be expanded as required.

Note that although many changes are tracked automatically, the Plugin making the changes still needs to call myBlackboardService.publishChange to post the changes!

In addition to implicitly collected changes, a Plugin may post additional ChangeReports explicitly by passing a Collection of appropriate ChangeReport instances as an optional second argument to myBlackboardService.publishChange. The implicit and explicit changes are merged at transaction close time before being sent to any other Plugins.

If no change reports are specified (implicit or explicit) then a default "Anonymous Change Report" is used. The anonymous change report is defined in "org.cougaar.core.blackboard.AnonymousChangeReport", and is a singleton instance that is reused whenever change reports are not specified. The anonymous change report allows a Plugin to know if either it or another Plugin changed the object without attaching a ChangeReport.

The anonymous change report is important when a Plugin must distinguish between its own changes and the changes of one or more other Plugins. For example, suppose one Plugin always attaches a custom ChangeReports whenever it makes a change. When it later sees that an object has changed, and it sees either an unknown or AnonymousChangeReport in the set of changes, the Plugin knows that some other Plugin has "publishChange()"'d the object.

Plugins may define their own ChangeReport classes to fit their requirements. All ChangeReport implementations must be Serializable. In addition, the equals() and hashCode() methods must be defined to operate solely on the type of change the report represents, ignoring any old or new values which might actually be stored in the ChangeReport. For instance, EstimatedResultChangeReport has both type and old value data members, but its equals method ignores the old value. This requirement allows the infrastructure to keep track of which ChangeReports are most important, while discarding redundant or uninteresting ones.

On the consumer side, a Plugin may ask any Subscription for details about any changed object. In particular, the method is:

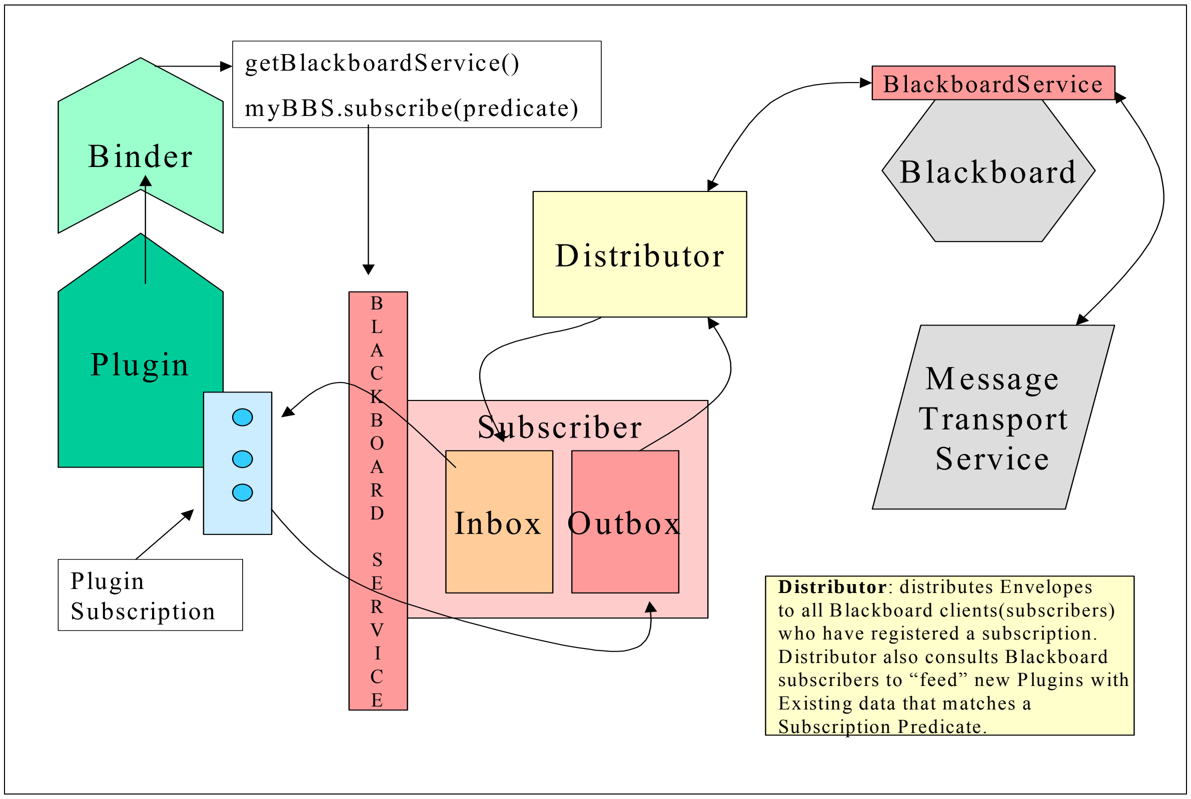
```
        Set CollectionSubscription.getChangeReports(Object);
```

The returned value may be null or a set of ChangeReport instances which represent the changes to that object that have occurred since the state of the object of the Plugin's previous open transaction.

Changes are discarded at the end of each transaction.

3.2.4.5 BlackboardService Internal Concepts

Internally, the BlackboardService implementation contains a number of objects that interact to provide its external functionality. The central character is the Distributor. The Distributor acts as a transaction update clearinghouse, distributing updates to all interested subscribers. The Distributor also keeps track of a special "Blackboard" subscriber that maintains the definitive set of all Blackboard objects. The Blackboard not only tracks all changes, but also provides the content for all new subscriptions. A Subscriber manages a set of Subscriptions (each with its own Container), interacts with the Distributor, and manages transactions for its client (usually a Plugin).



getBlackboardService()
myBBS.subscribe(predicate)
Binder
Plugin
Plugin Subscription
BLACKBOARD SERVICE
Distributor
Subscriber
Inbox
Outbox
BlackboardService
Blackboard
Message Transport Service
Distributor: distributes Envelopes to all Blackboard clients(subscribers) who have registered a subscription. Distributor also consults Blackboard subscribers to "feed" new Plugins with Existing data that matches a Subscription Predicate.

Figure 3-7: BlackboardService API Internal Concepts

3.2.4.6 BlackboardQueryService

BlackboardQueryService may be considered a minimal subset of BlackboardService for clients which need only perform transactionless and read-only queries of the Blackboard. The service offers only a single method: Collection query(UnaryPredicate) which functions identically to BlackboardService.query(UnaryPredicate). The only difference is that the BlackboardQueryService does not require that the client implement org.cougaar.core.blackboard.BlackboardClient and there is no need (or mechanism) to open a transaction.

3.2.5 Message Transport Service

3.2.5.1 Overview

The Cougaar Message Transport Service (MTS) is an adaptive Node-level service that handles all inter-Agent communication in a Cougaar society. It is componentized and includes adaptive features that can be selected at run-time. Adaptive features from the Ultra*log program include: adaptivity to different threat level of society (BBN), dependability (OBJS), security (NAI), performance (BBN QUO) and agent-mobility (BBN).

The MTS is organized in classical Object Oriented Programming (OOP) fashion around a collection of classes that define the structural components, augmented with a simple implementation of Aspect Oriented Programming (AOP) that facilitates crosscutting among the structural objects. Together, the OOP and AOP components provide a simple, robust core that's very easy to adapt and extend. Both the OOP and AOP components are described below. MTS code was refactored, out of the core, to allow ease of use and understanding for developers. A description of this refactoring follows.

3.2.5.2 MTS Design and Architecture

MTS Structural Components

The core of the MTS is implemented using classical OOP techniques. The classes are described in more detail in other sections. The intention of the overall MTS design is that the core structural components will remain as is, without addition (except for new Link Protocols) or modification (except for bug fixes). Rather than modifying the core to add new behaviors or modify existing ones, those additions and changes will be implemented as Aspects, described in more detail in the Aspects section.

The backbone of the MTS consists of a series of interfaces that define the flow of a message from the source Agent to the destination Agent, and a collection of Link Protocols that handle the various kinds of message delivery (e.g. local, remote-via-RMI, remote-via-CORBA, email). The flow interfaces are known as *stations*, since messages pass through them in a well-defined order. The stations and their implementation and factory classes, along with the Link Protocols and a few key utility classes, constitute the structural core of the MTS and are described in more detail in the Stations section

Packages and Organization

The Message Transport Service remains as service, and as such should be organized as any other service. To this end, MTS was refactored in 11.0 into its own module, with its own corresponding jar, for ease of developer understanding and use, and more easily facilitates other pluggable messaging options. The logic of the refactor focused on separation of base MTS implementation from Aspect code, used by outside developers. What was necessary to core remained, including the basic MessageTransportService interface. The rest was refactored into two packages for the standard service, plus another two for completely new LinkProtocols:

`org.cougaar.mts.base`

- Aspect-oriented implementation of minimal MTS including stations, data structures and standard Link Protocols
- Offers all the hooks for aspects to add adaptivity
- Should export NO services at the Node layer, except MessageTransportService

`org.cougaar.mts.std`

- Contains extensions, aspects local services and their required data structures
- May export Node-level service

`org.cougaar.mts.corba`

- Contains the implementation of the CORBA LinkProtocol

`org.cougaar.mts.html`

- Contains the CSI implementations of the HTTP and HTTPS LinkProtocols

3.2.5.3 MTS Station Backbone

Message Stations

The basic stations are described below (Figure 3-8) and also depicted in the "top view" diagram.

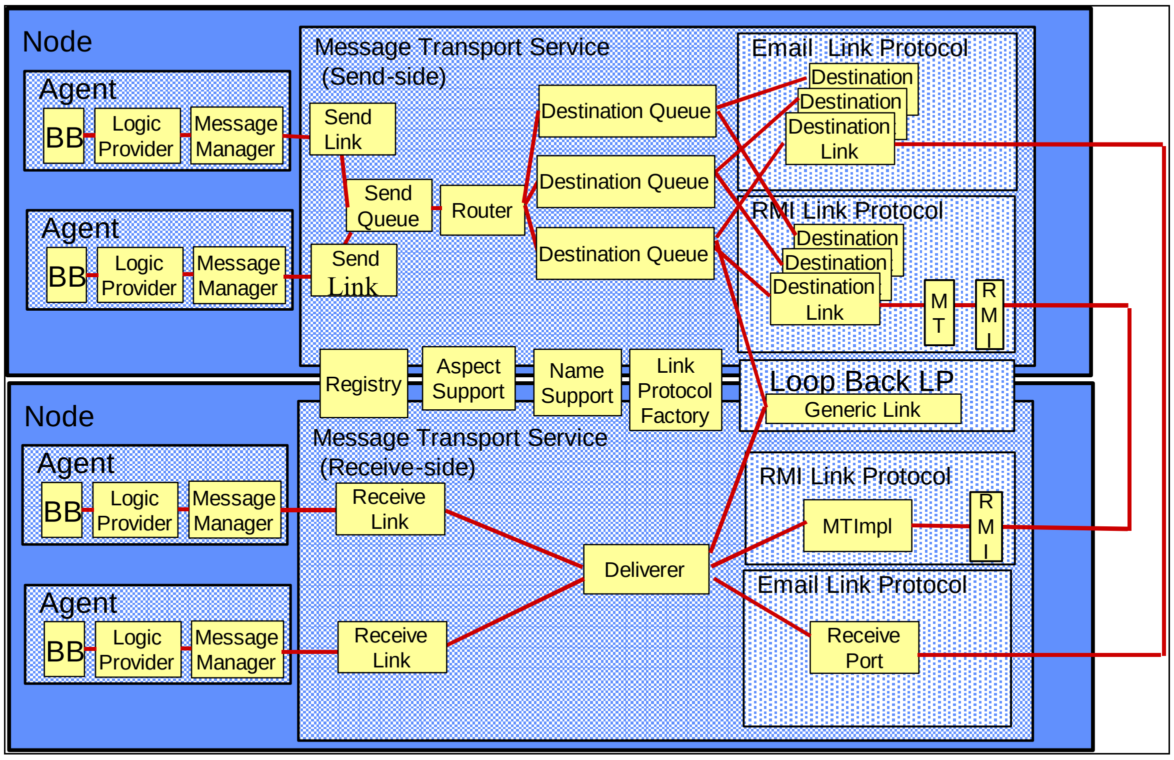


Figure 3-8: Componentization of Message Transport ("top view"). This figure diagrammatically enumerates all the Message Stations.

When a message is sent from the sending Agent to the receiving Agent, it flows through a predefined series of stations, described below in the "side view" of the Services (see Figure below). A message flows from the sending Agent through a per-Agent Send Link, associated with the Agent's Message Transport Service Proxy, into a shared Send Queue. The act of dropping the message in the Send Queue terminates the invoking call. The Send Queue's dedicated thread then passes the each message to the Router, which picks a Destination Queue for that message. Each Destination Queue has its own thread and is responsible for delivering messages to the corresponding receiving Agent, even if that Agents isn't active at the time. The Destination Queues select the "best" LinkProtocol for each message and hand it off to a DestinationLink created by the LinkProtocol. The Destination Link then moves the message to the MessageDeliverer on the destination Node, using the given protocol (eg RMI, email, nttp, loopback, CORBA, or raw sockets). The MessageDeliverer finds the corresponding ReceiveLink for the message, which in turn hands the message off to the destination Agent. If the receiving Agent is no longer local, the MessageDeliverer will throw an exception back to the DestinationQueue in the sending Node

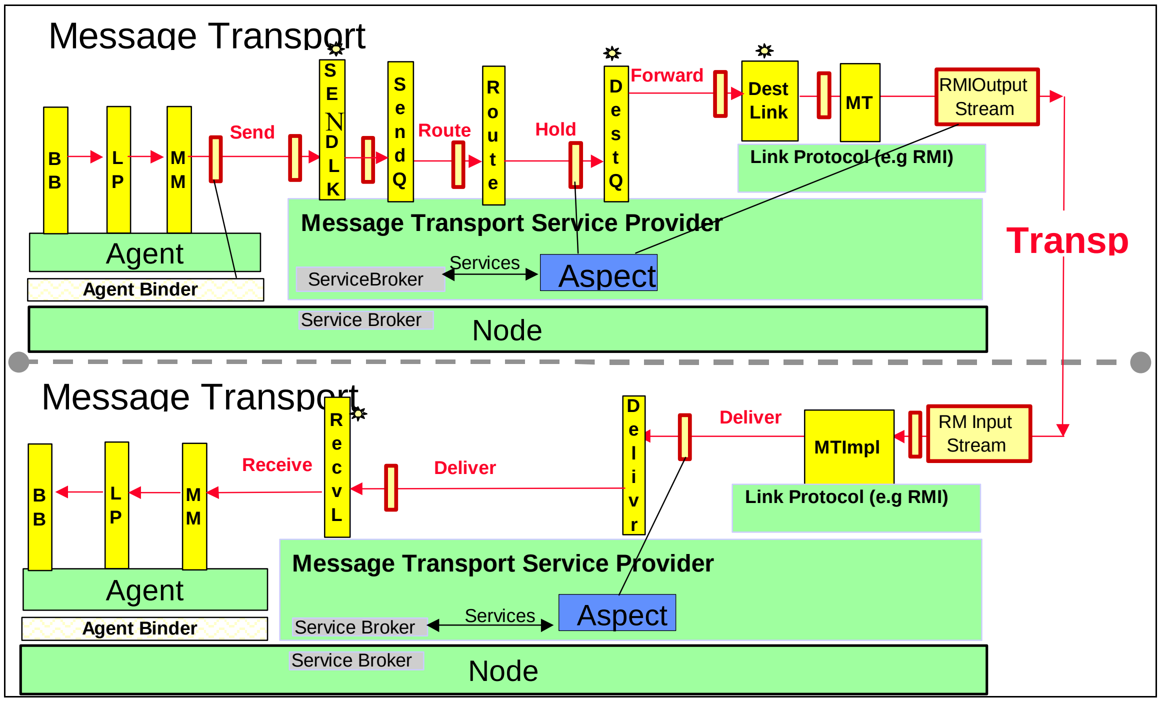


Figure 3-9: The "side view" shows the flow of messages through the stations.

MessageTransportService

This interface specifies the Service itself, as seen by the Agent. The implementation class is MessageTransportServiceProxy, exactly one of which is created for each Agent which requests the MessageTransportService. The MessageTransportServiceProvider makes MessageTransportServiceProxy instances on demand. The implementation class does very little work itself. Most of the work is handled by a SendLinkImpl, which is created for the MessageTransportServiceProxy by the MessageTransportServiceProvider.

SendLink

This is the first functional MTS station. The SendLink handles client registration, decides whether or not a given message can be sent, and in most circumstances starts the message on its way by placing it on the SendQueue. In ordinary message traffic, the calling thread of a message-send operation returns once the SendLink has placed the message on the SendQueue. SendLinkImpl, one of which is made for each MessageTransportServiceProxy by the MessageTransportServiceProvider, implements SendLink.

SendQueue

The SendQueue has two purposes: it keeps messages in order, and it prevents the calling thread of the sending Agent from blocking. As such the queue itself is simply a holding place for messages. SendQueueImpl, instances of which are created by the SendQueueFactory, implements the SendQueue interface. In the current design, SendQueueFactory is ServiceProvider for the SendQueue service. SendQueueImpl is a sublcass of MessageQueue and as such has an associated thread. In this case the thread pulls Messages off the queue and forwards them to the Router.

Router

The Router has the job of sorting Messages by target (destination) and passing them to the corresponding DestinationQueue for that target. It uses the DestinationQueueProviderService, which will find or make at most one DestinationQueue for any given target. RouterImpl, instances of which are created by the RouterFactory, implements the Router interface. In the current design, RouterFactory is a ServiceProvider for the Router service.

DestinationQueue

The DestinationQueues have the job of selecting the "best" LinkProtocol for any given message, and delivering the message, in the correct order, to a DestinationLink provided by that LinkProtocol. The selection itself is handled indirectly, via the LinkSelectionPolicy (see below). DestinationQueueImpl, instances of which are created by the DestinationQueueFactory, implements the DestinationQueue interface. One DestinationQueueImpl is made for each referenced destination MessageAddress by the DestinationQueueProviderService. DestinationQueueImpl is a subclass of MessageQueue and as such has an associated thread. In this case the thread pulls Messages off the queue and attempts to dispatch them to a DestinationLink, which can deliver them. For any given Message these attempts will continue until the Message is successfully delivered.

DestinationLink

DestinationLinks do the protocol-specific work of getting the message from the sender's Node to the receiver's Node. The DestinationLink implementation classes are specific to their corresponding LinkProtocol and are typically defined as inner classes of the protocol classes. See the LinkProtocol section for more details.

MessageDeliverer

The MessageDeliverer has the job of finding the right ReceiveLink for each Message, and passing the Message off to that link once it's been found. MessageDeliverer is an MTS-level service, provided by the MessageDelivererFactory and implemented in the current design by a singleton MessageDelivererImpl instance.

ReceiveLink

ReceiveLinks simply accept Messages from the MessageDeliverer and hands them off to the ultimate recipient. ReceiveLinkImpl, instances of which are created by the ReceiveLinkFactory, implements the ReceiveLink interface. The MessageTransportRegistryService makes requests for instantiation on demand via the ReceiveLinkProviderService.

Link Protocols

As the side-view diagram makes clear, message flow is divided into two parts, send and receive, with the linkage between the two left unspecified. There are a number of ways in which this linkage might be handled: it could be direct (for local messages), it could use an RMI or CORBA call, it could use email or news, it could use sockets, etc. The MTS implements these various kinds of linkages as instantiable subclasses of the abstract class LinkProtocol. These instantiable subclasses have full responsibility for getting a Message from the sender's Node to the receiver's Node. Some of these Link Protocols are part of the Cougaar 'core' module, some are from other modules supported by BBN, and some are from other developers. The BBN-supported Link Protocols are described here. Defining new LinkProtocol classes is one of two primary hooks through which the MTS can be extended (aspects are the other).

Link Protocols are connected to the message flow by the DestinationLink station. DestinationLinks can be thought of as destination-specific front ends to Link Protocols and are ordinarily defined as inner classes of LinkProtocol classes. The protocol itself is responsible for the real work of moving Messages, and also acts as the "factory" for its DestinationLinks.

Link Selection Policy

When more than one LinkProtocol is defined, the DestinationQueues need to pick one for each queued Message. The LinkSelectionPolicy, a service created by the MessageTransportServiceProvider, handles the job of selecting the best protocol for any given message. The default implementation class for this interface is MinCostLinkSelectionPolicy. Loading another implementation overrides this default.

The MinCostLinkSelectionPolicy does what its name implies: it selects the cheapest DestinationLink. The DestinationLinks, in turn, compute a cost by asking their enclosing LinkProtocol class.

Naming and Registration

Two forms of name registration are managed by the MTS. One, the MessageTransportRegistryService, handles local Agent registration by keeping a map from Agent addresses to the Agent's ReceiveLink representation. This simple map is used in the final step of Message delivery. The NameSupport Service represents the second naming class in the MTS. NameSupport provides a high-level front-end to the basic Cougaar naming services (eg WhitePages). NameSupport and MessageTransportRegistryService are available as Services to all MTS Components.

3.2.5.4 MTS Link Protocols

Overview

LinkProtocol instances are Components of the MTS and have access to all MTS Services, as well as Node-level Services. They are also Service Providers and may define protocol-specific services in the MTS Service Broker.

If no LinkProtocols are loaded, the LinkProtocolFactory will create a LoopbackLinkProtocol and an RMILinkProtocol.

RPC abstraction

A number of useful link protocols have an RPC-like structure, which has been captured in the abstract class RPCLinkProtocol. Many of the instantiated protocols extend this class and only need to define a few protocol-specific methods.

Protocols

`LoopbackLinkProtocol`

> This protocol handles intra-Node traffic. It's effectively a direct call from the DestinationLink to the MessageDeliverer.

`RMILinkProtocol`

> This sublass of RPCLinkProtocol provides the default implementation for handling inter-Node traffic, by using RMI calls to remote objects of type MT (implemented by MTImpl).

`SSLRMILinkProtocol`

> This subclass of RMILinkProtocol uses ssl encryption for the RMI communication by overriding RMILinkProtocol's getSocketFactory method.

`SerializedRMILinkProtocol`

> This subclass of RMILinkProtocol pre-serializes Message into byte arrays by overriding RMILinkProtocol's doForwarding method. It defines a new kind of MTImpl, which will handle the deserialization on the server side, and creates these new server objects by overriding RMILinkProtocol's make MTImpl method.

`CorbaLinkProtocol`

> This subclass of RPCLinkProtocol sends Messages via CORBA instead of RMI.

`HTTPLinkProtocol and HTTPSLinkProtocol`

> This subclass of RPCLinkProtocol sends Messages through servlets, using either http or https.

3.2.5.5 Aspects

The principles of Aspect Oriented Programming (AOP) provide a perfect mechanism for extending and adapting the basic structure of the MTS, since the nature of such extensions is generally a cut across two or more structural components. In order to facilitate this use of AOP without depending on third-party code generators, the MTS includes a simple, limited implementation directly in Java that uses aspect instances to provide delegates for the various station interfaces. This implementation centers on classes, which implement the interface MessageTransportAspect, in particular, subclasses of StandardAspect. All the aspects described below are sublcasses of StandardAspect. Aspects are Cougaar Components and are loaded in the usual way. Some Aspects are also created explicitly in code.

The application of aspects to station and other instances is handled by the AspectSupport Service, available to all MTS Components. The order of aspects is important and will always follow the Component loading order. The explicitly created aspects always come before the aspects loaded as Components.

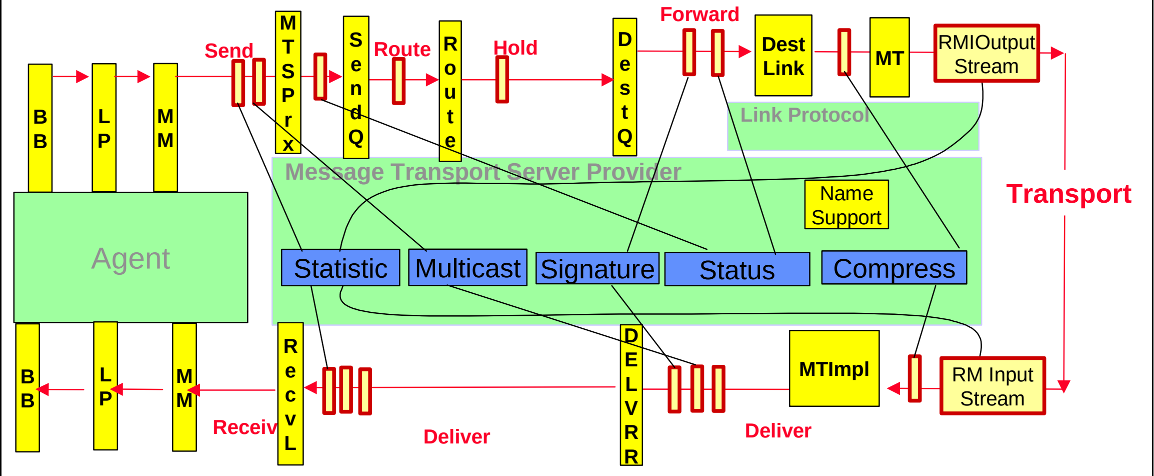
The application of aspects follows a simple pattern, using the MessageTransportAspect method getDelegate. After any "aspectizable" instance is created, whether by a factory or by any other means, each aspect in turn is given an opportunity to provide a delegate for that instance. The delegate, which will of course match the same interface as the original instance, will then take the place of the original instance or the previous delegate in the application of aspects. The final result is a nested series of objects matching the original interface whose innermost object is the original instance. The earlier an aspect appears in the global list of all aspects, the closer any delegate it provides will be to the raw instance for which the delegates are being provided.

One extra complication has arisen in the use of aspects. In some cases, proper nesting of send code with matching receive code requires that the order of aspects be reversed in the latter. In order to handle this, aspects are actually given two chances to provide delegates: once in the original aspect order, then again in the reversed order. The first opportunity is getAspect, described above. The second is getReverseAspect. In the second case, later aspect delegates will be closer the raw instance.

The following interfaces support aspects, in the sense that instances of these interfaces will always have aspect delegates attached immediately after instantiation: DestinationLink, DestinationQueue, MessageDeliverer, NameSupport, Router, SendLink, SendQueue. In addition, aspect delegates may be attached indirectly to serialization streams, which has the effect of converting a direct tcp connection into a series of Filter Streams (see org.cougaar.core.mts.MessageWriter and org.cougaar.core.mts.MessageReader).

Since most aspect delegates only want to provide a few methods of the given interface, rather than a complete reimplementation, we provide trivial delegating implementations for each of the station interfaces. In all cases, the names have the form {interface} DelegateImplBase. The aspect delegate classes should extend these and only have to provide the methods that are relevant to that aspect.

Here is a partial list of aspects in the Core MTS. The following figure depicts a few aspects and where they get attached to the message stations.



Aspect join-points are station proxies

Insertion of Station Proxies happen at Station creation time

Aspect State is held within the Aspect object

Export and import additional information from Outside message transport via Cougaar Services

Figure 3-10: Message Transport Aspects Cross Cut Multiple Stations

`AgentStatusAspect`

Collects Agent State information.

`DeliveryVerificationAspect`

Logs warnings if a message has been pending for too long.(default is 10 seconds)

`MessageProtectionAspect`

Provides an example of adding trailers to serialized AttributedMessages. Stores a message attribute reflecting the validity of the message bytecount.

`MessageSendTimeAspect`

Tags all messages with a "send time" attribute when they enter the MTS SendLink.

`MessageTimeoutAspect`

Aspect to throw out a timed out message. Necessary for MsgLog et. al. Checks every thread in MTS for timed out attributes on a message: SendLink, Router, DestinationLink, ReceiveLink

`MessageTraceAspect`

Debugging and testing Aspect - logs a successful or failed message entry per stage of the MTS. This entry contains useful information about the message as well as time deltas between stages. Use BBN Adaptive Robustness's csmart/rules/debug_rules/messageTrace.rule. Or, simply configure the society configuration to include the MessageSendTimeAspect, MessageProtectionAspect, and the MessageTraceAspect along with this argument:

```
   -Dorg.cougaar.core.logging.log4j.appender.MessageTrace.File =
#{cip}/workspace/log4jlogs/#{node.name}.trace
```

`RMISocketControlAspect`

This aspect adds simple statistics gathering to the client side OutputStream of RMI connections.

`WatcherAspect`

Implements the message watching functionality by attaching delegates to SendQueue and MessageDeliverer objects.

`MulticastAspect`

Multicast in the MTS is handled by this aspect.

`ForwardMessageLoggingAspect`

Logs interesting fields of received messages, in tab-separated format; used for debugging Link Protocols

`SecurityAspect`

Provides a simple form of message-level security by enabling message signing. This is more of an example than a functional aspect.

`SerializationAspect`

Forces the LoopbackLinkProtocol to serialize all messages; used to debug serialization error.

`StatisticsAspect`

Collects message transport statistics. This is used by the metric services PSP.

`TraceAspect`

Logs data about messages as they move through the message transport stations. This is used for debugging the message transport.

`ScrambleAspect`

Scrambles a message sequence; used to debug message transport.

`SequenceAspect`

Ensures that messages are delivered in right sequence.

`StubDumperAspect`

Displays information about LinkProtocol addresses; used to debug LinkProtocols.

`StepperAspect`

Provides a gui for "single-stepping" the sending of messages.

`ChecksumStreamsAspect`

An example of using FilterStreams in RMI. This particular example adds a checksum to the end of the message. Adding trailers onto message serialization could also be used to sign a message.

`CompressingStreamsAspect`

Another example of using FilterStreams in RMI. This example compresses and decompresses message data on the fly.

3.2.5.6 AOP is Not a Magic Bullet

By combining OOP and AOP, a simple and easily maintainable core design for message transport can be easily extended, either to provide new sorts of behavior altogether (e.g., new ,) or to adapt existing behavior in smart and dynamic ways (e.g. new Aspects), without compromising the simplicity of the core design. This is a significant win in every way. But the use of AOP in the MTS design is not a magic bullet: it won't eliminate every design problem that might ever arise, even if it greatly reduces the overall complexity.

In particular, considerable care is required when combining Aspects in previously untested ways. Aspects should usually be orthogonal to one another, and most of the current Aspects are. But even in this case, Aspect order can be very important. The user of a given set of Aspects needs to understand them well enough to choose a reasonable order at runtime.

When Aspects are not orthogonal, the interactions will of course become even trickier to handle properly (though not remotely as complicated as managing the same interactions would be in a monolithic design). In such situations, the lack of orthogonality needs to be examined very closly, since it may simply be an artifact of confused Aspect design. Non-orthogonal Aspects should be quite unusual, and as such can reasonably be expected to require special-case handling.

3.2.5.7 MTS Configuration

Loading Aspects

Aspects should be loaded as a components using the insertion point

```
Node.AgentManager.Agent.MessagTransport.Aspect
```

Loading Link Protocols

LinkProtocols should be loaded as components using the insertion point"

```
Node.AgentManager.Agent.MessagTransport.Component
```

If no LinkProtocol components are specified, RMI and Loopback will be loaded as defaults.

Single-Node MTS

A single-node MTS implementation was created for performance characterization and MTS overhead computation. It is a very lightweight version of the MTS whereby a single node is run without the overhead of costly services (naming,wp,etc.) It provides the developer with a stripped testcase for performance characterization.

Configuring a Single-Node MTS at Runtime

All that is required to run a Single-Node MTS society is to specify one machine with one or multiple agents, one node, and an argument sepcifying the desired Single-Node MTS implementation:

```
   -Dorg.cougaar.society.xsl.param.mts=singlenode
```

3.2.6 QoS Sensor Services

3.2.6.1 Overview

Previous Cougaar releases were not awareness of the underlying cyber-resources, such as CPU, storage, or bandwidth or the load that the society was putting on these resources. A society could not change its configuration or behavior based on changes in available resources or changes in mission.. Future releases of Cougaar will start introducing performance adaptive concepts, such as load balancing, agent mobility to improve performance, and variable plan fidelity. A necessary feature of performance adaptation is knowing the status and capacity of the available resources and how they map onto the society's topology. Note other necessary features not yet addressed include understanding the workload created by Agents, the specifications of QoS requirements, and the models of expected QoS, given workloads and available resources.

Cougaar 10.0 includes prototypes of services for sensing the Quality of Service (QoS) between Agents. The current capabilities include Node-level message statistics, Agent-level message statistics, society topology, and cyber-resource capacity. These services are currently available to all Node-level components and Agent-level Plugins.

In future releases, we expect these services to evolve rapidly between 10.0 and 10..4 to include additional information and will use different data formats. These capabilities will be refined and the information will be available on the Blackboard, via Senor Plugins. Finally, access to these raw services may be restricted for security reasons.

The Metric and Topology Services will integrate the raw-sensor services into a coherent real-time view of the underlying cyber-resources, agent-load, and delivered QoS. The Metric Service will be the preferred interface for the Node-Agent adaptivity engine and is described in Section 5.2.6

The raw collection of QoS data has been split into different services for Cougaar 10.0. Descriptions for each have been included in the sub-sections below.

3.2.6.2 Enabling and Testing QoS Sensor Services

An example plugin that retrieves data from various metrics services is included in the core src package in planning/examples/org/cougaar/planning/examples/MetricsComponentPlugin. This plugin can be added to any agent. The plugin will start a small one button gui; when the 'Get Metrics' button is pressed each metrics service will be asked for data from the specified API. The data retrieved will be printed to system.out. Note that the Node level metrics will apply to the node that the Agent is a member of and the Agent level services will only apply to the Agent that the plugin is loaded in. To add the plugin into an XML node configuration file, add the following line to any "<agent ...>" tag:

```
<component class="org.cougaar.core.examples.MetricsComponentPlugin"/>
```

Some of the services need to be enabled by loading a Message Transport Aspect, eg MessageStatisticsService (see services for details). To do so, add the following argument to the MYPROPERTIES variable found in

```
COUGAAR_INSTALL_PATH/bin/setarguments.bat if you are running windows or in
COUGAAR_INSTALL_PATH/bin/Node if your are running Linux.
```

3.2.6.3 NodeMetricsService

This Node level Service provides Java VM stats such as free memory, total memory and thread count. Note that *the NodeMetricService is always available*

```
package org.cougaar.core.node;
import org.cougaar.core.component.Service;

/** A NodeMetricsService is an API which may be supplied by a
 * ServiceProvider registered in a ServiceBroker that provides metrics for
 * the Node (VM).
 */
public interface NodeMetricsService extends Service {

/** Free Memory snapshot from the Java VM at the time of  the method call.**/
  long getFreeMemory();

  /** Total memory snaphsot from the Java VM at the time of the method call
**/
  long getTotalMemory();

  /** The number of active Threads in the main COUGAAR threadgroup **/
  int getActiveThreadCount();
}
```

3.2.6.4 MessageStatisticsService

Message Statistics Service is used to gather Node-level statistics on all messages being sent by the Node. The statistics include a message count, byte count, queue length, and message length histogram. The "histogram of message length" gives the same results as in Cougaar 7.2, but are really the length of internal objects being sent inside the messages. The semantics of this histogram will be redefined in future releases. For more details about these classes, see the Javadoc and source code in the core module, under the package `org.cougaar.core.mts`. *Note that the Message Statistics service is only available if the class* `org.cougaar.core.mts.StatisticsAspect` *is registered in the system property* `org.cougaar.message.transport.aspects.`

```
package org.cougaar.core.mts;
import org.cougaar.core.node.MessageStatistics;
import org.cougaar.core.component.Service;

public interface MessageStatisticsService extends Service
{
    public MessageStatistics.Statistics getMessageStatistics(boolean reset);
}
```

3.2.6.5 MessageWatcherService

Message Watcher Service is used to intercept each message that is sent or received. The service allows the addition and removal of call-back objects for monitoring messages. The call-back objects get a call back with each message sent or received. Because a reference to the raw message is given, the message content can be modified, but the message cannot be substituted or encapsulated.. For more details about these classes, see the Javadoc and source code in the core module, under the package `org.cougaar.core.mts`. *Note that the Message Watcher Service is always availiable*

```
package org.cougaar.core.mts;
import org.cougaar.core.component.Service;

public interface MessageWatcherService extends Service {

    /** Add a MessageTransportWatcher to the server. **/
    void addMessageTransportWatcher(MessageTransportWatcher watcher);

    /** Remove a MessageTransportWatcher from the server.**/
    void removeMessageTransportWatcher(MessageTransportWatcher watcher);
}
```

3.2.6.6 Agent Status Service

Agent Status Service gathers Node-level message statistics about Remote Agents. The main use for this service is to determine the observed status of a remote Agent based on message traffic to that Agent. The status includes the outcome of the last message sent to a Node and when it was sent. Also, statistics are gathered for messages sent to a specific Agent from all Agents local to this Node. For more details about these classes, see the Javadoc and source code in the core module, under the package `org.cougaar.core.mts`. *Note that the Agent Status Service is always available, i.e., the AgentStatusAspect is always at the Node-level.*

```
package org.cougaar.core.mts;

public interface AgentStatusService extends Service
{
  int UNKNOWN = 0;
  int UNREGISTERED = 1;
  int UNREACHABLE = 2;
  int ACTIVE = 3;

  class AgentState {
     public long timestamp;
     public int status;
     public int sendCount;
     public int deliveredCount;
     public int lastDeliverTime;
     public double averageDeliverTime;
     public int unregisteredNameCount;
     public int nameLookupFailureCount;
     public int commFailureCount;
     public int misdeliveredMessageCount;
  }
  AgentState getAgentState(MessageAddress address);
}
```

3.2.6.7 BlackboardMetricsService

This Agent level Service provides various counts of objects contained in the Blackboard such as Assets, PlanElements, etc *Note that the BlackboardMetricService is always available at the Agent-level.*

```
package org.cougaar.core.blackboard;
import org.cougaar.core.component.Service;
import org.cougaar.util.UnaryPredicate;

/** A BlackboardMetricsService is an API which may be supplied by a
 * ServiceProvider registered in a ServiceBroker that provides metrics for
 * the entire Blackboard.
 */
public interface BlackboardMetricsService extends Service {

  /** Get a count of objects currently in the Blackboard.
   * @param predicate Specify the objects to count in the Blackboard.
   * @return int The count of objects that match the predicate.
   **/
  int getBlackboardCount(UnaryPredicate predicate);

  /** @return int A count of the Asset's currently found in the Blackboard **/
  int getAssetCount();

  /** @return int A count of the PlanElements currently found in the
Blackbaord **/
  int getPlanElementCount();

  /** @return int A count of the Tasks currently found in the Blackbaord **/
  int getTaskCount();

  /** @return int A count of the total number of objects currently
   * found in the Blackboard **/
  int getBlackboardObjectCount();
}
```

3.2.6.8 PrototypeRegistryService

This Agent level Service provides prototype and property provider counts as well as an accessor for the number of cached Prototypes the LDM currently has registered. *Note that the PrototypeRegistryService is always available at the Agent-level.*

```
package org.cougaar.planning.ldm;
import org.cougaar.core.component.Service;

public interface PrototypeRegistryService extends Service {
   …
  // metrics service hooks
  /** @return int Count of Prototype Providers  **/
  int getPrototypeProviderCount();

  /** @return int Count of Property Providers **/
  int getPropertyProviderCount();

  /** @return int Count of Cached Prototypes **/
  int getCachedPrototypeCount();
}
```

3.2.7 WhitePagesService (Naming Service)

3.2.7.1 Overview

The white pages (WP) is the Cougaar naming service for name to address resolution, similar to DNS's job of resolving host names to IP addresses. The WP uses the in-band Cougaar message transport service and features DNS-styled caching, leasing, replication, and reconcilation.

A Cougaar agent "society" can be defined as all agents that share a common WP naming service, since this defines which agents can locate and communicate with one another. In multi-node configurations the WP configuration is crutial to ensure that the remote agents share a common naming service.

3.2.7.2 Features

Agent names should follow DNS hostname conventions (RFC 952), where the '.' character is reserved for hierarchical names. In Cougaar 10.0 the WP supports hierarchical names, and in Cougaar 12.0+ server hierarchies will be supported.

Individual agent components bind the agent in the WP when the agent is loaded. Message transport LinkProtocols use the WP to a) bind the network addresses of local agents, and b) get the addresses of remote agents. Another important WP client is the ServletService, which uses both bind and get to support HTTP redirects.

An agent's WP record contains a map of (type, scheme) pairs as keys, e.g. "(-RMI, rmi)" for a message transport RMI stub address. The values are AddressEntry data structures, which contain the (name, type, URI, cert) data, where the "cert" is an optional certificate. For example, AgentA's WP record may look like:

```
(AgentA, http, http://foo.com:8800/$AgentA, null_cert)
(AgentA, topology, node://foo.com/NodeB, null_cert)
(AgentA, -RMI,
rmi://123.45.67.89:57493/org.cougaar.mts.base.MTImpl_Stub/1_59a94c_bf48ca543b_
-8000_0/01, null_cert)
(AgentA,  version,  version:///1077744020642/1077744020642, null_cert)
```

Note that the WP entry can contain both network address details, such as an encoded RMI address and HTTP address, and also information about the agent, such as its current host/node topology and incarnation number. However, although the WP can store arbitrary URIs, it should not be used as a global database, due to scalability restrictions and the caching/leasing behavior of the underlying implementation.

The WP also supports a list operation to list the names that are currently bound. This feature is used by the "/agents" and "/tasks" servlets to display agent names. Names are listed hierarchically based upon the '.' separator character, where "." is the root of the hierarchy.

The "/wp" servlet can be used to browse the full WP contents:

```
  <component class="org.cougaar.core.util.WhitePagesServlet">
    <argument>/wp</argument>
</component>
```

3.2.7.3 Configuration

By default, the WP server is loaded onto every node and will accept lookup requests by the local agents. This simplifies single-node configurations, but in a multi-node society the WP configuration must be more explicit.

The simplest option for multi-node configurations is to specify a "-D" system property to identify the node and host:port where the WP should run. For example:

```
  -Dorg.cougaar.name.server=MyNode@foo.com:8888
```

Node "MyNode" on host "foo.com"'s WP component will see this system property and correctly act as a WP server. Other nodes will disable their WP servers and use MyNode as their WP server.

If an explicit WP server is identified, it is a good idea to disable the WP server that is (by default) loaded into every agent. On all nodes, set:

```
  -Dorg.cougaar.core.load.wp.server=true
```

and on WP nodes such as "MyNode", add the WP component:

```
  <component
    class="org.cougaar.core.wp.server.WPServer"
    insertionpoint="Node.AgentManager.Agent.WPServer"/>
```

To enable replicated WP servers, specify multiple "-Ds" for each server, with a "WP-*NUMBER*" prefix:

```
  -Dorg.cougaar.name.server=MyNode@foo.com:8888
  -Dorg.cougaar.name.server.WP-2=AnotherNode@foo.com:8888
```

Note that the WP component can be loaded in regular agents, not just nodes. Each node will "ping" the WPs to select a WP to use, and the WPs will forward bind requests to the other WP servers to replicate the data to all servers.

The WP uses a "bootstrap" discovery mechanism to local the initial WP server agent. This is analagous to the DNS problem of first contacting the DNS server – if the client only has a DNS server hostname, it is unable to ask a DNS server for its IP address!

The Cougaar WP supports several alternate WP bootstrap mechanisms, where the default is to create an RMI registry and find the WP server messaging address via RMI. Alternate protocols include fully resolved config files, "http", and "multicast". For example, to find the initial WP address via "http" and servlets, use:

```
-Dorg.cougaar.name.server=MyNode@http://foo.com:8800
```

See the javadoc in the core's "org.cougaar.core.wp.bootstrap" package for additional details and examples.

The WP bootstrap is integrated with its WP server selection design. At startup, the selector notes that it is unable to contact a WP, so it starts the bootstrap discovery process, which polls WP bootstrap addresses. As the discovery mechanisms find WP servers, they update the WP cache, which allows the selector's "ping" messages to the found WP servers to reach their targets. Once the selector has found a WP, it stops the discovery process, but will restart it if the current selection stops working or becomes very slow. See the javadoc for core's "org.cougaar.core.wp.resolver.SelectManager" for additional details.

A special case is the bootstrap for nodes containing WP servers, since they must find all their WP server peers for the replication to work correctly. See the javadoc for core's "org.cougaar.core.wp.bootstrap.EnsureIsFoundService" for details.

Many other "-D" system properties are supported to tune the WP performance; see the properties file ("api/Parameters.html") for a list of all "org.cougaar.core.wp" system properties. For example, there are system properties to reduce white pages message traffic by changing the default cache expiration time (90 seconds) and lease expiration time (4 minutes).

3.2.7.4 Developer APIs

Individual entries in the WP are specified by the AddressEntry data structure:

```
package org.cougaar.core.service.wp;
public final class AddressEntry {
  // factory method with optional certificate
  public static AddressEntry getAddressEntry(
      String name, String type, URI uri, Cert cert);
  // the name of the agent (e.g. "AgentA")
  public String getName();
  // the type (e.g. "-RMI")
  public String getType();
  // the network address (e.g. "rmi://foo.com:1234/bar")
  public URI getURI();
  // the certificate (often Cert.NULL)
  public Cert getCert();
}
```

The WhitePagesService defines the following operations:

- getAll(name) to get a Map of all AddressEntries for a given name
- get(name, type), to get a single AddressEntry, which is equivalent to getAll(name).get(type)
- list(suffix) to get a Set of agent names with a given suffix, e.g. list(".")
- bind(entry) and rebind(entry) to add an entry to the WP
- unbind(entry) to remove an entry from the WP
- flush(name, ..) to signal an early cache flush on a possibly stale entry, e.g. an RMI address that no longer works
- hint(entry) to add an entry to the local WP cache, which is used to bootstrap the node (e.g. all nodes must be bootstrapped with the WP agent's network address)
- unhint(entry) to remove a local WP cache hint
- submit(request), which is the generic form of all the above commands. For example, there's a "Request.GetAll" object that specifies a name and mirrors the getAll(name) method.

There are three variants for each of the above commands, where the deprecated variants will be removed in Cougaar 12.0+:

- A "callback" variant that passes a Callback class as the last method parameter and returns void. This is used for non-blocking lookups and is the preferred API. The Callback defines a void execute(Response res) method, where Response classes mirror Request classes (e.g. there's a "Response.GetAll" that contains a Map) and has a pointer to the matching Request object.

- A "timeout" variant, where a long is passed as the last method parameter. If "-1" is passed then this indicates a non-blocking cache-only request (e.g. a cache-only "list" request that returns the Set of names). Timeout values >=0 are deprecated, where "0" indicates block-forever and >0 indicates a time-limited block in milliseconds.

- A variant with no special last parameter, which is a deprecated block-forever operation. As noted above, this variant will be removed in Cougaar 11.2, so developers should avoid it.

The non-blocking variants are more difficult to use, but are required for efficent use of the WP and pooled threads.

Here's an example of a cache-only lookup. If the information is not in the cache then the "getAll" request will return null and the lookup will proceed in the background, so it may be available if the client asks again later:

```
public class Test extends ComponentPlugin
{
  private LoggingService log;
  private WhitePagesService wp;

  public void setLoggingService(LoggingService log) {
    this.log = log;
  }
  public void setWhitePagesService(WhitePagesService wp) {
    this.wp = wp;
  }

  protected void setupSubscriptions() {
    // do a lookup
    try {
      Map m = wp.getAll("AgentA", -1);
      log.shout(
         "getAll(AgentA): "+
         (m == null ? "not cached" : m.toString()));
    } catch (Exception e) {
      log.error("Failed", e);
    }
  }
}
```

Here's an example of a non-blocking lookup with a callback. Note that the callback will be invoked by the WP's thread, not the plugin's usual "execute" thread, so to avoid blocking WP progress the plugin must queue the callback result and request a plugin cycle. This also simplifies the plugin logic, since it allows the plugin to handle both subscription changes and WP responses in the same thread:

```
public class Test extends ComponentPlugin
{
  private LoggingService log;
  private WhitePagesService wp;

  private final Object lock = new Object();
  private Response queuedRes;

  public void setLoggingService(LoggingService log) {
    this.log = log;
  }
  public void setWhitePagesService(WhitePagesService wp) {
    this.wp = wp;
  }

  protected void setupSubscriptions() {
    submitLookup();
  }

  protected void execute() {
    handleNow();
  }

  private void submitLookup() {
    // create callback
    Callback cb = new Callback() {
      public void execute(Response res) {
        handleLater(res);
      }
    };
    wp.getAll("AgentA", cb);
    // returns immediately!  Note that the callback may be invoked
    // in the same thread if the result is already in the cache.
  }

  private void handleLater(Response res) {
    enqueue(res);
  }

  private void handleNow() {
    Response res = dequeue();
    if (res == null) {
      // no callback yet (e.g. a subscription change caused our
      // plugin to run)
      return;
    }
    // okay to process now, since we're in our usual "execute" thread
    if (res.isSuccess()) {
      // we know the type, since this example only submits a
      // "getAll", but we could do an instanceof check or check
      // the "res.getRequest()" type.
      Map m = ((Response.GetAll) res).getAddressEntries();
      log.shout("got: "+m);
    } else {
      // the lookup failed (e.g. security exception)
      Exception e = res.getException();
      log.error("failed: ", e);
    }
  }

  private void enqueue(Response res) {
    // add response to queue
    //
    // in this case a single-element queue, but we could keep
    // a list of queued responses
    synchronized (lock) {
      queuedRes = res;
    }
    // request an "execute()" call
    blackboard.signalClientActivity();
  }

  private Response dequeue() {
    // take response from queue
    synchronized (lock) {
      if (queuedRes == null) {
        return null;
      }
      Response res = queuedRes;
      queuedRes = null;
      return res;
    }
  }
}
```

Components that don't extend ComponentPlugin can use the ThreadService to guarantee that callbacks are handled in a separate thread. In above example we would replace "blackboard.signalClientActivity()" with "thread.start()" and register with the thread service:

```
  private Schedulable thread;
  public void setThreadService(ThreadService ts) {
    if (ts != null) {
      Runnable r = new Runnable() {
        public void run() {
          handleNow();
        }
      };
      thread = ts.getThread(this, r);
    }
  }
```

Components that must block for the WP result can use standard wait/notify patterns to block until the callback fires. This is not encouraged, since it ties up pooled ThreadService threads, but is required by some plugins and servlets:

```
  protected void setupSubscriptions() {
    WPCallback cb = new WPCallback();
    wp.getAll("AgentA", cb);
    Response res;
    try {
      res = cb.getResponse();
    } catch (InterruptedException ie) {
      throw new RuntimeException("Lookup failed", ie);
    }
    if (res.isSuccess()) {
      Map m = ((Response.GetAll) res).getAddressEntries();
      log.shout("got: "+m);
    } else {
      Exception e = res.getException();
      log.error("failed: ", e);
    }
  }

  private static final class WPCallback implements Callback {
    private final Object lock = new Object();
    private Response res;
    public Response getResponse() throws InterruptedException {
      synchronized (lock) {
        while (res == null) {
          lock.wait();
        }
        return res;
      }
    }
    public void execute(Response res) {
      if (res == null) {
        throw new RuntimeException("null res?");
      }
      synchronized (lock) {
        this.res = res;
        lock.notifyAll();
      }
    }
  }
}
```

If advertised, the WhitePagesProtectionService is used to wrap modifiation (bind/unbind) requests and unwrap them at the server. By default there is no implementation, but the white pages client and server will use the service if it's advertised by a HIGH-level node-agent component. For example, a WhitePagesProtectionService implementation could sign all client requests and the server unwrapping could check the signature and check a "denied-agents" list before allowing the modification.

3.2.8 YellowPagesService (Directory Service)

The YP Service (Yellow Pages Service) 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. The YPService is designed to be used with distributed, hierarchical YP Communities. To reduce bottlenecks, separate YPServers will be located in the different Communities, and the YPService will support searches that start in an Agent's own Community and incrementally broaden to larger Communities if necessary.

The primary client of the YPService is currently the Cougaar service discovery mechanism, which supports dynamic searching for Provider Agents. Understanding the details of the YPService is only required for those developers who find that the service discovery RegistrationService and RegistryQueryService do not meet their needs.

The primary interface to the Cougaar YellowPages is UDDI4J, a Java API for UDDI transactions. UDDI4J documentation and examples can be found on their web site (http://www.uddi4j.org).

UDDI was selected to support future interaction with external systems. UDDI as a primary interface may eventually be replaced by a more implementation-neutral web services registry implementation such as JAXR.

The YPService itself is very minimal; it primarily allows a component to obtain a YPProxy with various search settings. It is also used to determine the next broader Community for searching.

```
package org.cougaar.yp;

import org.cougaar.core.component.Service;
import org.cougaar.core.mts.MessageAddress;
import org.cougaar.core.service.community.Community;

public interface YPService extends Service {

  YPProxy getYP(String ypAgent); //deprecated
  YPProxy getYP(MessageAddress ypAgent);

  YPProxy getYP(Community community);
  YPProxy getYP(Community community, int searchMode);
  YPProxy getYP();
  YPProxy getYP(int searchMode);

  YPProxy getAutoYP(String context);
  YPProxy getAutoYP(MessageAddress ypAgent);
  YPProxy getAutoYP(Community community);

  YPFuture submit(YPFuture ypr);

  public void getYPServerContext(final String AgentName,
                          final NextContextCallback callback);
  public void nextYPServerContext(final Object currentContext,
                                  final NextContextCallback callback);

}
```

The getYP methods which only specify the ypAgent (the Agent running the YPServer) will return a YPProxy for only the specified YPServer. The other getYP methods will use a combination of the specified and default Community and searchMode, returning a YPProxy that may support incremental broadening of the search. Typically, a method would be invoked on the returned YPProxy, and the resulting YPFuture would be published to the blackboard and monitored for changes. The YPService monitors new YPFutures on the blackboard, submits them within a separate thread, and then publish changes the YPFuture when the submit call completes. Alternatively, clients can set the callback field in the returned YPFuture and then call the YPService submit statement. The submit call blocks until the interaction with the ypAgent is complete. YPService clients should consider calling submit within a separate thread so that the client is not blocked waiting for a remote Agent. This is the method currently used by Cougaar service discovery

The getAutoYP methods are analogous to getYP, but are intended for use when the calling entity has no blackboard access. When a YPProxy returned by getAutoYP is used, all methods executed with that YPProxy invoke YPService.submit directly rather than returning with the YPFuture. As the YPService uses Cougaar messaging to communicate with the ypAgent, this means that the thread in which the client executes the YPProxy call blocks on an interaction with a remote Agent.

getYPServerContext and nextYPServerContext offer the ability to find the appropriate YP Community. The YP Community is always returned via the specified NextContextCallback. getYPServerContext returns the YP Community for the specified Agent. nextYPServerContext returns the next YP Community up from the specified currentContext in the YP Community hierarchy.

The YellowPages is implemented entirely as a Cougaar application which implements a distributed network of UDDI servers reachable through standard Cougaar messaging. All Cougaar survivability features (e.g. high levels of security and robustness) apply to the YellowPages network within a Society. There is also an included YPServer implementation which acts as a gateway between in-band Cougaar Messaging UDDI requests and external UDDI servers (via SOAP+HTTP/S) which may be used to transparently access external web services systems.

3.2.8.1 YPProxy

The YPProxy interface extends the UDDI4J UDDIProxy API to support asynchronous operations. The YPProxy is a client proxy for the UDDI server. Its methods map directly to the UDDI Programmer's API Specification with the following differences. First, if the YPProxy was generated using some variant of YPService.getYPProxy(), methods executed on the YPProxy object will return a YPFuture object rather than the UDDI response. When the ypAgent has returned the results of the call, the results field of the YPFuture will be updated and the YPFuture callback, if specified, is executed. The following illustrates the difference between UDDIProxy.find_business and YPProxy.find_business.

UDDIProxy -

```
public BusinessList find_business(java.util.Vector names,
                                  DiscoveryURLS discoveryURLs,
                                  IdentifierBag identifierBag,
                                  CategoryBag categoryBag,
                                  TmodelBag tModelBag,
                                  FindQualifiers findQualifiers,
                                  int maxRows)
```

YPProxy -

```
public YPFuture find_business(java.util.Vector names,
                                  DiscoveryURLS discoveryURLs,
                                  IdentifierBag identifierBag,
                                  CategoryBag categoryBag,
                                  TmodelBag tModelBag,
                                  FindQualifiers findQualifiers,
                                  int maxRows)
```

The YPProxy call returns immediately with a YPFuture which will be filled in later with the ypAgent's response. The UDDIProxy returns the ypAgent's response, blocking until the interaction is complete. This is analogous to using an YPProxy returned by YPServer.getAutoYP.

In addition, the SearchMode field of YPProxy determines both the algorithm used to find the YPServers and the scope of the search used to complete the YPProxy call.

```
package org.cougaar.yp;

   …
public interface YPProxy {

public class SearchMode {

    public static final int NO_COMMUNITY_SEARCH = 0;
    public static final int HIERARCHICAL_COMMUNITY_SEARCH = 1;
    public static final int SINGLE_COMMUNITY_SEARCH = 2;

    static public boolean validSearchMode(int searchMode);

    static public boolean validCommunitySearchMode(int searchMode);
  }

  public int getSearchMode();
```

3.2.8.2 YPFuture

An outstanding YP response object, returned from all of the YPProxy methods. A consumer of the YP information would issue a query, then either publish the returned YPFuture to the blackboard or call YPService.submit. If the YPFuture is published to the blackboard, the YPService will publish change the YPFuture when the requested operation is complete. If YPService.submit was used, the YPService will execute the specified YPComplete callback.

```
 public interface YPFuture {

  boolean isReady();
  Object get() throws UDDIException;
  Object get(long msecs) throws UDDIException;
  void setCallback(YPComplete callable);
  Element getElement();
  boolean isInquiry();

  Object getInitialContext();
  Object getFinalContext();
  int getSearchMode();

  /** The interface which must be implemented by the argument to
#setCallback(Callback) **/
  interface YPComplete {
  }
  interface Callback extends YPComplete {
    void ready(YPFuture response);
  }

  /** Higher-level callback abstraction.  Similar to Callback but passes in
the results rather that
   * just notification of interaction completeness events.
   * @note The interface assumes that all important information is closed over
by the instance.
   **/
  interface ResponseCallback extends YPComplete {
    void invoke(Object result);
    void handle(Exception e);
  }
```

The get methods return the UDDI response. The msecs parameter specifies how long the get call should block if the response is not ready. If the response is ready, get always returns immediately.

The getInitialContext, getFinalContext, and getSearchMode methods describe the first and last YPServers searched and the algorithm used to find them. The SearchMode is determined by the YPProxy used in generating the call.

The setCallback method allows the caller to specify an YPComplete callback. The callback's ready method will be executed when the requested operation is complete. In addition, if the callback implements ResponseCallback, the invoke or handle methods will be called with the results of the operation: invoke with the UDDI response object, handle if the operation generates an Exception.

3.2.9 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 with which to form a relationship. The service discovery mechanism uses two services, the RegistrationService and the RegistryQueryService. Provider Agents use the RegistrationService to add their services to the YellowPages. Customer Agents use the RegistryQueryService to find Provider Agents for their required services. In both cases, YP Communities can be used to find the appropriate YPServers. Providers know which YP Communities they support and register with the YPServers for those Communities. Customers know to which YP Communities they belong and query the YPServers for those Commuties.

Both the Registration and RegistryQueryService offer only an asynchronous interface. All service calls require a callback method, which will be invoked once the operation is complete.

3.2.9.1 RegistrationService

The RegistrationService is used to create, modify, or delete information in the YellowPages about Provider Agents and the services (such as providing supplies or transportation) they can provide to Customer Agents. In general, the services provided by Agents will correspond to Cougaar Roles and will be associated with the ability to handle certain types of Cougaar Tasks.

```
package org.cougaar.servicediscovery.service;

import java.util.Collection;

import org.cougaar.core.component.Service;
import org.cougaar.servicediscovery.description.ProviderDescription;

public interface RegistrationService extends Service, YPServiceAdapter {

  void addProviderDescription(Object ypContext, ProviderDescription pd,
Callback callback);
  void addProviderDescription(Object ypContext, ProviderDescription pd,
Collection additionalServiceClassifications, Callback callback);

  void updateServiceDescription(Object ypContext, String providerName,
Collection serviceCategories, Callback callback);

  void deleteServiceDescription(Object ypContext, String providerName,
Collection serviceCategories, Callback callback);
}
```

The ypContext parameter in each call identifies the target YellowPages using either the name of the ypAgent or the YP Community name.

3.2.9.2 RegistryQueryService

The RegistryQueryService is used to search YPServers for Providers of a particular type of service. The findProviders methods result in collections of ProviderInfo objects, while the findServices methods produce ServiceInfo objects. The findServiceAndBinding methods produce lightweight ServiceInfo objects.

```
package org.cougaar.servicediscovery.service;
import org.cougaar.core.component.Service;
import org.cougaar.servicediscovery.transaction.RegistryQuery;

public interface RegistryQueryService extends Service, YPServiceAdapter {

  public void findProviders(RegistryQuery query, Callback callback);
  public void findProviders(Object lastYPContext, RegistryQuery query,
                    CallbackWithContext callback);

  public void findServices(RegistryQuery query, Callback callback);
  public void findServices(Object lastYPContext, RegistryQuery query,
                   CallbackWithContext callback);

  public void findServiceAndBinding(RegistryQuery query, Callback callback);
  public void findServiceAndBinding(Object lastYPContext, RegistryQuery query,
                            CallbackWithContext callback);
}
```

3.2.9.3 Service Discovery for Establishing Agent Relationships

Cougaar 11.0 supports the dynamic establishment of Customer/Provider relationships between Agents through service discovery. This mechanism may be used to match Agents based on their needs and capabilities, instead of specifying each Customer/Provider relationships by the Agent names. In order to be discovered, Providers must have registered information about themselves and their capabilities. The discovery process is initiated through a Customer query and can result in new relationships.

Provider Registration

Providers register information such as their name, Role capabilities, and supported organizations with the Yellow Pages. Although initial registration and later updates need not take place at any particular time, for optimal matching, the bulk of Customer queries should not preceed the bulk of Provider registrations.

The service discovery module provides a sample plugin, SimpleSDRegistrationPlugin that demonstrates the use of the RegistrationService for initial registration and updates. In order to use a common Plugin for all registering Provider Agents, specific capability information is relegated to separate data files. In particular, the SimpleSDRegistrationPlugin reads information from each Agent's Cougaar-extended OWL-S service profile files for initial registration. The extended OWL-S format provides flexibility for describing Agent capabilities. It is most useful when the characteristics registered by Provider Agents correspond to the characteristics that are the basis for Customer Agent queries. In this sample implementation, the Cougaar Role(s) fulfilled by the Provider is one important characteristic.

Updates to this registration information may be triggered during Cougaar society execution in reaction to user initiated perturbations or actions of other Agents. In this sample implementation, the initial Provider registration occurs during society start-up.

Customer Registry Query

Customer Agents can use the service discovery mechanism to specify search criteria and find out about previously registered Provider Agents matching those criteria. The Customer Agents can then select one or more Provider Agents to establish a relationship with. During society execution, the Customer Agent may discard existing relationships or form new ones.

In service discovery, most of the logic and "work" is on the Customer side. Once the Provider Agent has registered, it only needs to respond to any relationship requests from Customer Agents. On the other hand, the Customer Agent needs to know what sort of Provider it needs, search for appropriate Provider Agents, select one or more, and then request a relationship. The service discovery module provides a sample of how the Customer Agent can do this, splitting the responsibility between two plugins, the SimpleSDClientPlugin and the SimpleMatchmakerPlugin. The SimpleSDClientPlugin provides the query criteria input to the SimpleMatchmakerPlugin, based on the Customer Agent's needs. The SimpleMatchmakerPlugin demonstrates the use of the RegistryQueryService to search the Yellow Pages for Provider Agents which meet the Customer Agent's query criteria.

The SimpleSDClientPlugin also uses the query results from the SimpleMatchmakerPlugin to initiate a service relationship. Once a set of possible Provider Agents has been returned, the SimpleSDClientPlugin publishes a ServiceContractRelay directed to one of the Provider Agents which specifies the desired type and duration of service. The AggreableProviderPlugin on the Provider Agent demonstrates how the Provider Agent can respond through the ServiceContractRelay to confirm or deny its service availability. Automatic LP processing creates the Relationship entries and modifies the Customer and Provider OrganizationAssets.

In Cougaar 11.0, Yellow Pages queries and replies are simple and static. The information returned is valid at the time, but no effort is made to update the query results if Provider information changes during society execution. This means that a Customer Agent must repeat its queries anytime it wants to be assured that the results are up-to-date. For example, if a Customer whose initial search returned several possible Providers later finds that the selected Provider failed to satisfy some Tasks, the Customer should perform a second search for possible Providers. The Customer Agent can not assume that pre-existing query results are still valid.

3.2.10 Cougaar Communities

Communities provide a way of segmenting societies into distinct groups of agents organized for specific purposes. A community is basically a collection of agents and/or other communities. Agents and communities are permitted to be members of multiple communities. A common use of communities is to define a group of agents that will be used as a destination address for a Blackboard Relay. In addition to providing a way to define simple groups of agents, communities and their associated entities may also be associated with attributes that can be used to select a specific community member or subgroup. For instance, an Attribute Based Address (ABA) could be defined to send a Relay to all members in a Community using the "Role=Member" criteria. Alternately, the ABA could be addressed to those members having the "Role=Role-A" attribute in which case only those agents that had the “ROLE-A” attributed defined would receive the Relay.

Communities are defined using a couple simple constructs.

- At the top-level a Community consists of a unique name, zero or more attributes, and zero or more child entities.

- Each member (entity) has a unique name and is associated with zero or more attributes. Child entities are completely described using their associated attributes. The attribute "EntityType" has a special meaning when constructing communities as it will define whether the entity is an agent or a nested community. The value for this attribute must either be "Agent" or "Community". If the attribute is not specified the entity type "Agent" is assumed.

- Community and entity attributes are defined using the javax.naming.directory class which defines an attribute set consisting of zero or more javax.naming.directory.Attribute instances. An Attribute consists of an identifier and zero or more values. While JNDI permits the use of arbitrary objects for attribute values, community and entity attribute values must be a String.

3.2.10.1 CommunityService API

Clients interact with communities using the Cougaar CommunityService API. A CommunityService provider implementing this API is automatically loaded by the Cougaar infrastructure when an agent is created. A reference to the CommunityService provider is obtained by clients using the agent ServiceBroker. The CommunityService API provides methods enabling a client to:

1. Join and optionally create a community

2. Leave a community

3. Modify community or member attributes

4. Add/remove a listener component to receive change notifications

5. Obtain a list of communities

6. Search for communities and community members based on attributes

For a complete list of methods and a description of their usage refer to the javadoc for org.cougaar.core.service.community.CommunityService.

3.2.10.2 Forming Communities

Communities are created using the CommunityService *join* method. When invoking this method the client can specify whether the target community should be created if it doesn't already exist. If this option is enabled the calling agent will create the community and assume the community manager role if the target community cannot be located in the White Pages.

Automatic Creation of Communities During Startup

During society startup communities may be automatically created using a community definition file and the CommunityPlugin. The CommunityPlugin will look for a community definition file (named "communities.xml") on the config path during startup and will attempt to join all communities containing it as a member. This mechanism provides a flexible and convenient way to create static communities for a society. An example communities.xml file is shown below. In this example Agent1 is a member of both CommunityA and CommunityB. If the CommunityPlugin was loaded in AgentA it would automatically attempt to join both of these communities during startup.

```
  <?xml version="1.0" encoding="UTF-8"?>
 <Communities>
   <Community Name="CommunityA">
     <Entity Name="Agent1">
       <Attribute ID="Role" Value="Member" />
     </Entity>
     <Entity Name="Agent2">
       <Attribute ID="Role" Value="Member" />
     </Entity>
   </Community>
   <Community Name="CommunityB">
     <Entity Name="Agent1">
       <Attribute ID="Role" Value="Member" />
     </Entity>
     <Entity Name="Agent3">
       <Attribute ID="Role" Value="Member" />
     </Entity>
   </Community>
 </Communities>
```

By default any agent defined in the above configuration would attempt to create its parent community if the community was not found. This auto create capability can be deactivated or fine-tuned special attributes. The first attribute, "CommunityManager", is a community level attribute that can be used to explicitly define the community's manager agent. For instance,

```
<?xml version="1.0" encoding="UTF-8"?>
 <Communities>
   <Community Name="CommunityA">
     <Attribute ID="CommunityManager" Value="Agent2" />
     <Entity Name="Agent1">
       <Attribute ID="Role" Value="Member" />
     </Entity>
     <Entity Name="Agent2">
       <Attribute ID="Role" Value="Member" />
     </Entity>
         .
         .
         .
   </Community>
 </Communities>
```

With this configuration only Agent2 is allowed to create/manager CommunityA. If Agent1 starts before Agent2, its join request will not be processed until Agent2 has started and created CommunityA.

If the community-level "CommunityManager" attribute is not defined any of the member agents can potentially create/manage the community. An entity-level attribute, "canBeManager", can be used to inhibit one or more members from performing this function. For instance,

```
<?xml version="1.0" encoding="UTF-8"?>
 <Communities>
   <Community Name="CommunityA">
     <Entity Name="Agent1">
       <Attribute ID="Role" Value="Member" />
       <Attribute ID="canBeManager" Value="False" />
     </Entity>
     <Entity Name="Agent2">
       <Attribute ID="Role" Value="Member" />
     </Entity>
     <Entity Name="Agent3">
       <Attribute ID="Role" Value="Member" />
     </Entity>
        .
        .
        .
   </Community>
 </Communities>
```

In this configuration the community-level attribute "CommunityManager" is not defined. This would normally mean that any of the member agents could create/manage the community. However, in this case Agent1 has the attribute "canBeManager" set to "False" which eliminates it as a manager candidate.

Dynamic Creation of Ad-hoc Communities

In addition to the initial communities defined in a startup configuration, communities may also be created at run time. The Community Service API provides join and leave operations supporting the capability to dynamically create and populate communities. For example, suppose that Agent1 wanted to join a new community (i.e., CommunityA), it would call the CommunityService.join method to join the community. If the community did not exist and the "createIfNotFound" parameter on the join request was set to true the community would be created and Agent1 would be added as a member. If the community already had been created by another agent, the Join request would be forwarded to the associated community manager and Agent1 would be added to the community.

Community Search Filter Syntax

Information in this section was adapted from Sun's JNDI Tutorial (http://java.sun.com/products/jndi/tutorial/basics/directory/filter.html).

The CommunityService API provides a search capability that enables clients to identify entities (communities or agents) based on attribute values. Search criteria for selecting entities is defined using Java Naming and Directory Interface (JNDI) compliant filters.

The search filter syntax is basically a logical expression in prefix notation (that is, the logical operator appears before its arguments). The following table lists the symbols used for creating filters.

Symbol	***Description***
&	conjunction (i.e., and -- all in list must be true)
|	disjunction (i.e., or -- one or more alternatives must be true)
!	negation (i.e., not -- the item being negated must not be true
=	equality (according to the matching rule of the attribute)
~=	approximate equality (according to the matching rule of the attribute)
>=	greater than (according to the matching rule of the attribute)
<=	less than (according to the matching rule of the attribute)
=*	presence (i.e., the entry must have the attribute but its value is irrelevant)
*	wildcard (indicates zero or more characters can occur in that position); used when specifying attribute values to match
\	escape (for escaping '*', '(', or ')' when they occur inside an attribute value)

Each item in the filter is composed using an attribute identifier and an attribute value or symbols denoting the attribute value. For example, the item "`Role=Member`" means that the "`Role`" attribute must have the attribute value "`Member`" and the item "`FuelProvider=*`" indicates that the "`FuelProvider`" attribute must be present.

Each item must be enclosed within a set of parentheses, as in "`(Role=Member)`". These items are composed using logical operators such as "&" (conjunction) to create logical expressions, as in "`(&(Role=Member)(FuelProvider=*))`".

Each logical expression can be further composed of other items that themselves are logical expressions, as in "`(|(&(Role=Member)(FuelProvider=*))(Unit=L*))`". This last example is requesting either entries that have both a "`Role`" attribute of "`Member`" and the "`FuelProvider`" attribute or entries whose "`Unit`" attribute begins with the letter "`L`."

For a complete description of the syntax, see RFC 2254.

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.

To use, construct a new `Relay` object with an `AttributeBasedAddress` as its target. To use the Relay object interface, you must implement it with a class first. The implemented class in the example below is TestRelay, and resides in the Aggagent module in aggagent/src/org/cougaar/lib/aggagent/test/.

```
// Create recipient addresses
MessageAddress source = getBindingSite().getAgentIdentifier();
MessageAddress target = new AttributeBasedAddress("", "Role", "Manager");

// Create Relay object
TestRelay tr = new TestRelay( getUIDService().nextUID(), source, target,
"Foo", null);

// publish Relay object
getBlackboardService().openTransaction();
getBlackboardService().publishAdd(tr);
getBlackboardService().closeTransaction();
```

`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).

3.2.11 Agent Mobility Support

3.2.11.1 Introduction

Agent mobility is the infrastructure mechanism to dynamically transfer and agent from one node to another node, possibly to a different machine.

This mechanism by itself does not decide when or where to move agents. A Plugin or Component within an agent must decide when to issue the move, which agent should be moved (potentially itself) and which node it should move to.

There are many potential reasons that a developer may want to move an agent:

- Security issues *(e.g. move an Agent to a secure location)*
- Bandwidth *(e.g. move an Agent to reduce latency)*
- CPU/Memory resources *(e.g. implement load-balancing)*
- Data gathering and validation *(e.g. an Agent may "police" several nodes)*

The COUGAAR architecture provides basic support for requesting that an agent be moved. This support comes in two basic pieces:

- A Plugin-friendly "MoveAgent" blackboard object, a Domain and factory for creating these object ("mobility" domain), and a Plugin named "MoveAgentPlugin" that hides the use of the low-level services. The MoveAgent object uses Relays to transfer the request between agents, which allows a "third-party" agent to request that another agent be moved. The MoveAgent object also includes a Status object that is updated with the success or failure result of the move operation.

- A "/move" Servlet (MoveAgentServlet) that can be used to "publishAdd()" a new MoveAgent object to the blackboard. This allows a web-based client to initiate agent moves.

The low-level services are loaded into all nodes and agents by default.

The MoveAgent support (MobilityDomain, MoveAgentPlugin, and MoveAgentServlet) are not loaded by default – they must be specified in the XML files.   These components use the low-level services to provide a simplified API to Plugin developers.

3.2.11.2 Configuration

As noted above, use of the MoveAgent blackboard object requires configuration-time loading of a special domain and plugins.

The domain is in "org.cougaar.core.mobility.ldm.MobilityDomain" and has the domain name "mobility". This domain can be specified in the "LDMDomains.ini" file, or in the agent ".ini" file; see the Domain documentation for details. For the "LDMDomains.ini" approach, add this line:

```
    mobility=org.cougaar.core.mobility.ldm.MobilityDomain
```

The "MoveAgentPlugin" should be loaded on both the move-requestor agent and the agent to be moved. For simplicity this plugin can be loaded into all agents. In an XML file this plugin is loaded just like any other plugin:

```
  <component class="org.cougaar.core.mobility.plugin.MoveAgentPlugin"/>
```

The "MoveAgentServlet" is also loaded like a plugin, and can safely be added to all agents. It internally specifies the "/move" Servlet URL path. Here is the XML line:

```
  <component

    class="org.cougaar.core.mobility.servlet.MoveAgentServlet"/>
```

Logging can be enabled for more verbose output on the "org.cougaar.core.mobility" package. This is enabled by adding a "-D" system property to specify the log file:

```
  -Dorg.cougaar.core.logging.config.filename=log.props
```

and creating a "configs/common/log.props" file:

```
  log4j.rootCategory=WARN,A1
  log4j.appender.A1=org.apache.log4j.ConsoleAppender
  log4j.appender.A1.layout=org.apache.log4j.PatternLayout
  log4j.appender.A1.layout.ConversionPattern=%d %-4r [%t] %-5p %c %x - %m%n
  log4j.category.org.cougaar.core.mobility=DEBUG
```

See the section on "Logging" for further details on log configuration.

3.2.11.3 Test example

A good initial test is to use the example "MobilePlugin" to see if an Agent can move between two Nodes.

Create the above "configs/common/log.props" file.

First modify your Node scripts for persistence as noted above:

```
# shell script for Node
java \
  -Xbootclasspath/p:$COUGAAR_INSTALL_PATH/lib/javaiopatch.jar \
  -Dorg.cougaar.core.logging.config.filename=log.props \
  [etc]
```

Next create this XML file:

```
<society
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  name="MyNode"
  xsi:noNamespaceSchemaLocation="http://www.cougaar.org/2003/society.xsd">
  <host name="localhost">
    <node name="OrigNode">
      <agent name="MobileAgent">
        <component
          class="org.cougaar.core.mobility.plugin.MoveAgentPlugin"/>
        <component
          class="org.cougaar.core.mobility.servlet.MoveAgentServlet"/>
      </agent>
    </node>
    <node name="DestNode"/>
  </host>
</society>
```

Modify your "configs/common/LDMDomains.ini" to add the mobility domain:

```
  LDMDomains.ini
  [ other domains ]
  mobility = org.cougaar.core.mobility.ldm.MobilityDomain
```

Run both "OrigNode" and "DestNode" as separate Nodes. Assume that OrigNode prints out a standard-out line that says it is running its HTTP server on port 8800, and that DestNode is running its HTTP server on port 8801.

In a browser go to this URL: http://localhost:8800/agents?scope=local. The agents on the OrigNode are displayed – the MobileAgent should be listed. Go to http://localhost:8801/agents?scope=local, and the MobileAgent should not be listed on the DestNode.

In a browser go to this URL: http://localhost:8800/$MobileAgent/move.

In the form below fill in the agent name as "MobileAgent", the origin node as "OrigNode", and the destination node as "DestNode". The "force-restart" option is ignored when the origin and destination nodes are different.

The agent is moved. You may see some logging output if your logging level is turned up – see the section on logger configuration to enable more verbose logging output.

Repeat the "/agents?scope=local" test above on both HTTP ports. The agent should now be listed on the 8801 node (DestNode). The "/move" servlet should display the MoveAgent object and the SUCCESS Status of the move operation.

This test confirms that both the Agent and it's blackboard were moved. Another test is to specify the same destination and source nodes, both with the "force-restart" option set to "false" in one run, and in another run with the "force-restart" set to "true".

The “force-restart” is a useful feature for testing, since it performs many of the same steps as moving and agent without relocating an agent. This can help debug memory-leaks and unstopped threads. More details are provided later in this document.

Of course, these tests can be repeated for more complex agent societies.

3.2.11.4 Creating and watching a MoveAgent blackboard object

Creating a MoveAgent object is fairly easy:

Get the DomainService and the MobilityFactory. This can be done within a Plugin by adding this code:

```
private DomainService domain;
private MobilityFactory mobilityFactory;
public void setDomainService(DomainService domain) {
     this.domain = domain;
     this.mobilityFactory = (MobilityFactory)
domain.getFactory(“mobility”);
}
```

If the mobilityFactory is null then the mobility domain wasn’t correctly specified in the “LDMDomains.ini”.

When the plugin is ready to create the object (within “execute()”):

- First request a MoveAgent Identifier Object with “mobilityFactory.createTicketIdentifier()”.
- Next create a move “Ticket” (org.cougaar.core.mobility.Ticket) that specifies which agent to move, the destination node, etc. Use the identifier generated by the factory to specify the Ticket’s identifier (i.e. “new Ticket(*the-id*, …)”).
- Use the mobilityFactory to create a MoveAgent object from the Ticket.
- “blackboard.publishAdd()” the MoveAgent object to the blackboard.

If the blackboard access is taking place within a servlet, then you should use a blackboard transaction. See the example “MoveAgentService” for details.

The MoveAgent object is transferred to the mobile agent’s blackboard (using a Relay). Plugins in both agents can subscribe to MoveAgent objects and watch for additions and changes. The MoveAgent object has a “getStatus()” field that contains a SUCCESS/FAILURE code, a message, and (in case of failure) an exception. The status is filled in when the move completes or has failed.

3.2.11.5 Requirements for Component developers

Agent mobility actually works more like “replicate + destroy_original”, not a serialization of the actual Agent instance. This is done for several reasons:

- The Agent and its child Components must be cleanly suspended for a safe state capture
- Threads and their stacks can not be serialized and moved
- Services and local resources must be cleanly released and reacquired

- Some Objects require special serialization and coordination
- A full serialized Agent might be excessively large

The following diagram illustrates the basic sequence for Component Mobility from a Component's point of view:

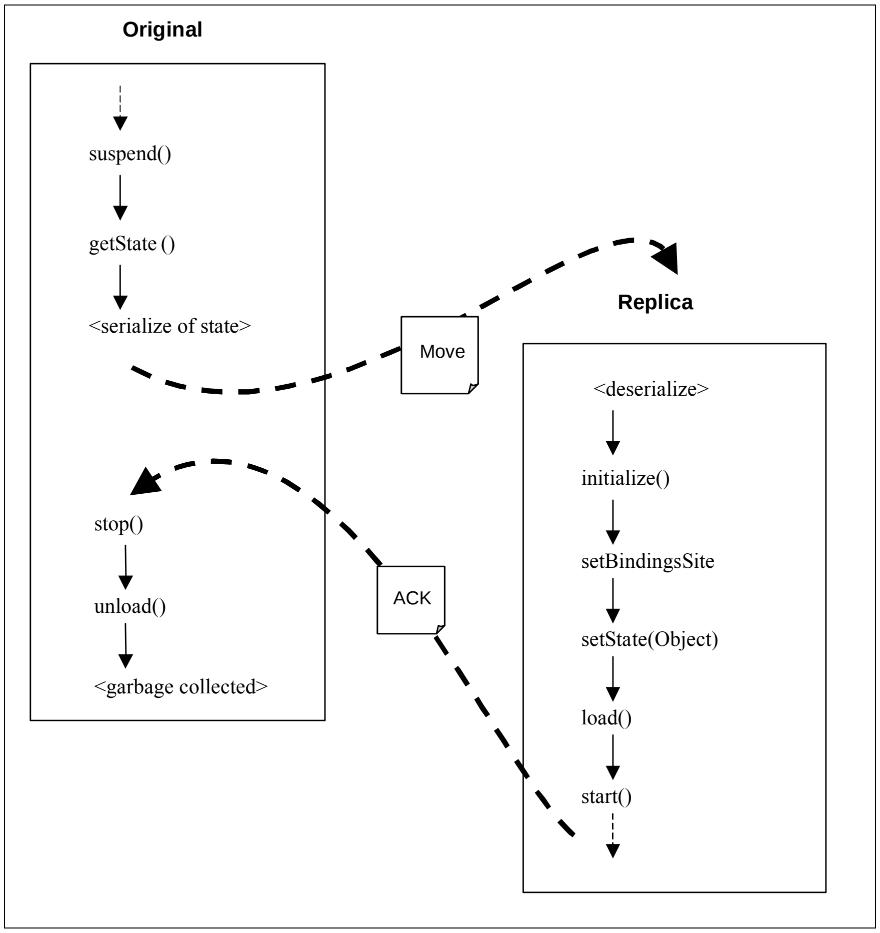


Figure 3-11: Agent Mobility Sequence

Components that need to do special suspend/stop/unload operations should override the default implementations in GenericStateModelAdapter. For example, a Component that manages a database should override these methods to clean up after itself and kill any Threads it might have running. In "unload" all Components should release any Services they have requested and revoke all Services they have provided to child Components, ideally in the reverse order that these were established.

Requested Services and ServiceRevokedListeners are *not* moved with the Components, in part because these Services may be unavailable at the new Agent location. During or after "load()" the Agent and child Component should request their Services again as necessary. Files, databases, system properties, and other localized resources are also not (automatically) moved with the Component.

All Components can implement the "`org.cougaar.core.component.StateObject`" interface to provide "Object getState()" and "void setState(Object)" methods. During state-capture "getState()" will be called, and at the new Agent location "setState(Object)" will be called by ContainerSupport after "initialize()" but before "load()". This state must be Serializable -- be careful not to drag the entire Node along!

It is recommended that Plugins keep their state in the Blackboard, and upon rehydration at the new Agent location they query the Blackboard for their state. If a Plugin instead implements StateObject then it should only refer to Blackboard Objects by UIDs, since Blackboard Objects (Tasks, etc) typically require special Serialization handling.

Containers should capture their children as part of their state. It is up to the Container implementation to decide which child components to capture as state, create by-default, and so on. This can partially be done with “org.cougaar.core.component.StateTuple” Objects, which is a pair of a ComponentDescription and a state Object. The advantage is that ContainerSupport's “add(Object)” will accept this StateTuple and call “setState(Object)” for the child at the appropriate time. See the PluginManager as an example.

3.2.12 Metrics Service

3.2.12.1 Overview

The Cougaar Metrics Service records and integrates metrics data captured by local sensors, such as CPU load and message traffic sensors, and advertises these real-time metrics to local components and remote user interfaces. For example servlet view snapshots, see the Operation Using Servlets page. The remainder of this manual covers the design, configuration, and developer use of the Metrics Service.

The Metrics Service constructs and maintains a data model which integrates raw sensor values from sources throughout any given Node's containment hierarchy, reconciling conflicting sensor view as needed. The resulting metrics are accessible to any client Component in that Node. The data-model perspective is always local to the given Node, though the information may also include data about, and in some cases from, remote Nodes and Agents.

The Metrics Service data model is updated in real-time in a demand-based way. As a result, a given data model becomes a more comprehensive view of the society as a whole not only as available data increases but also as demand increases. The services themselves do not do any measurements. Their function is only to collect and integrate raw values into a coherent and reliable data model, and to provide integrated best-guess metrics to clients that need them.

Note that the Metric Service role is limited to capturing the current state of the society and its underly resources. The Metrics Service gives the best guess at what the society is doing now from the local Nodes prespective. The Metrics Service does not support other kinds of statisics gathering, such as event/traps, time history, resource control, or policy management. Other Cougaar services handle these kinds of access mechanisms, even though the underlying statistics may be simular.

A Node's Metrics Service data model can be accessed by several ways, each tailored for a different type of consumer. When reading, Metrics are named based on looking of a formulain the data model. Since the lookup can follow relative relationship between resource contexts in the model, the lookup syntex is called a Path (see section 3.2.12.6).

- Components can read metrics using direct calls to the MetricsService (see section 3.2.12.4) interface. The interface supports both query or subscribe to specific metrics.
- Operators and debuggers can access tables of metrics using Servlets (see section 3.2.12.10).
- Management/Monitoring Agents can subscribe to metrics which are not collected locally, but which are collected on remote nodes. The Gossip Service (see section 0) will automatically transport these metrics to the local node.
- The Adaptivity Engine has a standard interface and naming convention for accessing metrics in rule-books.
- ACME and CSMART will be able to access metrics via a Servlet which returns queries in XML or Cougaar Properites format. (Not available in 10.4.0).
- The QuO framework for generating QoS adaptive Aspects and Comonents has access to metrics, by using QuO SysConds.

The Metrics Services has several ways for sensors to write metric values into the Node's local data model. When writing, Metrics are named using on a hieractical naming convention called Keys (see section 3.2.12.5).

- Components can write metrics using direct calls to the MetricsUpdateService (see section 3.2.12.4).

- Network/System Management Systems can import external information about the network and host configurations. Special DataFeeds can be created to import information in different formats. For example, default configuation information can be imported from a web page or file using a PropertiesDataFeed, see 3.2.12.8.

Sources of the Metrics Service can be explained as differing flows, with data from sensor code, network management configuration files, and even host-level probes. The Adding New Metrics section (see 3.2.12.8) describes each data flow, and the process in Cougaar in which to create new sensors.

3.2.12.2 Design and Architecture

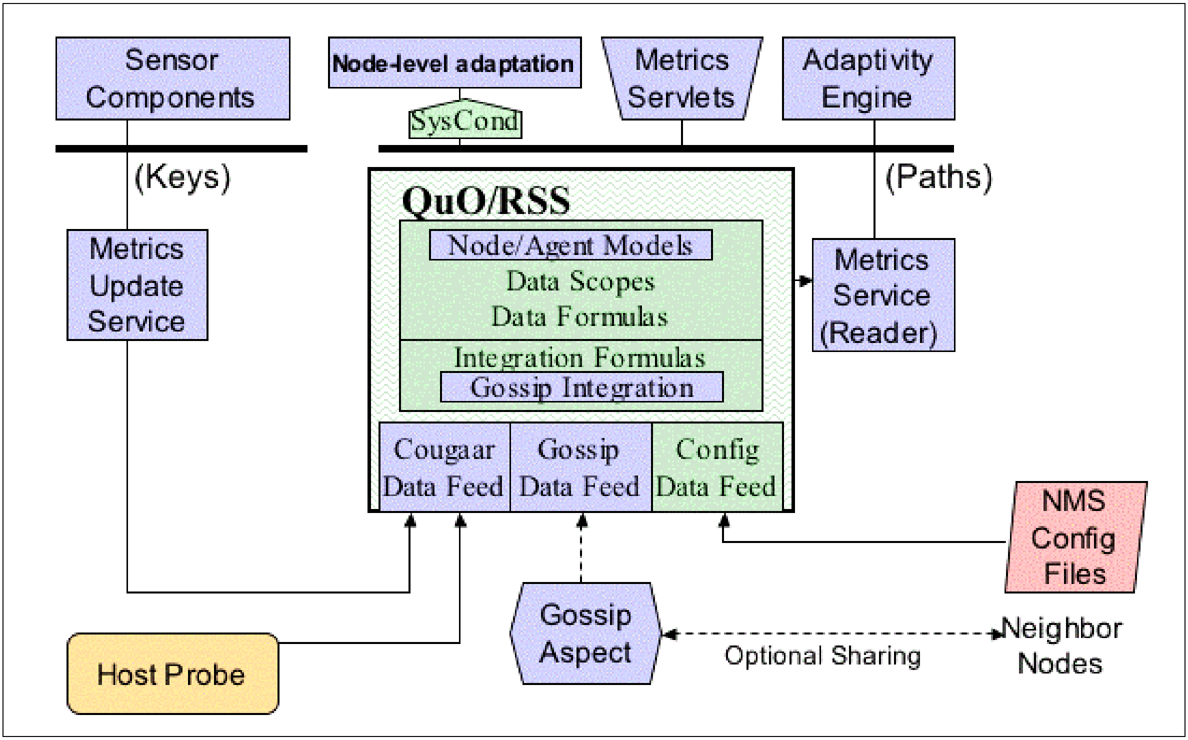


Figure 3-12: Metrics Service Internal Components

The core underlying infrastructure used by the Metrics Services is the Resource Status Service (RSS). The RSS accepts low-level tagged input in the form of key-value pairs, which it can process very efficiently. The entities which handle this input are known as DataFeeds. This low-level data is then propagated through a hierarchical graph of data-encapuslating nodes known as Resource Contexts, using both forward and backward chaining. Each Resource Context supports one or more DataFormulas, each of which in turn can calculate a value using the encapsulated data as well as data derived from other DataFormulas. DataFormulas, in other words, form a dependency chain that transforms and processes data. The final result of all this is raw sensor-like data coming in and processed domain-relevant data going out.

The specific subclasses of Resource Context in any given RSS depends on the domain. In the case of COUGAAR, interesting Resource Contexts represent Agents, Nodes, Hosts, inter-Agent data flows, inter-Host data flows, etc. Similarly, the specific subclasses of DataFeed in any given RSS depends on the raw data sources that are available. In the case of COUGAAR these would include URLs that provide default values in property-list syntax, real-time measured host data (load average, number of open sockets, etc), and data specified directly through the MetricsUpdateService.

Queries into the MetricsService use path specifications to indicate the location of a given Resource Context in the RSS knowledge base, and a formula on that Resource Context. If the Resource Context itself can handle the query it will do so. If not it will pass it to its parent. This makes it possible to ask, say, an Agent Resource Context for Node or Host data.

Each COUGAAR Node includes its own RSS, which is primarily responsible for handling local data in the Node. In order to provide a more global view, the GossipAspect was added. Gossip defines a new kind of DataFeed which is used for inter-RSS communication, and which piggybacks requests and responses onto ordinary COUGAAR messge traffic. This allows Nodes to share metrics without requiring any new form of inter-Node communication.

As an example of how data moves through the RSS, the rest of this section describes the flow of raw sensor data into the RSS from a COUGAAR Component via the MetricsUpdateService, and back out again as processed higher level data to some other Component via the MetricsService. *The rest of this is still TBD.*

For more details on keys, see section 3.2.12.5 . For more detail on paths, see section 3.2.12.6. And for more information on creating new metrics and the sources of data into these metrics, see Adding New Metrics (3.2.12.8).

Gossip Subsystem

The Gossip subsystem allows Metrics collected on one Node to be disseminated to other Nodes in the society. The Metrics are transfered by piggybacking Gossip objects as attributes of ordinary messages being sent between Nodes. Gossip takes two forms: requests for Metrics from a neighbor Node, and responses to those requests. Request are only made for Metrics used by the subsystems within the Node. Also, Gossip is sent only when the value of requested Metrics change. So if no Metrics are requested or the Metrics do not change, no Gossip is sent.

Gossip is spread based on the underlying message structure of the society. No additional messages are sent just to disseminate Gossip. Gossip can request metrics from neighbors which are N hops away which makes the Gossip system act like a limited range flooding algorithm. But in practice, N is set to only one hop.

The following figure shows the two types of Metrics sharing that can be accomplished by one-hop gossip.

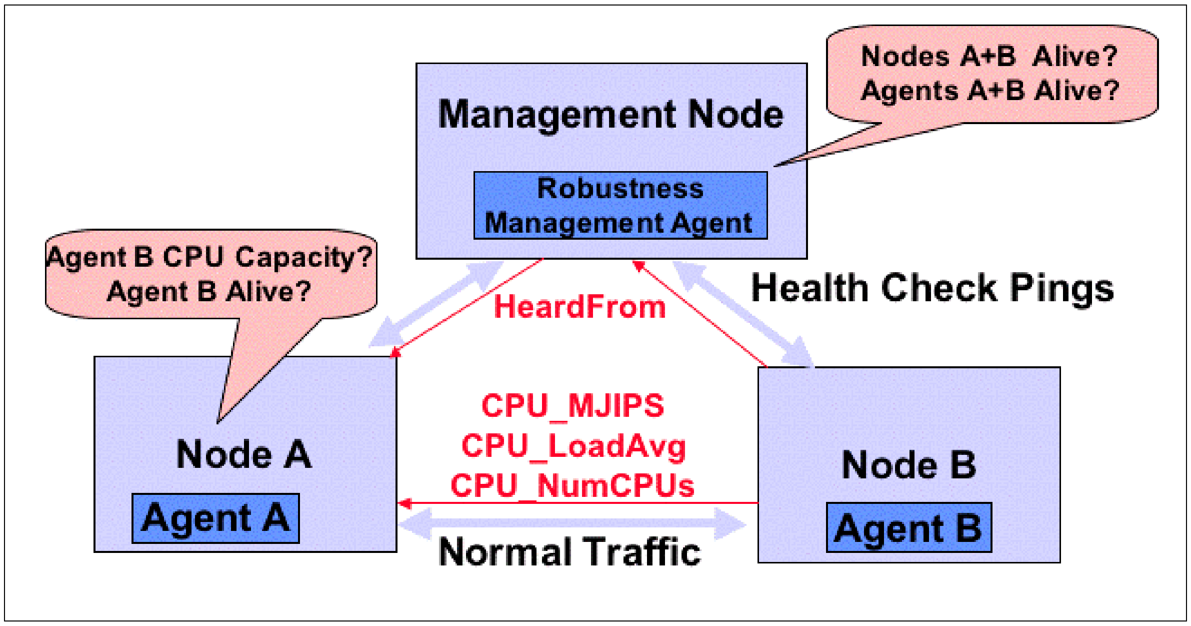


Figure 3-12a: Gossip Overview of Flow Between Nodes

First, Metrics about the neighboring Node can be sent and received directly from the neighbor on normal traffic. The first three messages exchanged between neighbors is enough to disseminate the static characteristics of both neighboring Agents/Nodes/Hosts, such as CPU capacity, Memory Size, Number of processors. Subsequent, messages will carry dynamic information, such as CPU Load Average.

Second, Gossip can also be sent indirectly through a chain of neighbors, which are all requesting the same Metrics. This means that one Node can have up-to-date Metrics about another Node, even though it has not recently received a message from the other Node. A typical example is a case wherethe intermediate Node contains a manager or name server Agent. These management Nodes tend to periodically send messages to a group of Nodes. The Gossip arrives on messages coming into the management Node and is propagated on all its out-bound messages. For example, a management Node may send health check pings to its managed Nodes. The sending pings will carry the dynamic Metrics for all the other managed Nodes to the pinged Node. (Note the recipient will only get information about the Metrics it has requested, even though the management Node may know addition information about other Metrics and other Nodes.) The reply from the pinged Node will carry the updated Metrics requested by the management Node.

Gossip Implementation

The Gossip implementation takes advantage of special characteristics of both the MTS and the Metrics service. Below is a figure which show the data flow between components which make up the Gossip subsystem. The red arrows are the flow of Key Requests and the blue arrows are Metric Value replies.

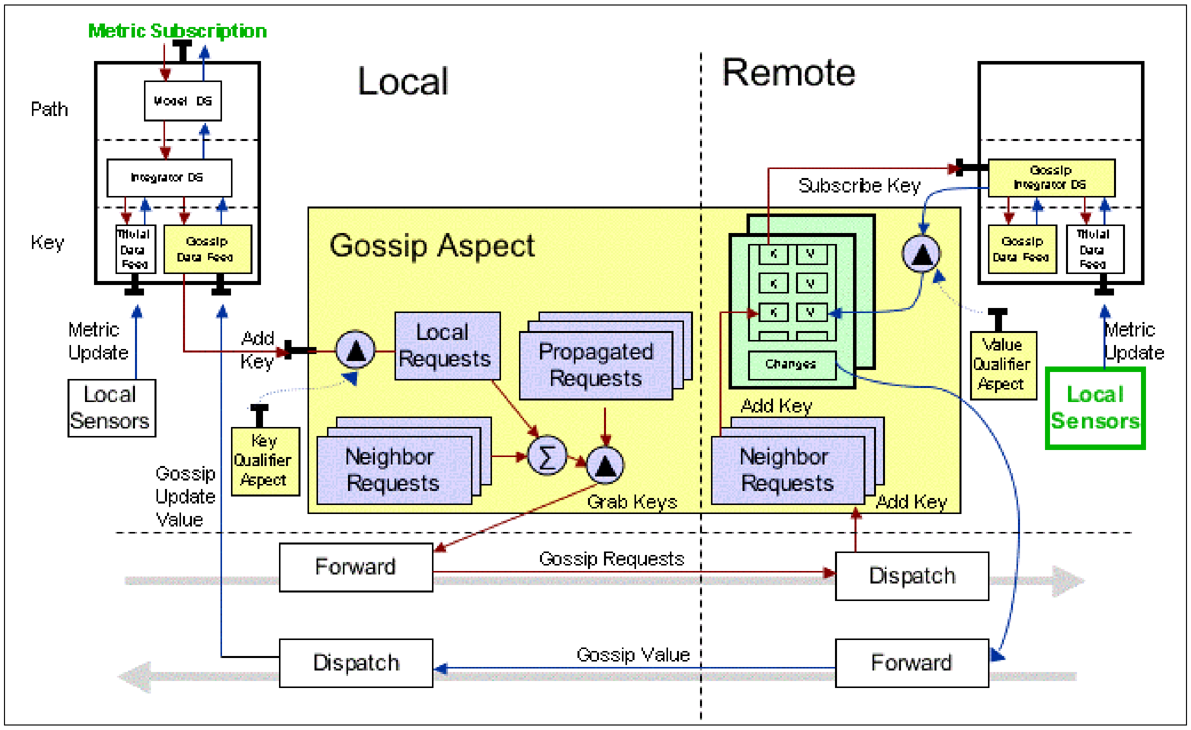


Figure 3-12b: Gossip Internal Components

Processing Trace

The following example trace shows how a Metric Request on one Node results on an update for a metric locally, even though the information about the Metric was collected remotely.

1. A Metrics Service Client makes a request for a Metric via a Path. The Metric Formula that implements that Path, depends on many raw Metrics Values, who's names are specified by Metrics Keys.

2. The GossipDataFeed listens for requests for Keys and forwards them on to the GossipKeyDistributionService (implemented by the GossipAspect).

3. The GossipAspect checks if the Key qualifies to be sent using the GossipQualifierService (implemented by the SimpleGossipQualifierComponent).

4. The Key is added to a list of local Key requests.

5. When a message is sent to a neighbor, the GosipAspect check if there are any outstanding keys that need to be requested to this neighbors. The requests are put in a KeyGossip object and added to an attribute of the messages. (Note a separate lists of outstanding requests is kept for each neighbor, since Gossip is sent opportunistically and different neighbors have different message patterns)

6. On the receiver Node, the request is stored in the Neighbor request record.

7. A subscription is added to the local Metrics service for the request. The special GossipIntegratorDS is used to subscribe to the Key, so that the GossipDataFeed can ignore these requests as non-local and not request them (again) for its neighbors. The GossipQualifierService is used get a MetricsNotificationQualifier from the SimpleGossipQualifierComponent. The MetricsNotificationQualifier is given to the MetricService.subscribeToValue as a way to limit values returned. The qualifier sets thresholds on the change in value and credibility of requested gossip metrics. If the Metric is has not changed enough or is not credible, it is not sent.

8. The GossipIntegratorDS acts like an IntegratorDS and subscribes to Keys from all the DataFeeds. Key subscriptions can be satisfied by any DataFeed, such as the local host probe, the local metric service, or from the GossipDataFeed (neighbor's neighbors).

9. When the local value has changed, the subscription is updated. The changed values are remembered for each neighbor.

10. When a message is sent back to the original requester, the values changed for the requester put into the Message Attributes.

11. On the receive-side of the original requester, the changed values are removed from the message and published in the GossipUpateService (implemented by the GossipDataFeed).

12. The GossipDataFeed will compare the new value with the Key's current value, which could have come from any other neighbor. If the Metrics is old, lower credibility, or the same Value, the new Metric is ignored. Otherwise it becomes the current value for the Key.

13. The new value is propagated to the IntegratorDS formulas. These in-turn will propagate to other subscribing formulas, eventually calling the original Metrics service Subscriber.

3.2.12.3 Use Cases and Examples

Example Metrics Reader

An example Metrics Service subscriber can be found in the core module.

```
package org.cougaar.core.qos.metrics;

/**
 * Basic Metric Service Client subscribes to a Metric path given in
 * the Agent's .ini file and prints the value if it ever changes
 */
public class MetricsClientPlugin
    extends ParameterizedPlugin
    implements Constants
{
    protected MetricsService metricsService;
    private String paramPath = null;
    private VariableEvaluator evaluator;

  /**
   * Metric CallBack object
   */
    private class MetricsCallback implements Observer {
      /**
       * Call back implementation for Observer
       */

      public void update(Observable obs, Object arg) {
         if (arg instanceof Metric) {
           Metric metric = (Metric) arg;
           double value = metric.doubleValue();
           System.out.println("Metric "+ paramPath +"=" + metric);
         }
     }
    }


  public void load() {
    super.load();
    ServiceBroker sb = getServiceBroker();

    evaluator = new StandardVariableEvaluator(sb);

    metricsService = ( MetricsService)
      sb.getService(this, MetricsService.class, null);

    MetricsCallback cb = new MetricsCallback();
    paramPath = getParameter("path");
    // If no path is given, default to the load average of the
    // current Agent.
    if (paramPath == null)
     paramPath ="$(localagent)"+PATH_SEPR+"LoadAverage";

    Object subscriptionKey=metricsService.subscribeToValue(paramPath,
cb,
                                        evaluator);

  }
}
```

Example Metrics Writer

An example Metric Service writer has the following form:

```
package org.cougaar.core.qos.metrics;

public class MetricsTestComponent
    extends SimplePlugin
    implements Constants
{

    public void load() {
     super.load();

     ServiceBroker sb = getServiceBroker();
     MetricsUpdateService updater = (MetricsUpdateService)
         sb.getService(this, MetricsUpdateService.class, null);

     String key = "Current" +KEY_SEPR+ "Time" +KEY_SEPR+ "Millis";
        // Publish the current time
     Metric m = new MetricImpl(new Long(System.currentTimeMillis()),
                         SECOND_MEAS_CREDIBILITY,
                         "ms",
                                   "MetricsTestComponent");
     updater.updateValue(key, m);

    }

}
```

3.2.12.4 APIs

org.cougaar.core.qos.metrics.MetricsService

`Metric getValue(String path, VariableEvaluator evaluator, Properties qos_tags)`

> The most general query function in the Metric Service. The `path` specifies the metric to be returned. For a description of the syntax, see the Path section (3.2.12.6). The `evaluator` is used to handle shell-variable-style references in generic paths. For more details on this, see VariableEvaluator (in section 3.2.12.4). The `qos_tags` are for future use.

`Metric getValue(String path, Properties qos_tags)`

> As above but with no variable replacement.

`Metric getValue(String path, VariableEvaluator evaluator)`

> As above but with no QoS tags.

`Metric getValue(String path)`

> As above but with neither variable replacement nor QoS tags.

`Object subscribeToValue(String path, Observer observer, VariableEvaluator evaluator, MetricNotificationQualifier qualifier)`

> The most general subscription function in the Metric Service. The `path` specifies the metric to subscribe to. For a description of the syntax, see the Path section (3.2.12.6). The `observer` is the entity that will receive a callback when the specified metric changes. The `evaluator` is used to handle shell-variable-style references in generic paths. For more details on this, see VariableEvaluator (in section 3.2.12.4). The `qualifier` is used to restrict the frequency of callbacks, for example by specifying the smallest change that should trigger one. For more details, see MetricNotificationQualifier (in section 3.2.12.4). The value returned by this method is a subscription handle and should only be used for a subsquent call to `unsubscribeToValue`.

`Object subscribeToValue(String path, Observer observer, VariableEvaluator evaluator)`

> As above but without callback qualification.

`Object subscribeToValue(String path, Observer observer, MetricNotificationQualifier qualifier)`

> As above but without variable replacement.

`Object subscribeToValue(String path, Observer observer)`

> As above but with neither variable replacement nor callback qualification.

`void unsubscribeToValue(Object handle)`

> Terminates a previously established subscription. The handle is as returned by one of the `subscribeToValue` calls.

org.cougaar.core.qos.metrics.MetricsUpdateService

`void updateValue(String key, Metric value)`

Adds a new or updated metric to the Metrics Service. The syntax of a key is described in the Keys section (3.2.12.5).

org.cougaar.core.qos.metrics.Metric

`Object getRawValue()`

Returns the raw value of the metric. If you know what type the value should be, use one of the type-specific reader methods instead.

`String stringValue()`

Returns the raw value of the metric as a `String`, via `toString()`.

`byte byteValue()`

Returns the raw value of the metric as a `byte` if it's a `Number`. Otherwise returns 0.

`short shortValue()`

Returns the raw value of the metric as a `short` if it's a `Number`. Otherwise returns 0.

`int intValue()`

Returns the raw value of the metric as a `int` if it's a `Number`. Otherwise returns 0.

`long longValue()`

Returns the raw value of the metric as a `long` if it's a `Number`. Otherwise returns 0.

`float floatValue()`

Returns the raw value of the metric as a `float` if it's a `Number`. Otherwise returns 0.

`double doubleValue()`

Returns the raw value of the metric as a `double` if it's a `Number`. Otherwise returns 0.

`char charValue()`

Returns the raw value of the metric as a `char` if it's a `String` or a `Number`. Otherwise returns '?'.

`boolean booleanValue()`

Returns the raw value of the metric as a `boolean` if it's a `Boolean`. Otherwise returns `false`.

`double getCredibility()`

Returns the credibility of the metric as value between 0.0 (no credibility) and 1.0 (absolute certainty).

`String getUnits()`

Returns the units of the metric. In the current release this field is not generally filled in .

`String getProvenence()`

Returns a tag indicating the source of the metric. In the current release this field is not generally filled in .

`long getTimestamp()`

Returns the time at which the metric was entered into the Metric Service.

`long getHalflife()`

Returns the time after which the credibility of this metric will begin to degrade. For 10.4 the degrade feature is not implmented.

org.cougaar.core.qos.metrics.MetricNotificationQualifier

`boolean shouldNotify(Metric metric)`

Returns true if the given metric has changed enough to generate notifications to listeners.

org.cougaar.core.qos.metrics.VariableEvaluator

Path specifications in the Metrics Service support a limited notion of variables for which actual values are substituted dynamically. The syntax is taken from make: `$(var)`. The evaluation of such variables is handled by a user-supplied `VariableEvaluator` instance. The parsing and substitution is handled by the Metrics Service.

A useful base-class for classes which implement this interface `StandardVariableEvaluator`, which can supply values for the variables `localagent`, `localnode` and `localhost`.

`String evaluateVariable(String var)`

Given a variable reference in a path this method should return the String to be spliced in as a replacement. The argument is the variable name itself, without the $ or the parens.

3.2.12.5 Metric Keys

This section discusses the structure of the Metrics Keys name-space and defines common Keys.

Metrics are written into the MetricsUpdateService (see section 3.2.12.4).as Key-Value pairs. While the key itself is just a string, we impose a structure to help organize the information, loosely modeled after SNMP MIBs. The value is a Metric (see section 3.2.12.4), a structured object with slots for value, credibility, time-stamp, units, etc.

At runtime, each Node has a store of Key-Value pairs. When the MetricsUpdaterService receives a new Key-Value pair, it looks up the Key in the store. If the value has changed, all the Metrics Formulas that have subscribed to this Key will be called-back with the new Value. The Metrics Service in each Node has one and only one effective Value for each Key, which is derived through a complicated set of integration rules based on credibility and timestamps.

Normally, Key-Value pairs are not read from the Metrics Service, but are only processed by internal formulas. One debugging trick to view the effective value of a key is to the Path Integrater (3.2.12.6), which takes a Key as a parameter and returns the effective metric. Currently, there is no way to list all the Keys available to a Node.

Key Syntax

A Metrics Key is a string divided into fields using the Key Separator, e.g.
`Host_128.33.15.114_CPU_Jips`

The current Key Separators is "_". But your code you should use the string constant `KEY_SEPR` defined in `org.cougaar.core.qos.metrics.Constants`:

```
"Host" +KEY_SEPR+ "128.33.15.114" +KEY_SEPR+ "CPU"

      +KEY_SEPR+ "Jips"
```

Notice that some of the fields are types (e.g. Host, CPU, Jips) and others are identifiers (e.g. 128.33.15.114). The types are fixed and case sensitive. Identifiers are variable and define branches in naming hierarchy.

In the following section we will use the convention of writing identifier fields in brackets [ ], and using "_" as the Key Separator

```
Host_[Host IP]_CPU_Jips
```

Host

Host Keys start with identifying the host and then have several optional field for different host characteristics. Host identifiers are raw IP V4 addresses, not domain names.

`Host_[Host IP]_CPU_Jips`

CPU capacity in Java Instructions Per Second. JIPS is determined through a benchmark.

`Host_[Host IP]_CPU_loadavg`

CPU load average, i.e. the average number of processes that are ready to run.

`Host_[Host IP]_CPU_count`

The number of CPUs in this host

`Host_[Host IP]_CPU_cache`

Size of CPU level 2 cache

`Host_[Host IP]_Memory_Physical_Total`

Total Physical Memory

`Host_[Host IP]_Memory_Physical_Free`

Free Physical Memory

`Host_[Host IP]_Network_TCP_sockets_inuse`

TCP sockets in use

`Host_[Host IP]_Network_UDP_sockets_inuse`

UDP sockets in use

IP Flow

Network Keys start with identifying the IP Flow and then have several optional field for different network characteristics. End-point addresses are raw IP V4 addresses and not domain names.

`Ip_Flow_[src IP]_[dst IP]_Capacity_Max`

The maximum bandwidth (kbps)for path across the network, i.e. with no competing traffic.

`Ip_Flow_[src IP]_[dst IP]_Capacity_Unused`

The expected available bandwidth (kbps) for a path across the network, i.e. the max minus the competing traffic.

Site Flow

Sites are an IP subnetwork which can be represented with a simple mask (number of bits with leading 1's). Site_Flows between sites can be used as defaults, instead of specify specific Ip_Flows. For example, `Site_Flow_128.89.0.0/16_128.33.15.0/28_Capacity_Max`

`Site_Flow_[src IP/mask]_[dst IP/mask]_Capacity_Max`

Maximum bandwidth (kbps) between Sites. The current formulas can model asymmetric bandwidth between Sites, by publishing Site_Flows for both direction. If only one direction is published, the formulas will assume the bandwidth is the same in both directions.

Agent

Agents are identified by their message Id.

`Agent_[Message ID]_HeardTime`

System.currentTimeMillis() time-stamp for the last time some component has heard from this agent

`Agent_[Message ID]_SpokeTime`

System.currentTimeMillis() time-stamp for the last time some component has attempted to speak to this agent

Node

Nodes are identified by their message Id. Currently, there are no Node-specific Keys

3.2.12.6 Metric Paths

This section discusses the structure of the Metrics Paths and defines common Paths.

Metrics read from the MetricsService (see section 3.2.12.4) are the values from a real-time model of the status of the Cougaar Society and system resources. The MetricsService allow access to *formulas* in the model. Formulas are relative to a *Scope*, which define instances of modeled entries and there relationship between each other.

Containment is a major relationship which is modeled. Child scopes inherit all the formulas of its parent. The useful containment relationship in Cougaar are `Host->Node->Agent`. For example, Agents have all the formulas of their hosts, and when agents moves to a new host the metrics track new hosts values.

Path Syntax

A Path is a series of Contexts followed by a Formula. If there is more than one child Context of the same type in the parent Context, then the child Context is narrowed by Parameters. Each Context has a fixed number of parameters and parameter order is important. The Context type is used as the name of the context. Context and Formula names are case sensitive. For example:

```
IpFlow(128.33.15.114,128.33.15.113):CapacityMax
```

The separator between Contexts and formulas is the ":" and the separator between parameters is the ",". But your code you should use the constant `PATH_SEPR` defined in interface `org.cougaar.core.qos.metrics.Constants`:

```
"IpFlow(128.33.15.114,128.33.15.113)" +PATH_SEPR+ "CapacityMax"
```

The following sections will use the convention of writing variable parameter fields in brackets [] for variable parameters; using ":" as the Path Separator; and "," as the parameter separator. The syntax is as follows:

```
ContextType1([parameter1],[parameter2]...):ContextType2([parameter]):For
mula(parameter)
```

The Averaging Period are parameters for some Formulas. The Averaging Periods are: "10", "100", and "1000". This section will use the convention 1xxxSecAvg to represent the averaging period.

Also, please use the constants interface in core module `org.cougaar.core.qos.metrics.Constants` whenever possible. This will allow compile time detection of errors, when the Metrics Service interface inevitably changes in the future

Host Scope

Host Contexts can be accessed at root level and take one parameter which is the Host address. The Host address can be IP V4 address or the domain name. Hosts contexts are not contained in other Contexts.

`Host([host Addr]):EffectiveMJips`

> The expected per thread Millions of Java Instructions Per Second. The formula takes into account the base CPU JIPS, the number of CPUs, and the Load Average on the Host

`Host([Host Addr]):Jips`

> CPU capacity in Java Instructions Per Second. JIPS is determined through a benchmark.

`Host([Host Addr]):LoadAverage`

> CPU load average, i.e. the average number of processes that are ready to run.

`Host([Host Addr]):Count`

The number of CPUs in this host

`Host([Host Addr]):Cache`

Size of CPU level 2 cache

`Host([Host Addr]):TotalMemory`

Total Physical Memory

`Host([Host Addr]):FreeMemory`

Free Physical Memory

`Host([Host Addr]):TcpInUse`

TCP sockets in use

`Host([Host Addr]):UdpInUse`

UDP sockets in use

Node Context

Node Contexts can be accessed at root level and take one parameter which is the the Node's Message address. The Node is also contained within a Host Context and inherits all the Host Context's formulas.

`Node([Node ID]):CPULoadAvg(1xxxSecAvg)`

The average number of threads working for all component on this Node during the Averaging Period.

`Node([Node ID]):CPULoadMJips(1xxxSecAvg)`

The average MJIPS used by all the threads for all components on this Node during the Averaging Period.

`Node([Node ID]):MsgIn(1xxxSecAvg)`

The average messages into all Agents on this Node during the Averaging Period.

`Node([Node ID]):MsgOut(1xxxSecAvg)`

The average messages out of All Agents this Node during the Averaging Period.

`Node([Node ID]):BytesIn(1xxxSecAvg)`

The average number of bytes (for all messages) into all Agents on this Node during the Averaging Period.

Node([Node ID]):BytesOut(1xxxSecAvg)

The average number of bytes (for all messages) out of All Agents on this Node during the Averaging Period.

Node([Node ID]):VMSizet

Size in Bytes of the Node's VM

Node([node Addr]):Destination([Agent ID]):MsgTo(1xxSecAvg)

Message per second from all agents on the Node, to the destination Agent

Node([node Addr]):Destination([Agent ID]):MsgFrom(1xxSecAvg)

Message per second to any agent on the Node, from the destination Agent

Node([node Addr]):Destination([Agent ID]):BytesTo(1xxSecAvg)

Bytes per second from all agents on the Node, to the destination Agent

Node([node Addr]):Destination([Agent ID]):BytesFrom(1xxSecAvg)

Bytes per second to any agent on the Node, from the destination Agent

Node([node Addr]):Destination([Agent ID]):AgentIpAddress

Ip address of destination Agent

Node([node Addr]):Destination([Agent ID]):CapacityMax

Maximum capacity of the network between Node and destination Agent.

Node([node Addr]):Destination([Agent ID]):OnSameSecureLAN

True if Node and Destination are on the same secure LAN. The current formula just checks if the network capacity between the Node and Agent is greater than or equal to 10 Mbps.

Agent Context

Agent Contexts can be accessed at root level and take one parameter which is the the Agent's Message address. The Agent Context is also contained within a Node Context and inherits all the Node Context's formulas. When an Agent moves to a new Node, the Agent Context will be moved the the corresponding Node, i.e. all the re-wiring is automatic, but may be slightly delayed due to discovery issues.

Agent([Agent ID]):LastHeard

Seconds since some component has heard from this agent. This can be from any source such as *successful* communication, acknowledgment, or gossip.

Agent([Agent ID]):LastSpoke

Seconds since some component attempted to Speak to the Agent. The attempt does not have to be successful. For example, if a failed message is retied, LastSpoke will be the time of last retry

Agent([Agent ID]):SpokeErrorTime

Seconds since last error in communications. This is a large number with 0.0 credibility, if no error has occurred.

Agent([Agent ID]):CPULoadAvg(1xxxSecAvg)

The average number of threads working for this Agent during the Averaging Period.

Agent([Agent ID]):CPULoadMJips(1xxxSecAvg)

The average MJIPS used by all the threads for this Agent during the Averaging Period.

Agent([Agent ID]):MsgIn(1xxxSecAvg)

The average messages into this Agent during the Averaging Period.

Agent([Agent ID]):MsgOut(1xxxSecAvg)

The average messages out of this Agent during the Averaging Period.

Agent([Agent ID]):BytesIn(1xxxSecAvg)

The average number of bytes (for all messages) into this Agent during the Averaging Period.

Agent([Agent ID]):BytesOut(1xxxSecAvg)

The average number of bytes (for all messages) out of this Agent during the

Agent([Agent ID]):PersistSizeLast

Size in Bytes of the last persistence file for this agent

IpFlow Context

IpFlow Context can be accessed at root level and takes two parameters which is the source and destination host addresses. The Host address can be the IP V4 address or the domain name.

IpFlow([SrcAddr],[DstAddr]):CapacityMax

Maximum Bandwidth between two hosts

IpFlow([SrcAddr],[DstAddr]):CapacityUnused

Unused Bandwidth between two hosts

AgentFlow Context

AgentFlow Context can be accessed at root level and takes two parameters which is the source and destination agent message address.

Since 11.2 the AgentFlow context is no longer used, because it added too many formulas to the metrics service when societies are large.

`AgentFlow([SrcAddr],[DstAddr]):MsgRate(1xxxSecAvg)`

Messages per Second from Source Agent to Destination Agent."

`AgentFlow([SrcAddr],[DstAddr]):ByteRate(1xxxSecAvg)`

Bytes per Second from Source Agent to Destination Agent.

3.2.12.7 Gossip-based Dissemination Design

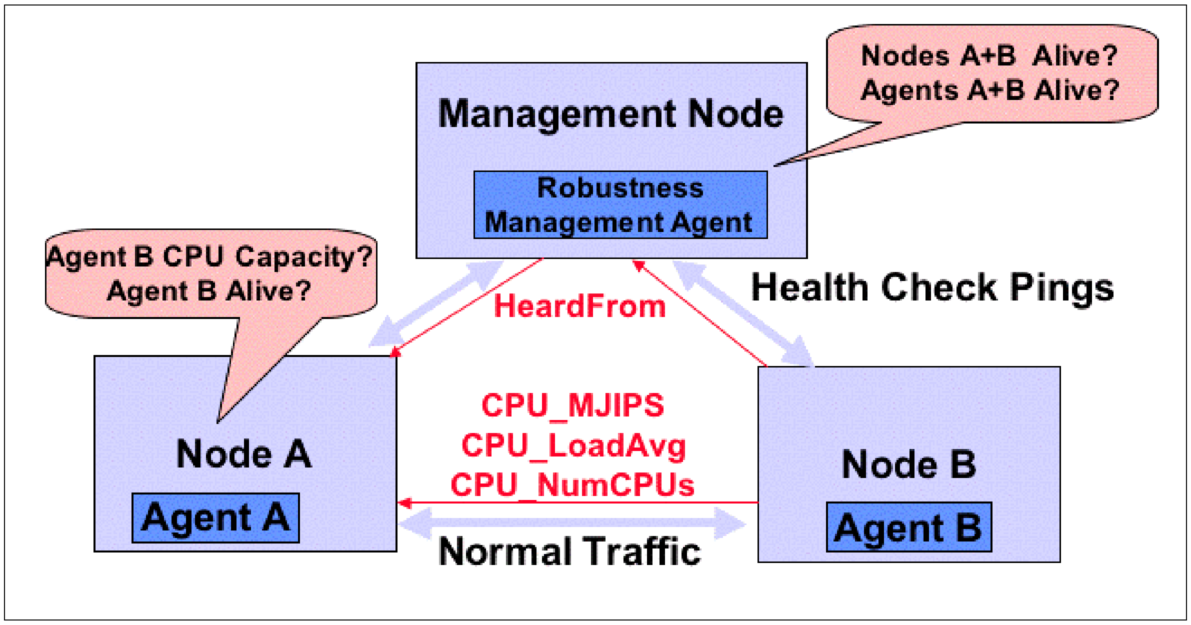


Figure 3-13: Gossip Overview of Flow Among Nodes

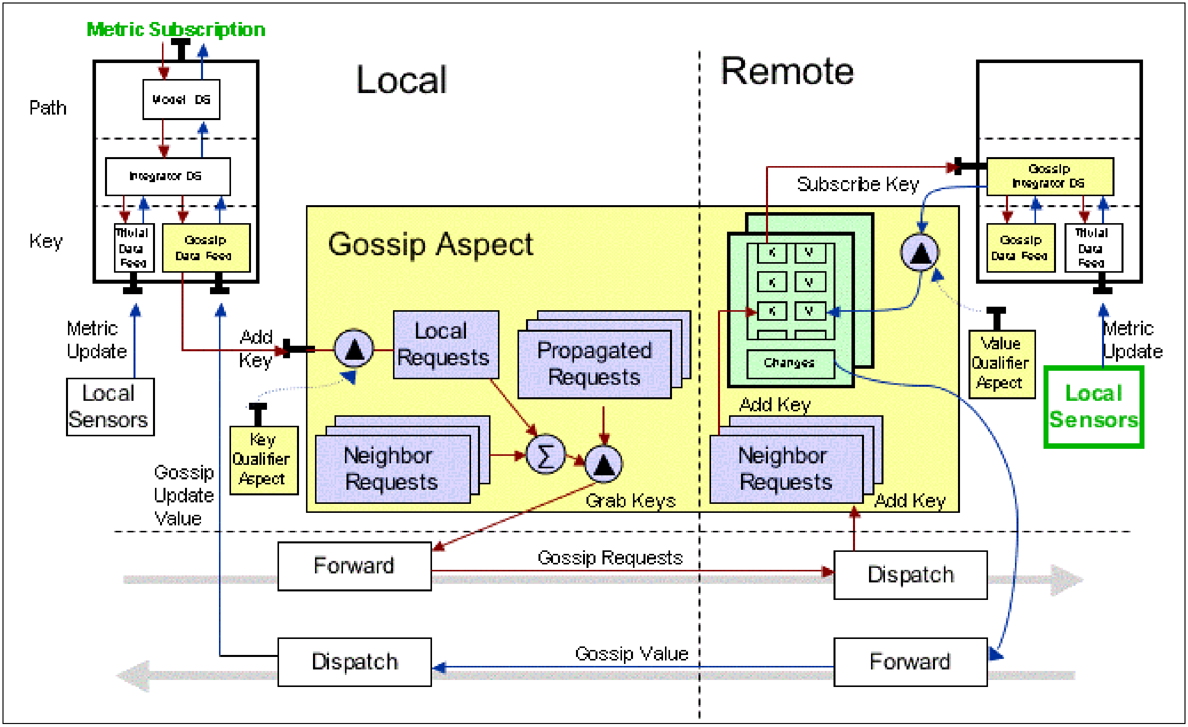


Figure 3-14: Gossip Internal Components

3.2.12.8 Adding New Metrics

The Metrics Package has helper classes for collecting raw sensor values, converting them into metrics, adding them into the path model, and displaying them in a servlet. In this section, we explain how these helper classes can be used to add new metrics to the Metrics Service.

The Metric Service gets its data from several sources and integrates them into a up-to-date picture of the environment in which the Node finds itself. There are three ways of originating data into the Metrics Service, these are:

- Sensor data from Cougaar code measures the internal consumption of resource by Agents and Nodes. For example, the number of messages sent by an Agent or the amount of CPU or memory consumed by an Agent

- Network Management Configuration Files are used to get information about network characteristics from external Network Management Systems.

- Host Level probes are used to measure the characteristics of the Node's host

All these sources of data use the same "patterns" for capturing and processing the data. The processing of Metrics can be described as a data flow, which is processed by different components. Each component does a different job in converting raw sensor data into usable metrics. Figure 3-14a. represents the processing of data as it flows through the Metrics Service:

- *Sensors* instrument the system and store raw statistics records. The records can be accessed via a service, which takes a snapshot of the record.

- *Rate-Converters* gathers snapshots of sensor records and converts them into metrics. The metrics are published into the Metrics service

- *Metrics Service* models the environment. Internal to the model, formulas can depend on the raw metrics that are published into the Metrics service. The formulas can be accessed from the metrics service using paths.

- *QoS Adaptive Components* use the Metrics Service environment model to adapt to the Node's processing to changes in the environment.

- *Servlets* are used to view both Metrics and raw sensor records.

The patterns used to create these components are described in the following sections. Some explanations of this process are more easily understood when referencing specific pieces of the code, which will be highlighted in green for emphasis.

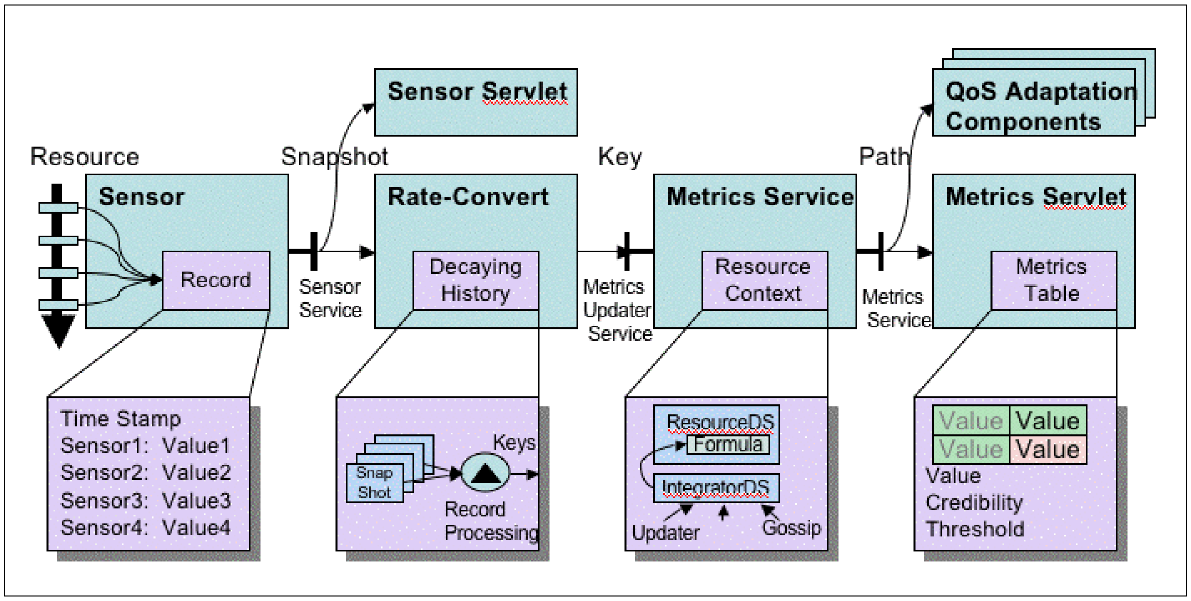


Figure 3-14a: Sensor Dataflow

Sensors

Sensors have probes in the critical path of the basic operation of the Node. The probes must have very little overhead, so they have limited functionality. Probes only have time to detect that an event has happened and tally the results in a statistics record. The record holds many raw sensor values. There are 3 kinds of raw sensors values, which can reside in either the code or host probes. which are:

- *Gauge:* This is an instantaneous value, such as utilization, current resource capacity.

- *Counter:* Continuously increments some value like MsgsIn/Out or BytesIn/Out. org.cougaar.mts.std.AgentStatusAspect is an example of such a Sensor, continuously counting the MsgIn/Out & BytesIn/Out for local agents of a node, and the sums for the node itself. org.cougaar.core.qos.metrics.AgentLoadServlet is one gui where they user can observe this data.

- *Integrator:* Counter that is a weighted summation of delta t * the value in that period. Integrators are used find the average value of gauge values over a time period. For example, Agent's CPU consumption is measured by the average number of threads used to process the Agent's code. (This is call the Agent's Load Average) org.cougaar.core.thread.AgentLoadSensorPlugin is an example of such a Sensor, continuously counting the MsgIn/Out & BytesIn/Out for local agents of a node, and the sums for the node itself.

An external clients of the Sensor component can sample the statistics record using a service offered by the Sensor. When a client samples data, it takes a *Snapshot* at a point in time, resulting in a collection of these snapshots which need to be processed to create usable Metrics. Notice the processing of the raw statistics is not done as part of the critical path, but done by an independent component with its own threads. The processing of snapshots is the subject of the next section.

Rate Conversion

Rate converter are clients to a sensor service and periodically take snapshots of the sensors record. The two snapshots are processed to create meaningful metrics. The time between snapshots defines the averaging interval for the metric. The processing is different for each field in the record, depending on the kind of sensor value. For example to calculate rate of messages being sent by an agent, the sensor value for the count of messages from the agent is needed. The change in count is subtracted from two snapshots and the change in time is subtracted from the snapshot timestamps. The message rate is calculated by dividing the change in count by the change in time.

Rate converters have the following internal structure:

- *Record Processing:* Data snapshots are collated and computed as a rolling average of changes in time & value. A Decaying History object is used to store the snapshots and call the record processing.

- *Decaying History:* Is an object which coordinates the storage and processing of snapshots. It maintains a decay stor of snapshots and call the processing of data snapshots for several averaging periods.(10, 100, 1000-second averages are common). org.cougaar.core.qos.metrics.AgentStatusRatePlugin is one plugin which implements the DecayingHistory's newAddition(), which updates the Metrics Service with the processed value.

- *Key generation:* After the snapshots are processed and represent a value per some time duration, a Key is generated in relation to the rolling average of time, and used to insert this Metric into the Metrics Service.

Metrics Service Models

The Metrics Service maintains a model of the environment for the Cougaar society. The model includes contexts for the Hosts, Nodes, and Agents in the society. Each Node maintains its own instance of the model and updates it using Metrics published into the Metrics Updater Service.

- *Resource Context:* model a specific entity in the environment. Resource Contexts can contain child contexts. The children inherit all the attributes of its parent. For example, an Agent context is a child of Node, which is a child of Host. The Agent has a CPU count attribute that it inherits from its Host. org.cougaar.core.qos.rss.AgentDS is an example of a Resource Context.

- *Formula:* calculate attributes of a context. Formulas may depend on other formulas or raw Metrics published to the Metrics service. The org.cougaar.core.qos.rss.DecayingHistoryFormula is a helper class which create formulas that get their values directly from Metrics published by a Decaying History object.

- *IntegratorDS* integrate published metrics from many DataFeeds into a single value. Resource Contexts Formulas get the values for published Metrics by subscribing to Integrator formulas, instead of getting the Metrics directly from the feeds themselves. The IntegratorDS is part of the QuO RSS, which is outside of the Cougaar code.

Metrics Servlet

Evaluating the Metrics Service & Observing the system is usually done through GUIs in the Cougaar system, using Cougaar Servlets. In the Metrics Service, there resides many differnent servlets pertaining to communications, resource constraints, etc., please refer to Operation Using Servlets for the different kinds of servlets currently available.

Metrics Servlets all extend a Metrics Servlet template, which is formatted to make easy interpretation of the Metrics with tables, background & text colors. Here's a quick guide to interpreting a Metrics Servlet: Refer to the servlet page or the picture below for a visual.

- *query:* This is a servlet window into the Metrics Service, or when a servlet is queried it reveals a current snapshot of those current Metrics Service values.
- *Value:* Dislayed as the base text, this is the Metrics name (e.g. AgentA Load)
- *Credibility:* Displayed as the color of the text: light gray to indicate that the metric value was only determined as a compile-time default; gray to indicate that the metric value was obtained from a configuration file; black to indicate that the metric value was obtained from a run-time measurement. A metric's credibility metric is an approximate measure of how much to believe that the value is true. Credibility takes into account several factors, including when, how, by whom a measurement was made.
- threshold: Displayed in the credibility, and when this this certain Metric's value crosses a threshold, it may be interesting enough to warrant attention. This is shown by the color of the background. A green background indicates that the value is in the normal range. A yellow background indicates that the values is in a typical ground state, i.e nothing is happening. A red background indicates that the value has crossed a metric-specific threshold and may be interesting.

Sensor Servlet

Sensor Servlets allow viewing of a raw sensor servlet. The main use of sensor servlets is for debugging.

QoS Adaptive Components

QoS Adaptive components subscribe to Metrics using the Metrics Service. How to create QoS Adaptive Components is out of scope for this section.

Constants

The same name for a metric must be used throughout all components of the metrics service. To aid consistency, we recommend using the metrics constants. The list of constants is documented in org.coguaar.core.qos.metrics.Constants

- Metrics Name: Quite obviously, the name of the Metric, as it applies to its measurement & key generation. Ususally, CPU_LOAD_AVG, CPU_LOAD_MJIPS, MSG_IN, MSG_OUT, BYTES_IN, BYTES_OUT
- Averaging intervals: Used in the Key of a Metric, referring to the averaging interval of value to time of that value. The current intervals are: 1_SEC_AVG, 10_SEC_AVG, 100_SEC_AVG, 1000_SEC_AVG.

3.2.12.9 Configuration

The Metrics Service is made of many components all of which are optional. If no components are added the Metrics Service will act as a black-hole with MetricsUpdates writing, but no call-backs or queries returned from the MetricsService (reader). The following types of components can be configured. The specific components are listed in the following sections along with their function, insertion point and module (jarfile)

Metrics Servlets

The following components allow viewing of Metrics through servlets. (See 3.2.12.10). The components are in core.jar and should be loaded into the NodeAgent.

`org.cougaar.core.qos.metrics.MetricsServletPlugin"`

> Loads the Metric Service Servlets, which are: AgentLoadServlet, RemoteAgentServlet, NodeResourcesServlet, MetricQueryServlet, MetricsWriterServlet,

CPU Load Components

The following components collect metrics about the CPU consumption of Agents. The components are in core.jar and should be loaded into the NodeAgent.

`org.cougaar.core.thread.AgentLoadSensorPlugin`

> Sensor for measuring the CPU load for Agents.

`org.cougaar.core.thread.AgentLoadRatePlugin`

> Converts the raw CPU sensor measurements into Metrics

`org.cougaar.core.thread.TopPlugin`

> servlet for viewing running threads (TopServlet). Also loads the RogueThreadDetector.

Message Load Components

The following component collect metrics about the message traffic into and out of Agents. The components are in the mtsstd.jar and should be load at the insertion point Node.AgentManager.Agent.MessageTransport.Component

`org.cougaar.core.qos.metrics.AgentStatusAspect`

> Sensor for Measuring the message flow into and out of Agents (This Aspect is always loaded in the Base template)

`org.cougaar.mts.std.StatisticsAspect`

> Sensor for measuring the size of messages

`org.cougaar.core.qos.metrics.AgentStatusRatePlugin`

> Converts the raw message sensor measurements into Metrics

`org.cougaar.mts.std.StatisticsPlugin`

Servlet for viewing raw message sensor counters, which include: StatisticsServlet, AgentRemoteStatusServlet, AgentLocalStatusServlet

Agent Mobility and QuO Components

The following Components pull the topology service to detect when Nodes and Agents move. Also, they offer a service for QuO Sysconds to subscribe to Metrics. They should be loaded in the insertion point Node.AgentManager.Agent.MetricsServices.Component

`org.cougaar.core.qos.rss.AgentHostUpdaterComponent`

Periodically polls the topology service and updates the internal Metric Service models to keep the Host-Node-Agent containment hierarchy up to date.

`org.cougaar.lib.mquo.SyscondFactory`

Factory for creating QuO Sysconds which track Metrics.

Persistence Size Components

The follow component measure the memory consumption of Agents. These components are in the core.jar and need to be loaded into every Agent.

`org.cougaar.core.qos.metrics.PersistenceAdapterPlugin"`

Sensor for measuring the Agent persistence size.

Gossip Components

The gossip subsystem disseminates metrics between nodes. (See 3.2.12.10). The components are in qos.jar

`org.cougaar.core.qos.gossip.GossipAspect`

Piggybacks Metric requests and Metric Values on messages. Load at the insertion point Node.AgentManager.Agent.MessageTransport.Component

`org.cougaar.core.qos.gossip.SimpleGossipQualifierComponent`

Limits which Metrics should be requested or value forwarded Load at the insertion point Node.AgentManager.Agent.MessageTransport.Component

`org.cougaar.core.qos.gossip.GossipFeedComponent`

Metrics Service Feed for updating metrics from remote Nodes. Load at the insertion point Node.AgentManager.Agent.MetricService.Component

`org.cougaar.core.qos.gossip.GossipStatisticsServiceAspect`

Collects gossip overhead statistics Load at the insertion point Node.AgentManager.Agent.MessageTransport.Component `org.cougaar.core.qos.gossip.GossipStatisticsPlugin`

> Load Servlet for viewing Gossip overhead statistics. Load at the insertion point Node.AgentManager.Agent.MessageTransport.Component

RSS-Resource Status Service

The Metric Service need access to configuration files to define the expected network and host capacity. The Metrics-Sites.conf file is required and the Metrics-Defaults.conf file is optional. Example files are in the overlay at .../configs/rss or source in in qos/configs/rss. Also, the data feed with the name "sites" has the special purpose of defining the sites themselves (i.e.subnet masks), so other Metrics Keys should not be put in this conf file.

The cougaarconfig: url scheme means the files are on the cougaar config path. Otherwise, the url is normal and will just down load the conf files. Putting the configuration files on a web server is useful for a cougaar applications which run at a site with a complicated topology. Also, Network and Host management systems can update the files with real data. (Note the conf files are down-loaded only once at startup)

`org.cougaar.core.qos.rss.ConfigFinderDataFeedComponent`

Two Components should be loaded with the following parameters

> `"name=sites","url=cougaarconfig:Metrics-Sites.conf"`
>
> `"name=hosts","url=cougaarconfig:Metrics-Defaults.conf"` The insertion point is Node.AgentManager.Agent.MetricsServices.Component

Acme Scripts

Acme rule scripts are available for loading the Metrics Service. The rules are in directory `qos/csmart/config/rules/metrics`. The `rule.txt` list the recommended rules to load the standard Metrics Service. Addition rules must be added in order to measure the length of messages (serialization) and setup the network/host configuration files (rss). The rule directories have the following purpose:

`sensors`

> Adds servlets that look at the raw sensors. These rules do not load any of the Metrics Service runtime, so they should not impact performance. These rules are useful for debugging and we recommended that they should alway be loaded.

`basic`

> Loads the basic Metrics Service which includes some potentially high overhead components, such as the Agent-to-Agent traffic matrix and Gossip.

`serialization`

> The basic MTS serialization rule for measures message length. This rule conflicts with other serialization rules, such as bandwidth measurement, security, and compression. These other aspects must be loaded in a specific order which is explained in the `metrics-only-serialization.rule` Only one serialization rule should be loaded.

`rss`

Supplying network and host configuration data to the RSS is very network specific. Each test site has its own rules which tap into the local sources of system information. If you are running at the TIC, you should load the rules in `rss/tic`

3.2.12.10 Using Metrics Servlets

Cougaar includes optional servlet components to display metrics data in a browser. These servlets can be used to:

- View the real-time performance of the society, such as CPU load and message traffic,
- Debug a running application, for example to find blocked messages, and
- Illustrate more complex examples of the basic metrics service (see section 3.2.12.3).

Servlets

The servlets are documented in more detail below. Here is a brief summary of the servlets, in order of most common use by developers:

`/metrics/agent/load`

Current top-level view of the CPU load, message traffic, and persistence activity of all agents on the node.

`/metrics/host/resources`

Current node resource usage (load average, sockets, heap size, etc)

`/metrics/remote/agents`

Current message traffic to/from all agents on the node, including the most recent communication time and the number of queued messages.

`/threads/top`

Although not a metrics servlet per-se, the thread service's "top" servlet displays running and queued pooled threads.

`/message/statistics`

Cumulative message traffic by the node (aggregate of all agents on that node), plus a message size histogram.

`/message/between-Any-agent-and-Local-Agent?agent=AGENT`

Cumulative message traffic by the specified local agent to/from any target.

`/message/between-Node-and-Agent?agent=AGENT`

Cumulative message traffic by the node (aggregate of all agents on that node) to/from the specified target agent.

`/metrics/query?paths=PATHS:`

Read raw metric(s) as XML.

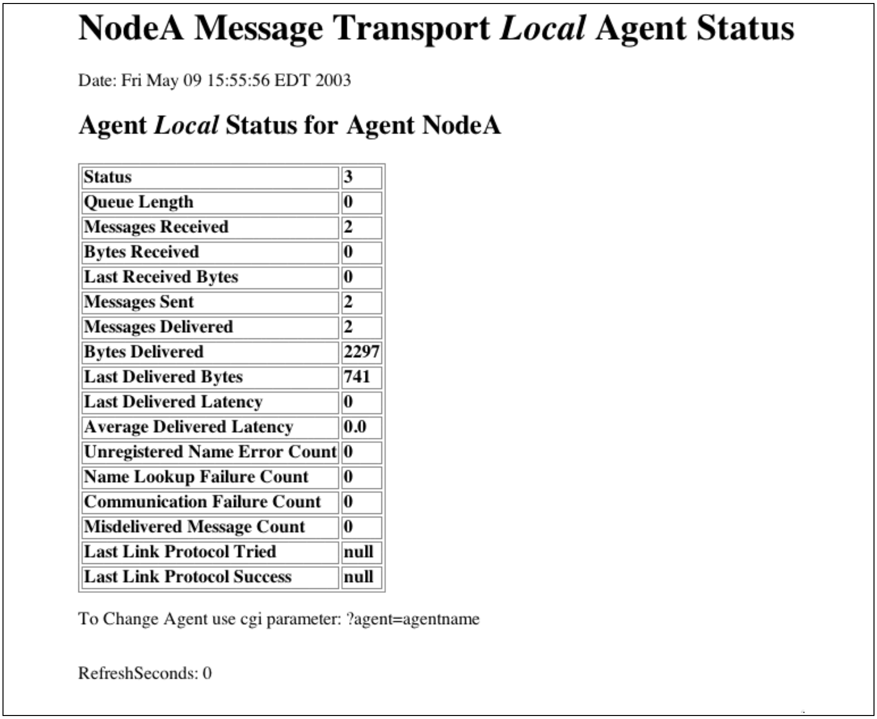
`/metrics/writer?key=KEY&value=VALUE:`

Write a raw metric into the metrics service.

The rest of this section describes each servlet in more detail.

`/message/between-Any-agent-and-Local-Agent`

This servlet displays status of communications from an agent on this node to ALL other agents in the society. The servlet is a dump of the RAW contents of the Agent Status Service and does not use the Metrics Service.



NodeA Message Transport *Local* Agent Status

Date: Fri May 09 15:55:56 EDT 2003

Agent *Local* Status for Agent NodeA

Status	**3**
Queue Length	**0**
Messages Received	**2**
Bytes Received	**0**
Last Received Bytes	**0**
Messages Sent	**2**
Messages Delivered	**2**
Bytes Delivered	**2297**
Last Delivered Bytes	**741**
Last Delivered Latency	**0**
Average Delivered Latency	**0.0**
Unregistered Name Error Count	**0**
Name Lookup Failure Count	**0**
Communication Failure Count	**0**
Misdelivered Message Count	**0**
Last Link Protocol Tried	**null**
Last Link Protocol Success	**null**

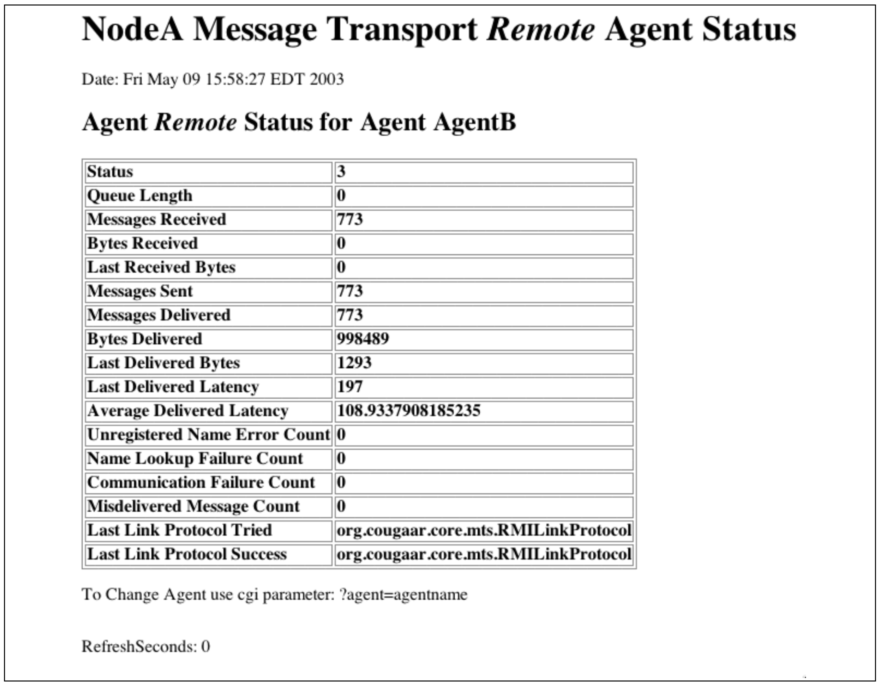
To Change Agent use cgi parameter: ?agent=agentname

RefreshSeconds: 0

Figure 3-15:Local Agent Messages Servlet

`/message/between-Node-and-Agent`

This servlet displays status of communications from ALL agents on this node to a specified agent. The servlet is a dump of the RAW contents of the Agent Status Service and does not use the Metrics Service.



NodeA Message Transport *Remote* Agent Status

Date: Fri May 09 15:58:27 EDT 2003

Agent *Remote* Status for Agent AgentB

Status	**3**
Queue Length	**0**
Messages Received	**773**
Bytes Received	**0**
Last Received Bytes	**0**
Messages Sent	**773**
Messages Delivered	**773**
Bytes Delivered	**998489**
Last Delivered Bytes	**1293**
Last Delivered Latency	**197**
Average Delivered Latency	**108.9337908185235**
Unregistered Name Error Count	**0**
Name Lookup Failure Count	**0**
Communication Failure Count	**0**
Misdelivered Message Count	**0**
Last Link Protocol Tried	**org.cougaar.core.mts.RMILinkProtocol**
Last Link Protocol Success	**org.cougaar.core.mts.RMILinkProtocol**

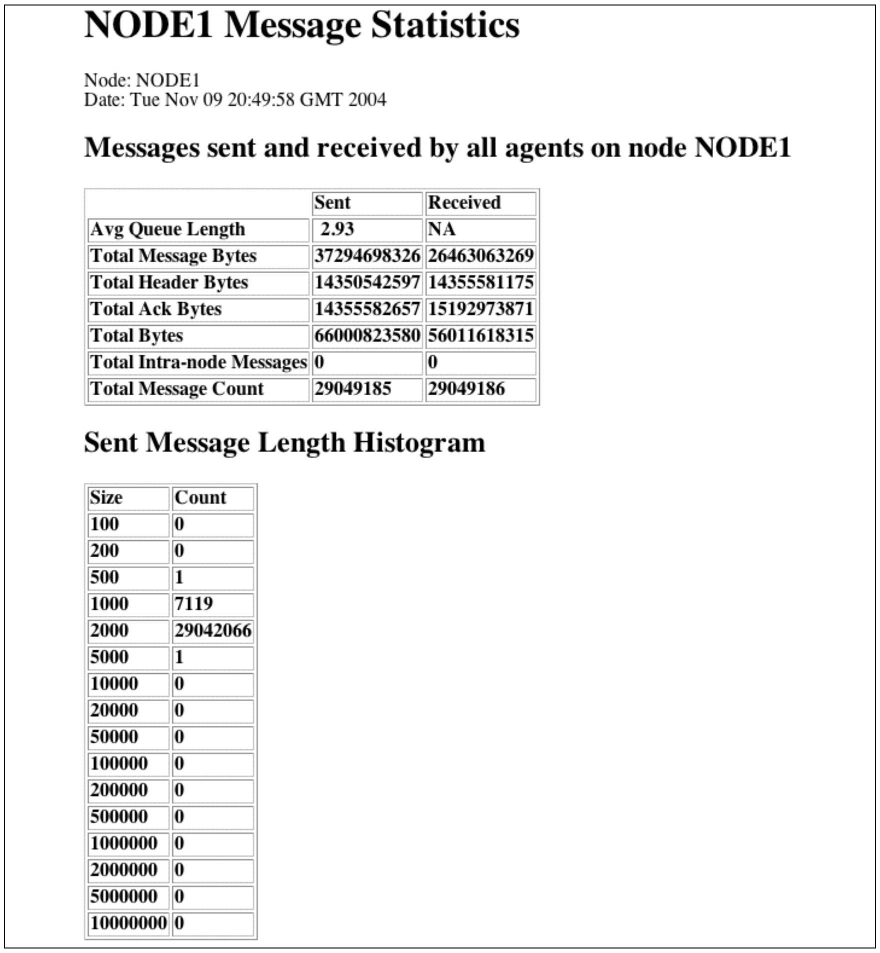
To Change Agent use cgi parameter: ?agent=agentname

RefreshSeconds: 0

Figure 3-16: Remote Agent Messages Servlet

`/message/statistics`

This servlet summarizes the Messages statistics for communications out of all agents on this node. This is a raw dump of the legacy MTS Message Statistics Service and does not use the Metrics service.



NODE1 Message Statistics

Node: NODE1
Date: Tue Nov 09 20:49:58 GMT 2004

Messages sent and received by all agents on node NODE1

	Sent	Received
Avg Queue Length	2.93	NA
Total Message Bytes	37294698326	26463063269
Total Header Bytes	14350542597	14355581175
Total Ack Bytes	14355582657	15192973871
Total Bytes	66000823580	56011618315
Total Intra-node Messages	0	0
Total Message Count	29049185	29049186

Sent Message Length Histogram

Size	Count
100	0
200	0
500	1
1000	7119
2000	29042066
5000	1
10000	0
20000	0
50000	0
100000	0
200000	0
500000	0
1000000	0
2000000	0
5000000	0
10000000	0

Figure 3-17: Message Statistics Servlet

`AvgQueueLength`

Average of the number of messages waiting on all Destination Links. The average is calculated using a decaying average

`Total Bytes`

Count of the number of bytes set from all agents on this node. This statistic does not include inter-node messages, which are not serialized

`Total Count`

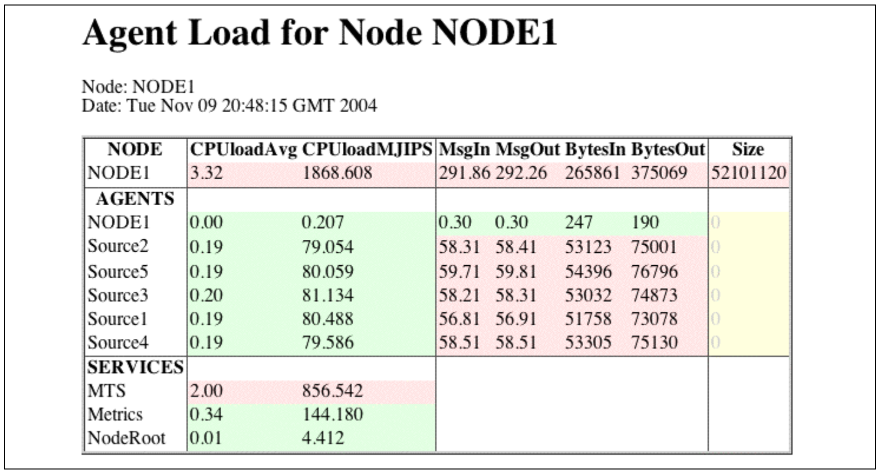
Count of Messages sent from all agents on this node. This count includes both inter-node messages and intra-node messages

`Message Length Histogram`

Count of Messages sent from all agents on this node, organized by size. The bins are labeled by the max size in the bin

`/metrics/agent/load`

This servlet shows the amount of resource consumption for each agent and service that is resident to this node. The resources include CPU, Communications, and Storage. This servlet is used to see which agents are resident on the node and their level of activity. The metrics are all average rates over the averaging interval.



Agent Load for Node NODE1

Node: NODE1
Date: Tue Nov 09 20:48:15 GMT 2004

NODE	CPUloadAvg	CPUloadMJIPS	MsgIn	MsgOut	BytesIn	BytesOut	Size
NODE1	3.32	1868.608	291.86	292.26	265861	375069	52101120
AGENTS							
NODE1	0.00	0.207	0.30	0.30	247	190	0
Source2	0.19	79.054	58.31	58.41	53123	75001	0
Source5	0.19	80.059	59.71	59.81	54396	76796	0
Source3	0.20	81.134	58.21	58.31	53032	74873	0
Source1	0.19	80.488	56.81	56.91	51758	73078	0
Source4	0.19	79.586	58.51	58.51	53305	75130	0
SERVICES							
MTS	2.00	856.542					
Metrics	0.34	144.180					
NodeRoot	0.01	4.412					

Figure 3-18:Agent Load Servlet

`CPULoadAvg`

Average number of thread servicing the agent. For example, a load average of 2, means that two threads were servicing the agent for the whole averaging interval. This is unlikely and is a indicator that one of the Agent's plugins is inappropriately holding onto a thread during a sleep, or wait for I/O

`CPULoadMJIPS`

Average rate of CPU used servicing the agent. This is the integration load average * the effective cpu.

`MsgIn`

Message Per Second from all agent to this agent

`MsgOut`

Message Per Second from this agent to all other agents

`BytesIn`

Bytes Per Second from all agent to this agent

`BytesOut`

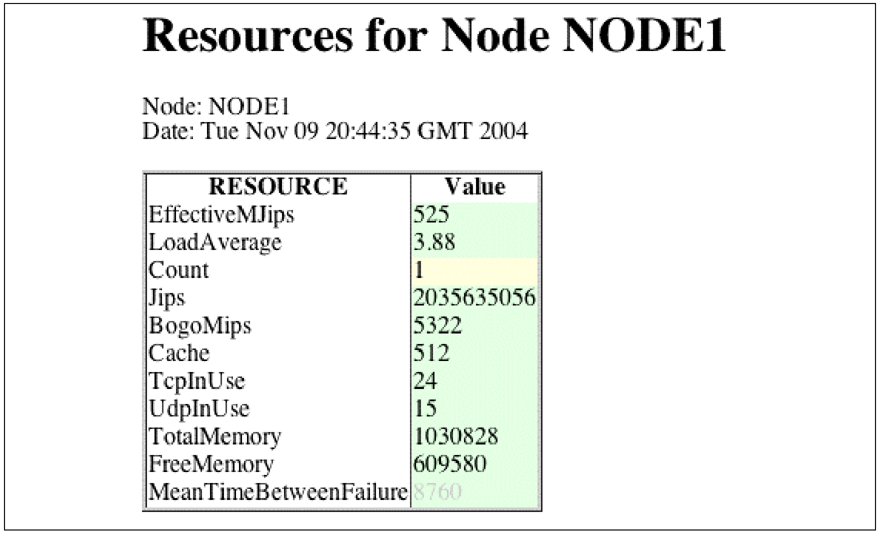
Bytes Per Second from this agent to all agent

`PersistSize`

Size of the last Agent Persist

`/metrics/host/resources`

This servlet shows the status of Host resources for the Node. The most of basic values come from polling Linux /proc. MJIPS (Million Java Instructions per Second) comes from running benchmark.



Resources for Node NODE1

Node: NODE1
Date: Tue Nov 09 20:44:35 GMT 2004

RESOURCE	Value
EffectiveMJips	525
LoadAverage	3.88
Count	1
Jips	2035635056
BogoMips	5322
Cache	512
TcpInUse	24
UdpInUse	15
TotalMemory	1030828
FreeMemory	609580
MeanTimeBetweenFailure	8760

Figure 3-19:Node Resources Servlet

`EffectiveMJips`

Effective CPU that one thread will experience. This is a model of the Linux CPU scheduling algorithm. The model takes into account the number of CPUs, MJIPS for each CPUs, and the load average of the host.

`Load Average`

Host Load Average is the number of processes running in Host.

`Count`

Number of Processors

`TCP In Use`

The number of TCP Sockets

`UDP In Use`

The number of UDP Sockets

`Total Memory`

Host Memory

`Free Memory`

`Cache`

Size of a CPU's level 2 cache

`/metrics/query`

This servlet allows the operator to query the MetricsService directly. The result can either be displayed as a web page as XML or returned to the invoker as serialized Java HashMap, depending on the value of the format uri argument. One or more query paths (see 3.2.12.6) should be supplied as the value of the paths uri argument, with | as the separator.

Usage:

```
"http://localhost:8800/$nodename/metrics/query?format=xml&paths=A
gent(3-69-ARBN):Jips|Agent(3-69-ARBN):CPULoadJips10SecAvg"
```

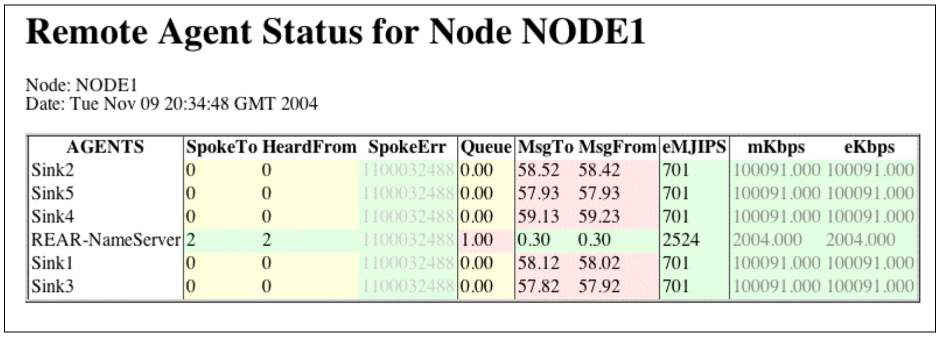
The 'format' argument is optional, but if left out defaults to xml return of metric data to the browser.

An optional Java version of a metrics query client was written and resides in core/examples/org/cougaar/core/examples/metrics/ExampleMetricQueryClient, returning a hashmap of path values from the query-specified node. This code is shown below:

```
/*
 * Stand alone java client that opens up a URL connection, reads an
object stream and
 * outputs the result (will be a java ArrayList of 'Path|Metric's )
 * Usage: java -cp . ExampleMetricQueryClient "http://localhost:8800/\$3-
69-ARBN/metrics/query?format=java&paths=Agent(3-69-
ARBN):SpokeTime|IpFlow(blatz,stout):CapacityMax"
 * Must specify 'format=java' as the default is a string of xml printed
out to browser
 */
package org.cougaar.core.examples.metrics;

import java.net.URL;
import java.io.IOException;
import java.io.InputStream;
import java.io.ObjectInputStream;
import java.io.ByteArrayInputStream;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

// This example requires quoSumo.jar, qos.jar and core.jar
public class ExampleMetricQueryClient
{
  // One arg here - the query string "http://host..." see usage above
  public static void main( String args[] ) throws IOException
  {
    String url = null;
    try {
      url=args[0];
      URL servlet_url = new URL(url);

      // open up input stream and read object
      try {
 // open url connection from string url
 InputStream in = servlet_url.openStream();
 ObjectInputStream ois = new ObjectInputStream(in);

 // read in java object - this a java client only
 HashMap propertylist = (HashMap)ois.readObject();
 ois.close();

 if (propertylist == null) {
     System.out.println("Null Property List returned");
     return;
 }
 // can do anything with it here, we just print it out for now
 Iterator itr = propertylist.entrySet().iterator();
 while (itr.hasNext()) {
     Map.Entry entry = (Map.Entry) itr.next();
     System.out.println(entry.getKey() +"->"+ entry.getValue());
 }
      } catch (Exception e) {
 System.out.println("Error reading input stream for url " + url + " Make
sure the stream is open. " + e);
 e.printStackTrace();
      }
    } catch(Exception e){
      System.out.println("Unable to acquire URL, Exception: " + e);
    }
  }
}
```

`/metrics/remote/agents`

This servlet shows the status of resources along the path for communications from any agent on the node to a specific agent. This servlet is useful for debugging. For example, if the Queue length is greater than one, then messages are backed up waiting to be transmitted to the agent. Since messages are usually sent right away this indicates a problem along the path. Likewise, if he Node has not HeardFrom an agent recently, the agent or its node may have failed. Also, the table show the capacity of the network path to the agent and the agent's host capacity.



Remote Agent Status for Node NODE1

Node: NODE1
Date: Tue Nov 09 20:34:48 GMT 2004

AGENTS	SpokeTo	HeardFrom	SpokeErr	Queue	MsgTo	MsgFrom	eMJIPS	mKbps	eKbps
Sink2	0	0	1100032488	0.00	58.52	58.42	701	100091.000	100091.000
Sink5	0	0	1100032488	0.00	57.93	57.93	701	100091.000	100091.000
Sink4	0	0	1100032488	0.00	59.13	59.23	701	100091.000	100091.000
REAR-NameServer	2	2	1100032488	1.00	0.30	0.30	2524	2004.000	2004.000
Sink1	0	0	1100032488	0.00	58.12	58.02	701	100091.000	100091.000
Sink3	0	0	1100032488	0.00	57.82	57.92	701	100091.000	100091.000

Figure 3-20: Remote Agent Status Servlet

`Spoke To`

number of seconds since this node has spoke to this agent.

`Heard From`

number of seconds since something has heard from this agent (includes other node via gossip)

`Spoke Error`

number of seconds since the last communication error

`Queue`

instantaneous queue length for messages waiting to be sent to agent. (Includes messages in the process of being sent)

`MsgTo`

message per second from all agents on this node to agent

`MsgFrom`

messages per second from agent to any agent on this node

`eMJIP`

effective Million Java Instruction Per Second that a single thread on the agent host.

`mKbps`

maximum kilobits per second for the network path between this node and the agent

`eKbps`

expected kilobits per second for the network path between this node and the agent

`/metrics/writer`

This servlet allows the operator to write values into the MetricsService directly. The key and value should be supplied with uri arguments of the same name. For now the value must be parseable as a double. The metric will be entered with USER_DEFAULT_CREDIBILITY (0.3) and with the client host as the provenance.

Usage:

Specified by a prefix of protocol, host, port, nodename and path, followed by some key-value pair in the usual http::get parameter format.

e.g.:

```
"http://localhost:8800/$nodename/metrics/writer/?key=Site_Flow_10
.200.2.0/24_10.200.4.0/16_Capacity_Max&value=5600"
```

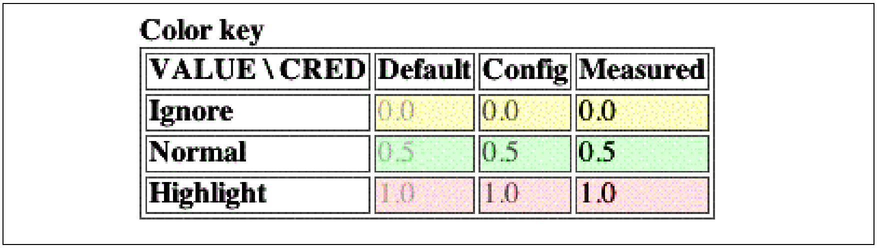
Servlets' GUI Conventions

Several metrics servlets uses gui conventions to show three attributes of each metric. In addition, some are mouse sensitive: all the metric's attributes are displayed on the browser's documentation line.

The *value* of the metric is displayed as the base text.

The *credibility* of the metric is displayed as the color of the text: light gray to indicate that the metric value was only determined as a compile-time default; gray to indicate that the metric value was obtained from a configuration file; black to indicate that the metric value was obtained from a run-time measurement. A metric's credibility metric is an approximate measure of how much to believe that the value is true. Credibility takes into account several factors, including when, how, by whom a measurement was made.

When the value of a metric crosses a *threshold*, its value may be interesting enough to warrant attention. This is shown by the color of the background. A green background indicates that the value is in the normal range. A yellow background indicates that the values is in a typical ground state, i.e nothing is happening. A red background indicates that the value has crossed a metric-specific threshold and may be interesting.



Color key

VALUE \ CRED	Default	Config	Measured
Ignore	0.0	0.0	0.0
Normal	0.5	0.5	0.5
Highlight	1.0	1.0	1.0

Figure 3-21: Color Key

3.2.12.11 Engineering Assessment (Null Metrics Service)

For engineering assessment, it is possible to configure the metric service to add or remove functionality. The assessment idea is to remove as much functionality as possible to obtain a base line. The functionality is then added back piece by piece and the system is re-evalutate to determine the overhead.
The first level of testing is to remove the metrics service sensors and clients. To remove the clients of the service, the metrics rules should not be loaded into the society XML. Past measurements show that the metric service clients and less than 10% load on a Node.

Another level of base-lining is to remove the metric-service implementation itself. The Metric Service is really a wrapper around the QuO Resource Status Service. If the RSS class is not found, the wrapper will act as a blackhole. The `MetricsService` will accept subscriptions, queries, but the wrapper will always return a NO-VALUE Metric. Also, `MetricsUpdaterService` can input with key-value pairs, but the wrapper will just drop them.
The empty implementation of `MetricsService` returns a 0-credibility metric with the provenance set to "undefined" (this constant `MetricImpl.UndefinedMetric`) for all `getValue` calls and does nothing for all `subscribeToValue` and `unsubscribeToValue` calls. The empty implementation of `MetricsUpdateService` does nothing for the `updateValue` call.
Of course none of the Metrics clients will work, and should not be configured in the society XML.

In 11.0 you can disable the QuO-RSS implementation, getting empty implementations of the MetricsService and the MetricsUpdateService, if qos.jar isn't accessible. Simply remove the quoSumo.jar from the Class path `$CIP/sys/qouSumo.jar`.
In subsequent releases you get empty implementations of the MetricsService and the MetricsUpdateService by loading the component `org.cougaar.core.qos.metrics.MetricsServiceProvider`.
Full Component-loading isn't functional in 11.0, so provided is a workaround to do this with a -D flag:

`-Dorg.cougaar.society.xsl.param.metrics=trivial`.

3.2.13 Alarm Service

This service is an Agent level service that is designed to allow Plugins to access the current time of the system as well as set Alarms to be awoken at a specific system time or after a certain amount of real time. This is useful for Plugins that wish to be executed at a certain time regardless of subscription (blackboard) changes.

```
package org.cougaar.core.agent;
import org.cougaar.core.component.Service;

/** A Service for registering Alarms **/
public interface AlarmService extends Service {
  long currentTimeMillis();
  void addAlarm(Alarm alarm);
  void addRealTimeAlarm(Alarm alarm);
}
```

Related to the AlarmService, the DemoControlServiceis an Agent level service that provides users the ability to set the system time and advance the system time as well as setting specific time rates. This facility should be used for Cougaar demonstration purposes such as Cougaar execution capability demonstrations.

```
package org.cougaar.core.agent;

import org.cougaar.core.component.Service;

public interface DemoControlService extends Service {

  // get the current execution rate, usually 1.0
  double getExecutionRate();

  // set the local agent's execution time.
  // the "changeTime" is when it takes effect.
  void setLocalTime(long offset, double rate, long changeTime);

  // set the time at all the targets (nodes).
  // the optional callback is invoked to indicate progress and completion.
  boolean setSocietyTime(
    long offset, double rate, long changeTime,
    Set targets, Callback cb);

  // same as above, but use the naming service to list all nodes.
  boolean setSocietyTime(
    long offset, double rate, long changeTime,
    Callback cb);

  .. lots of backwards-compatible variations on the above methods ..

  // progress callback
  public interface Callback {
    void sendingTimeAdvanceTo(Set addrs); // starting
    void updatedTime(MessageAddress addr, long rtt);
    void completed(Map addrsToRTT);  // finished
 }
}
```

3.2.14 SchedulerService

This service provides Plugins with a way to register with a scheduling service to be awoken under specific circumstances. This service can be used with the BlackboardService and the AlarmService. The new `BlackboardClientComponent` base class utilizes this service.

```
package org.cougaar.core.agent;
import org.cougaar.core.component.Service;
import org.cougaar.core.component.Trigger;

/** Schedules plugins. Tells them by calling a registered callback */

public interface SchedulerService extends Service {

  /**
   * Tells Scheduler to handle scheduling this object
   * @param managedItem the trigger to be pulled to run the object
   * @return a handle that the caller can use to tell the scheduler that it
wants to be run.
   **/
  Trigger register(Trigger managedItem);

  /** @param removeMe Stop scheduling this item **/
  void unregister(Trigger removeMe);
}
```

As of Cougaar 11.0, the SchedulerService is a thin wrapper around the standard ThreadService, and may be removed in a future Cougaar release.

3.2.15 Thread Services

3.2.15.1 Overview

The COUGAAR Thread Services are designed to replace the direct use of Java Threads with a threading system that can be more tightly controlled. The central new interface is Schedulable (see org.cougaar.core.thread.Schedulable), which can be thought of as taking the place of a Java native thread. Schedulables are created and returned by the ThreadService and started by client code. Unlike native Java threads, Schedulables don't necessarily run immediately. The Threading Services allow only a certain number of threads to be running at once. If a Schedulable is started when all threads slots are in use, it will be queued and will only run when enough running threads have stopped to allow it reach the head of the queue. The maximum number of running Schedulables as well as the queue ordering can be controlled by the ThreadControlService (see org.cougaar.core.service.ThreadControlService). The Thread Services can also be used to schedule Java TimerTasks.

In addition to the running of threads and tasks, the COUGAAR Thread Services offers two other features: an event-like mechanism for receiving callbacks when 'interesting' events occur (ThreadListenerService; see: org.cougaar.core.service.ThreadListenerService), and a certain amount of explicit control over scheduling (ThreadControlService; see: org.cougaar.core.service.ThreadControlService). These interfaces, as well as the Schedulable and ThreadService interfaces, are described in detail below.

The COUGAAR Thread Services are hierchical. Each Agent has a set of Thread Services of its own, as does the MTS and certain other Node-level components. These local Thread Services handle threading at their own level and are in turn controlled by a root Thread Service at the Node level. The inter-level control mechanism takes the form of "rights" given by the higher to the lower level services, and returned after use by the lower to the higher level services. When the higher level service has multiple children, a RightsSelector is used to choose the child service that will be the next to receive rights (round-robin by default). In principle this hierarchy could be extended to a further depth but so far we haven't found any good reason to do that in practice.

Within a level, scheduling is further subdivided into lanes. There are currently four hardwired lanes. A lane is associated with a Schedulable at creation time and remains fixed for the lifetime of the Schedulable.

The design of the thread services has important implications for the runtime pattern of the Runnables it will be running. In particular, it's not generally a good idea for these Runnables to block. The common Java pattern of a loop with a wait or sleep call should usually be unwrapped into a simple 'strip' of code that reschedules itself if it needs to run again. Examples can be seen in 3.2.15.3 Use Cases and Examples. Code which needs to block or to use an unusual amount of resources should run in the appropriate lane if at all possible.

3.2.15.2 Design and Architecture

The overall structure of the thread service system is a tree that tends to mimic the tree of Containers in COUGAAR. Each level of this tree includes its own trio of thread services which are directly responsible for the Threads used at that level of the hierarchy and indirectly responsible for its children.

Each level other than the root requests run-rights from its parent and only runs as many java Threads as it has rights. With the default scheduler, all requests for run-rights ultimately propogate to the root service, which keeps a count of running threads and refuses to allocate further rights if that count hits a given maximum. As each Thread finishes, its run-right is released, again propagating to the root service. The released right will then be made available to the children, using a layer-specific algorithm. Each layer can (a) consume the right itself (if the given layer has queued threads); (b) recursively give it to a child; or (c) decline to accept it.

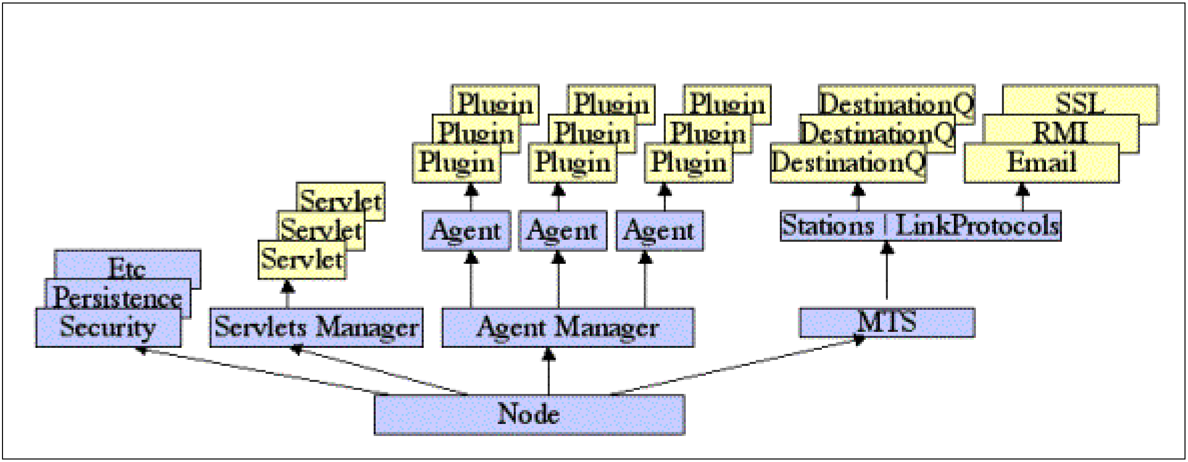


Figure 3-22: Thread Service Design

The direct control of local threads at any given level is handled by a per-lane Scheduler. If a thread at that level wants to run, the Scheduler for its lane is asked for a right. If a right is available a true java Thread will be run; if not, the request will be queued until sufficient rights are available. The order in which items are removed from the queue depends on a Comparator, which by default uses time (i.e., fifo) but which at any time can be replaced by an arbitrarily complex and dynamic Comparator via the ThreadControlService. Schedulers use a RightsSelector to determine the possible re-allocation of a released run-right. The default RightsSelector uses a round-robin algorithm, which provides fair scheduling between the layer's own queued requests and its children. The RightsSelector can be replaced at any time via the ThreadControlService.

The hierarchy skeleton is represented by a set of TreeNodes. The TreeNode at any given level holds pointers to the level's scheduler, selector, parent node and child nodes.

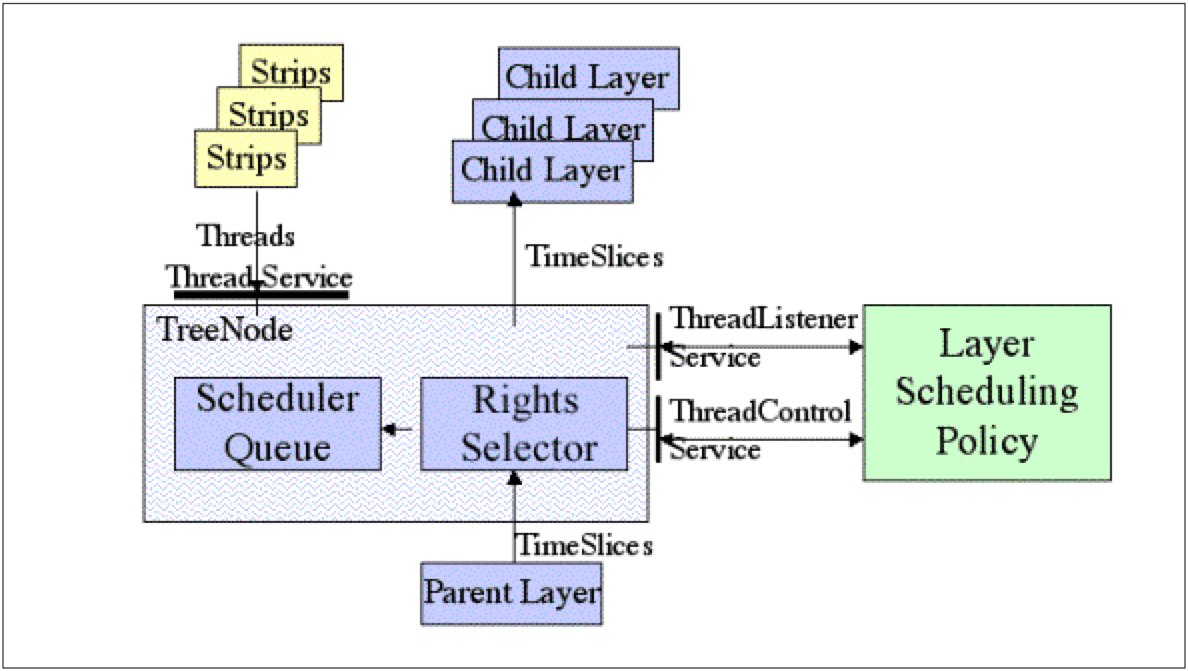


Figure 3-23: Thread Services Layer Architecture

Any Container whose components wish to use the thread services should provide those services locally, using ThreadServiceProvider. This will ensure that the thread service hierarchy maps properly into the Component hierarchy.

Core Internal Classes

ThreadPool

> This is an implementation of a classic thread-pool. The threads it provides are defined by an inner class, PooledThread.

SchedulableObject

> This is the implementation of Schedulable. It communicates with a Scheduler when it wants to request or return rights and with the ThreadPool when it needs a Thread.

Scheduler

This is the simplest scheduler class. It deals with run-rights locally, completely ignoring the hierarchy. It's also the implementation of the ThreadControlService.

PropagatingScheduler

This is an extension of Scheduler that provides the standard hierarchical behavior, as described above.

ThreadListenerProxy

This is the implementation of ThreadListenerService.

ThreadServiceProxy

This is the implementation of ThreadService. It creates the Thread TreeNode and the thread pool and links the pool to a Scheduler.

Interaction Among Internal Classes

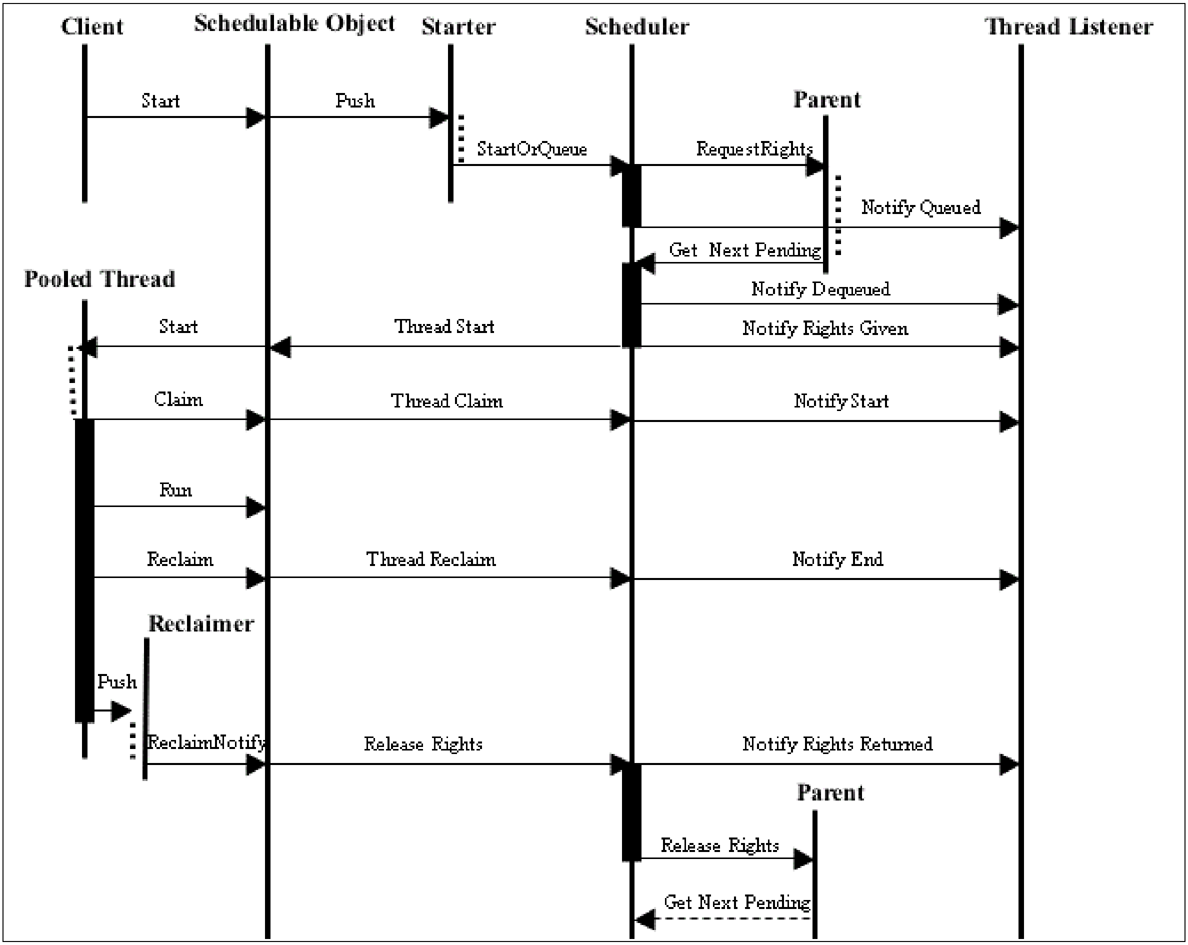


Figure 3-24: Thread Lifecycle Execution Graph

3.2.15.3 Use Cases and Examples

Code Strips : avoiding calls to wait()

As explained in the overview, Schedulables run more effectively as strips of code than as loops with a wait, since the blocking behavior of the wait will tie up a limited resource needlessly.

As an example, consider a thread-based queue pattern, which might look something the following.

```
class QueueRunner implements Runnable
{
    private SimpleQueue queue;
    private Thread thread;

    QueueRunner() {
     queue = new SimpleQueue(); // make the internal queue
     thread = new Thread(this, "My Queue");
     thread.start();  // start the thread
    }

    public void run() {
     Object next = null;
     while (true) {
         while (true) {
           synchronized (queue) {
               if (!queue.isEmpty()) {
                 next = queue.pop();
                 break; // process the next item
               }
               // queue is empty, wait to be notified of new items
               try { queue.wait(); }
               catch (InterruptedException ex) {}
           }
         }
         processNext(next);
     }
    }

    public void add(Object x) {
     synchronized (queue) {
         queue.add(x);
         queue.notify(); // wake up the wait()
     }
    }

}
```

This thread will spend most of its time in the wait call, which is not at all an effective use of a limited resource. As a Schedulable it would be better written as follows:

```
class QueueRunner implements Runnable
{
    private SimpleQueue queue;
    private Schedulable schedulable;

    QueueRunner() {
     queue = new SimpleQueue();
     // Create the Schedulable but don't start it yet.
     schedulable = threadService.getThread(this, this, "MyQueue");
    }

    public void run() {
     // Handle all items currently queued but never block
     Object next = null;
     while (true) {
         synchronized (queue) {
            if (queue.isEmpty()) break; // done for now
            next = queue.pop();
         }

         processNext(next);
     }
    }

    void add(Object next) {
     synchronized (queue) {
         queue.add(next);
     }
     // Restart the schedulable if it's not currently running.
     schedulable.start();
    }
}
```

Code strips: avoiding calls to sleep()

Another popular Java Thread pattern uses a sleep in a never-ending loop. As above, this uses up a thread resource even though the thread is rarely running.

```
class SleepingPoller implements Runnable
{

    private int period;
    private String name;

    SleepingPoller(String name, Object client, int period) {
     this.name = name;
     this.period = period;
     ThreadService ts = (ThreadService)
         sb.getService(this, ThreadService.class, null);
     ts.getThread(client, this, name).start();
        sb.releaseService(this, ThreadService.class, ts);
    }


    void initialize() {
     // Initialization code here
    }

    void executeBody()
     throws Exception
    {
     // Thread body here.
    }

    public void run() {
     initialize();
     while (true) {
         try {
            executeBody();
         }
         catch (Exception ex) {
            log.error("Error in thread " + name, ex);
         }
         try { Thread.sleep(period); }
         catch { InterruptedExecption (ex) }
     }
    }
}
```

In this case the use of sleep can be replaced by having the run() method restart the thread after a delay.

```
class Poller implements Runnable
{

    private Schedulable schedulable;
    private boolean initialized = false;
    private int period;
    private String name;
    private ThreadService ts;

    Poller(String name, Object client, int period) {
     this.name = name;
     this.period = period;
     this.ts = (ThreadService)
         sb.getService(this, ThreadService.class, null);
     this.schedulable = ts.getThread(client, this, name);

     schedulable.start();
    }

    void ensureInitiolized() {
     synchronized (this) {
         if (initialized) return;
         initialized = true;
     }

     initialize();
    }

    void initialize() {
     // Initialization code here
    }

    void executeBody()
     throws Exception
    {
     // Thread body here.
    }

    public void run() {
     ensureInitiolized();
     try {
         executeBody();
     }
     catch (Exception ex) {
         log.error("Error in thread " + name, ex);
     }

        // Restart after period ms
     schedulable.schedule(period);
    }
}
```

Code strips: Avoiding TimerTasks

TimerTasks are not controllable and should generally be avoided. Instead use the schedule methods on Schedulable for equivalent functionality. In this case the body of the task will run as a COUGAAR thread. Compare runTask() and runThreadPeriodically() below.

```
class MyPeriodicCode
{
    private int period;
    private Schedulable schedulable;

    public void body() {
     // The body of code to be run periodically goes here.
    }

    // In this version body() runs in the Thread Service's Timer
    // thread, which can be problematic if it takes too long.
    void runTask() {
     ThreadService ts = sb.getService(this, ThreadService.class, null);

     TimerTask task = new TimerTask() {
            public void run() {
                body();
            }
         };
     ts.schedule(task, 0, period);

     sb.releaseService(this, ThreadService.class, ts);
   }


    // In this version body() runs periodically as a cougaar thread.
    void runThreadPeriodically() {
     ThreadService ts = sb.getService(this, ThreadService.class, null);

     Runnable code = new Runnable () {
            public void run() {
                body();
            }
         };
     schedulable = ts.getThread(this, code, "MyPeriodicCode");

        // Restart the Schedulable every period ms
        schedulable.schedule(0, period);

     sb.releaseService(this, ThreadService.class, ts);
    }

}
```

3.2.15.4 APIs

org.cougaar.core.thread.Schedulable

`void start()`

Starting a `Schedulable` is conceptually the same as starting a Java native thread, but the behavior is slightly different in two ways. First, if no thread resources are available, the `Schedulable` will be queued instead of running righy away. It will only run when enough resources have become available for it to reach the head of the queue. Second, if the `Schedulable` is running at the time of the call, this call will cause the `Schedulable` restart itself after the current run finishes (unless it's cancelled in the meantime). If the Schedulable is pending at the time of the call, this method is a no-op.

`void schedule(long delay)`

Equivalent to calling `start()` after the given delay (in millis).

`void schedule(long delay, long period)`

Runs a `Schedulable` periodically at the period given, where each run is equivalent to `start()` (ie it may not start immediately). The delay applies to the first run.

`void scheduleAtFixedRate(long delay, long period)`

This is to the method above as the method by the same name on `TimerTask` is to the `schedule` method on that class (see Sun's TimerTask javadoc at http://java.sun.com/j2se/1.4.1/docs/api/java/util/TimerTask.html).

`void cancelTimer()`

This method cancels the active periodic schedule. It does not cancel the `Schedulable` entirely (see `cancel()`).

`boolean cancel()`

Cancelling a `Schedulable` will prevent it starting if it's currently queued or from restarting if it was scheduled to do so. It will not cancel the current run. This method also runs `cancelTimer`.

`int getState()`

Returns the current state of the `Schedulable`. The states are defined in org.cougaar.core.thread.CougaarThread and consist of the following:

- THREAD_RUNNING
- THREAD_PENDING (ie queued)
- THREAD_DISQUALIFIED (see `boolean setQualifier` in org.cougaar.core.service.ThreadControlService below.)
- THREAD_DORMANT (ie none of the above)
- THREAD_SUSPENDED (not currently used)

`Object getConsumer()`

Returns the COUGAAR entity for which the `ThreadService` made this `Schedulable`. See `Schedulable getThread` in org.cougaar.core.service.ThreadService below.

`int getLane()`

Returns the lane that this `Schedulable` was assigned to. See `Schedulable getThread` in org.cougaar.core.service.ThreadService below. Lane constants can be found in the ThreadService interface.

org.cougaar.core.service.ThreadControlService

`void setDefaultLane(int lane)`

Sets the default lane for any newly created `Schedulables`. This has no effect on extant `Schedulables`.

`void setMaxRunningThreadCount(int count, int lane)`

Sets the maximum number of `Schedulables` that are allowed to run at any one time (across all levels) for the given lane.

`void setQueueComparator(Comparator comparator, int lane)`

Sets the `Comparator` used by the queue at this level for the given lane to order its elements (`Schedulables`). The 'smallest' value, as determined by the `Comparator`, is the first element of the queue. By default the queue is sorted by time (fifo).

`void setRightsSelector(RightsSelector selector, int lane)`

By default, the "right" to run is handled in a round-robin fashion. This can be overridden by providing a different `RightsSelector`. The rights-selection mechanism is experimental and lightly tested, and shouldn't be used (yet) except for experimenting.

`boolean setQualifier(UnaryPredicate predicate, int lane)`

Sets the qualifier at this level for the given lane. The qualifier is used to disqualify queued `Schedulables` temorarily. Any `Schedulable` on the queue which doesn't satisfy the predicate is removed and held in another list. Such a `Schedulable` will only be returned to the queue (and thus given the opportunity to run eventually) if the qualifier is unset (ie, set to `null`).

NB: The qualifier can only be set to a non-`null` value if it's current `null` .

`boolean setChildQualifier(UnaryPredicate predicate, int lane)`

Sets the qualifier for child schedulers at this level for the given lane. The qualifier is used to prevent children from gaining rights they might otherwise have access to. This is useful to keep one child from using up all its parents rights (for example).

NB: The qualifier can only be set to a non-`null` value if it's current `null` .

`void setMaxRunningThreadCount(int count)`

Sets the maximum number of `Schedulables` that are allowed to run at any one time (across all levels) for the default lane.

`void setQueueComparator(Comparator comparator)`

Sets the `Comparator` used by the queue at this level for the default lane to order its elements (`Schedulables`). The 'smallest' value, as determined by the `Comparator`, is the first element of the queue. By default the queue is sorted by time (fifo).

`void setRightsSelector(RightsSelector selector)`

By default, the "right" to run is handled in a round-robin fashion. This can be overridden by providing a different `RightsSelector`. The rights-selection mechanism is experimental and lightly tested, and shouldn't be used (yet) except for experimenting.

`boolean setQualifier(UnaryPredicate predicate)`

Sets the qualifier at this level for the default lane. The qualifier is used to disqualify queued `Schedulables` temorarily. Any `Schedulable` on the queue which doesn't satisfy the predicate is removed and held in another list. Such a `Schedulable` will only be returned to the queue (and thus given the opportunity to run eventually) if the qualifier is unset (ie, set to `null`).

NB: The qualifier can only be set to a non-`null` value if it's current `null` .

`boolean setChildQualifier(UnaryPredicate predicate)`

Sets the qualifier for child schedulers at this level for the default lane. The qualifier is used to prevent children from gaining rights they might otherwise have access to. This is useful to keep one child from using up all its parents rights (for example).

NB: The qualifier can only be set to a non-`null` value if it's current `null` .

`int getDefaultLane()`

Returns the current default lane.

`int runningThreadCount(int lane)`

Returns the number of `Schedulables` that are currently running at this level for the given lane.

`int pendingThreadCount(int lane)`

Returns the number of `Schedulables` that are currently queued at this level for the given lane.

`int activeThreadCount(int lane)`

Returns the number of `Schedulables` that are currently either running or queued at this level for the given lane.

`int maxRunningThreadCount(int lane)`

Returns the maximum number of `Schedulables` that are allowed to run at any one time (across all levels) for the given lane.

`int runningThreadCount()`

Returns the number of `Schedulables` that are currently running at this level for the default lane.

`int pendingThreadCount()`

Returns the number of `Schedulables` that are currently queued at this level for the default lane.

`int activeThreadCount()`

Returns the number of `Schedulables` that are currently either running or queued at this level for the default lane.

`int maxRunningThreadCount()`

Returns the maximum number of `Schedulables` that are allowed to run at any one time (across all levels) for the default lane.

org.cougaar.core.service.ThreadControlService

`void setDefaultLane(int lane)`

Sets the default lane for any newly created `Schedulables`. This has no effect on extant `Schedulables`.

`void setMaxRunningThreadCount(int count, int lane)`

Sets the maximum number of `Schedulables` that are allowed to run at any one time (across all levels) for the given lane.

`void setQueueComparator(Comparator comparator, int lane)`

Sets the `Comparator` used by the queue at this level for the given lane to order its elements (`Schedulables`). The 'smallest' value, as determined by the `Comparator`, is the first element of the queue. By default the queue is sorted by time (fifo).

`void setRightsSelector(RightsSelector selector, int lane)`

By default, the "right" to run is handled in a round-robin fashion. This can be overridden by providing a different `RightsSelector`. The rights-selection mechanism is experimental and lightly tested, and shouldn't be used (yet) except for experimenting.

`boolean setQualifier(UnaryPredicate predicate, int lane)`

Sets the qualifier at this level for the given lane. The qualifier is used to disqualify queued `Schedulables` temorarily. Any `Schedulable` on the queue which doesn't satisfy the predicate is removed and held in another list. Such a `Schedulable` will only be returned to the queue (and thus given the opportunity to run eventually) if the qualifier is unset (ie, set to `null`).

NB: The qualifier can only be set to a non-`null` value if it's current `null` .

`boolean setChildQualifier(UnaryPredicate predicate, int lane)`

Sets the qualifier for child schedulers at this level for the given lane. The qualifier is used to prevent children from gaining rights they might otherwise have access to. This is useful to keep one child from using up all its parents rights (for example).

NB: The qualifier can only be set to a non-`null` value if it's current `null` .

`void setMaxRunningThreadCount(int count)`

Sets the maximum number of `Schedulables` that are allowed to run at any one time (across all levels) for the default lane.

`void setQueueComparator(Comparator comparator)`

Sets the `Comparator` used by the queue at this level for the default lane to order its elements (`Schedulables`). The 'smallest' value, as determined by the `Comparator`, is the first element of the queue. By default the queue is sorted by time (fifo).

`void setRightsSelector(RightsSelector selector)`

By default, the "right" to run is handled in a round-robin fashion. This can be overridden by providing a different `RightsSelector`. The rights-selection mechanism is experimental and lightly tested, and shouldn't be used (yet) except for experimenting.

`boolean setQualifier(UnaryPredicate predicate)`

Sets the qualifier at this level for the default lane. The qualifier is used to disqualify queued `Schedulables` temorarily. Any `Schedulable` on the queue which doesn't satisfy the predicate is removed and held in another list. Such a `Schedulable` will only be returned to the queue (and thus given the opportunity to run eventually) if the qualifier is unset (ie, set to `null`).

NB: The qualifier can only be set to a non-`null` value if it's current `null` .

`boolean setChildQualifier(UnaryPredicate predicate)`

Sets the qualifier for child schedulers at this level for the default lane. The qualifier is used to prevent children from gaining rights they might otherwise have access to. This is useful to keep one child from using up all its parents rights (for example).

NB: The qualifier can only be set to a non-`null` value if it's current `null` .

`int getDefaultLane()`

Returns the current default lane.

`int runningThreadCount(int lane)`

Returns the number of `Schedulables` that are currently running at this level for the given lane.

`int pendingThreadCount(int lane)`

Returns the number of `Schedulables` that are currently queued at this level for the given lane.

`int activeThreadCount(int lane)`

Returns the number of `Schedulables` that are currently either running or queued at this level for the given lane.

`int maxRunningThreadCount(int lane)`

Returns the maximum number of `Schedulables` that are allowed to run at any one time (across all levels) for the given lane.

`int runningThreadCount()`

Returns the number of `Schedulables` that are currently running at this level for the default lane.

`int pendingThreadCount()`

Returns the number of `Schedulables` that are currently queued at this level for the default lane.

`int activeThreadCount()`

Returns the number of `Schedulables` that are currently either running or queued at this level for the default lane.

`int maxRunningThreadCount()`

Returns the maximum number of `Schedulables` that are allowed to run at any one time (across all levels) for the default lane.

org.cougaar.core.thread.ThreadListener

`void threadQueued(Schedulable schedulable, Object consumer)`

Subscribed `ThreadListeners` at any given level will receive this callback when any `Schedulable` is placed on the queue of pending 'threads' for that level. The consumer on whose behalf the `Schedulable` was created is also provided.

`void threadDequeued(Schedulable schedulable, Object consumer)`

Subscribed `ThreadListeners` at any given level will receive this callback when any `Schedulable` is popped of the queue of pending 'threads' for that level, in preparation for being run. The consumer on whose behalf the `Schedulable` was created is also provided.

`void threadStarted(Schedulable schedulable, Object consumer`

Subscribed `ThreadListeners` at any given level will receive this callback when any `Schedulable` at that level starts running. The consumer on whose behalf the `Schedulable` was created is also provided.

`void threadStopped(Schedulable schedulable, Object consumer)`

Subscribed `ThreadListeners` at any given level will receive this callback when any `Schedulable` at that level stops running. The consumer on whose behalf the `Schedulable` was created is also provided.

`void rightGiven(String consumer);`

Subscribed `ThreadListeners` at any given level will receive this callback when the next higher Thread Services have granted this level a "right" for the given thread consumer. No `Schedulable` has claimed the right at this point so none is passed in the callback.

`void rightReturned(String consumer)`

Subscribed `ThreadListeners` at any given level will receive this callback when that level returns a "right" to the next higher level. The `Schedulable` which used the right is already back in the thread pool at this point, so it isn't passed in the callback.

org.cougaar.core.service.ThreadListenerService

`void addListener(ThreadListener listener, int lane)`

Adds the given listener to the list of those which will receive callback notifications for 'interesting' thread events (queued, dequeued, started, stopped, etc) at this level of the Thread Services, for the given lane.

`void removeListener(ThreadListener listener, int lane)`

Removes the given listener from the list of those which will receive callback notifications for 'interesting' thread events (queued, dequeued, started, stopped, etc) at this level of the Thread Services, for the given lane.

`void addListener(ThreadListener listener)`

Adds the given listener to the list of those which will receive callback notifications for 'interesting' thread events (queued, dequeued, started, stopped, etc) at this level of the Thread Services, for the default lane.

`void removeListener(ThreadListener listener)`

Removes the given listener from the list of those which will receive callback notifications for 'interesting' thread events (queued, dequeued, started, stopped, etc) at this level of the Thread Services, for the default lane.

org.cougaar.core.service.ThreadService

`WILL_BLOCK_LANE`

This constant refers to the lane that should be used by threads which are known to block on io. MTS Destination Queue threads currently use this lane.

`CPU_INTENSE_LANE`

This constant refers to the lane that should be used by threads which are known to run unusually long [?]. This lane isn't currently in use.

`WELL_BEHAVED_LANE`

This constant refers to the lane that should be used by threads which are known to behave properly, ie not to block and not to run unusually long. The Metric Service threads use this lane.

`BEST_EFFORT_LANE`

This constant refers to the lane which should be used by all other threads. This is the initial default lane (for any given Thread Service, the default lane can also be changed dynamically).

`Schedulable getThread(Object consumer, Runnable runnable, String name, int lane)`

Creates a `Schedulable` with the given name and running the given `Runnable` on behalf of the given consumer (and `Object`). The `Schedulable` will be assigned to the given lane.

`Schedulable getThread(Object consumer, Runnable runnable, String name)`

Creates a `Schedulable` with the given name and running the given `Runnable` on behalf of the given consumer (and `Object`). The `Schedulable` will be assigned to this `Thread Service's` default lane.

`Schedulable getThread(Object consumer, Runnable runnable)`

Creates an anonymous `Schedulable` running the given `Runnable` on behalf of the given consumer (and `Object`). The `Schedulable` will be assigned to this `Thread Service's` default lane.

`void schedule(TimerTask task, long delay)`

This is the equivalent of the corresponding call on `java.util.Timer`. In fact the current implementation of the Thread Services creates a Timer at each layer and uses that Timer in this method, as well as in the next two methods.

THIS METHOD IS DEPRECATED.
Use the corresponding `schedule` on `Schedulable` instead.

`void schedule(TimerTask task, long delay, long interval)`

As above, this is the equivalent of the corresponding call on `java.util.Timer`.

*THIS METHOD IS DEPRECATED***.**
Use the corresponding `schedule` on `Schedulable` instead.

`void scheduleAtFixedRate(TimerTask task, long delay, long interval)`

As above, this is the equivalent of the corresponding call on `java.util.Timer`.

THIS METHOD IS DEPRECATED.
Use the corresponding `schedule` on `Schedulable` instead.

org.cougaar.core.thread.SchedulableStatus

The ScheduleStatus class is used to dynamically mark when a thread ventures into a blocking region. It is preferable to not allow threads to block and to refactor (see Use Cases and Examples) your component into a non-blocking form. Sometimes it is not possible to refactor out the blocking call. In these cases, the region of the code that blocks should be marked when the pooled thread enters and leaves the blocking region.

To help debugging, the blocking regions are displayed as part are as part of the TopServlet and the RogueThreadDetector. Since the `Schedulable` is marked with the blocking region, Cougaar thread scheduler policies can also use the status to help determine which `Schedulable` to run next or the max number of pooled threads.

The following status types are defined: `OTHER, WAIT, FILEIO, NETIO`
A `NOT_BLOCKING` type is a negitive number.

`void beginBlocking(int type, String excuse)`

> Mark the current running `Schedulable` as Blocking, with a given type and excuse. The static method finds the `Schedulable` for the running thread, so you do not need to do any bookkeeping your self. This call is cheap and should be used as close to the blocking region as possible

`void beginWait(String excuse)`

> Sets status type to `WAIT`

`void beginFileIO(String excuse)`

> Sets status type to `FILEIO`

`void beginNetIO(String excuse)`

> Sets status type to `NETIO`

`void endBlocking()`

> Call this method when leaving the blocking region

Here is an example usage:

```
try{
 SchedulableStatus.beginNetIO("RMI reference decode");
 object = RMIRemoteObjectDecoder.decode(ref);
 } catch (Throwable ex) {
   loggingService.error("Can't decode URI " +ref, ex);
 } finally {
  SchedulableStatus.endBlocking();
 }
```

3.2.15.5 Configuration

Command-line (-D) Properties

`org.cougaar.thread.trivial`

Set this to "true" to use the extremely simple Thread Service implementation (with no Schedulers, control, or listener callbacks). Later this will be handled via Component Descriptions.

Component Parameters

`BestEffortAbsCapacity`

Set this to an integer to specify the maximum number of threads that can ever run in the `BEST_EFFORT_LANE`. This only has relevace for the Node-level Thread Service. The default value is 300.

`WillBlockAbsCapacity`

Set this to an integer to specify the maximum number of threads that can ever run in the `WILL_BLOCK_LANE`. This only has relevace for the Node-level Thread Service. The default value is 30.

`CpuIntenseAbsCapacity`

Set this to an integer to specify the maximum number of threads that can ever run in the `CPU_INTENSE_LANE`. This only has relevace for the Node-level Thread Service. The default value is 2.

`WellBehavedAbsCapacity`

Set this to an integer to specify the maximum number of threads that can ever run in the `WELL_BEHAVED_LANE`. This only has relevace for the Node-level Thread Service. The default value is 2.

Servlets

`org.cougaar.core.thread.TopPlugin`

Dumps the currently running or queued Schedulables. It does not depend on the Metrics Service. This should be loaded into the NodeAgent.

`org.cougaar.core.thread.AgentSensorPlugin`

Publishes per-agent load average data into the Metrics Service. This should be loaded into the NodeAgent.

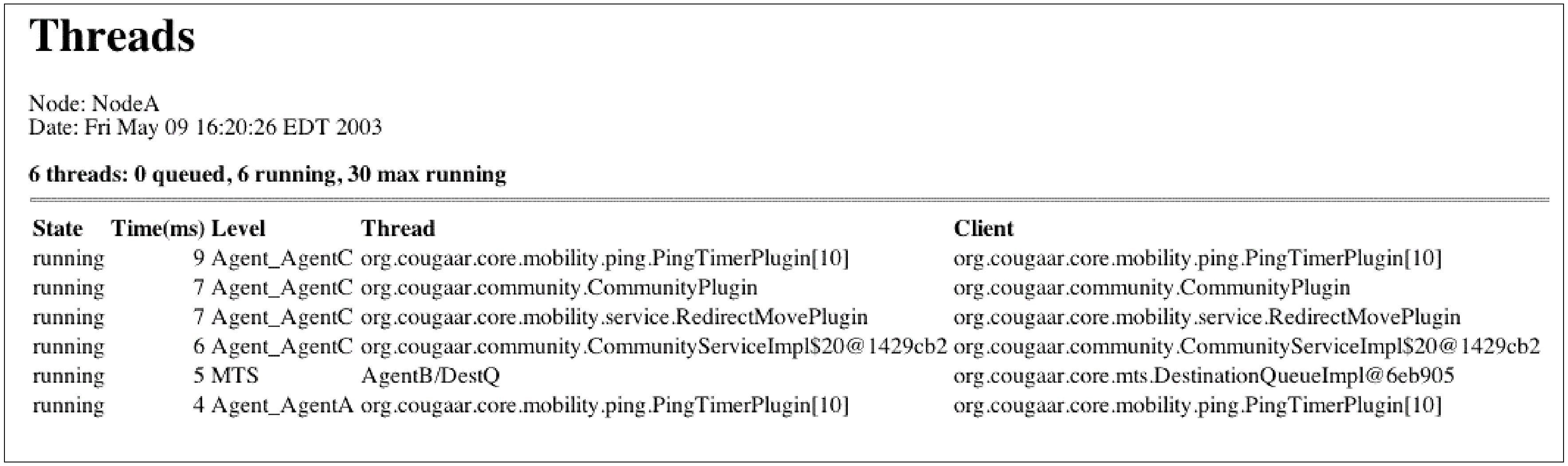
`org.cougaar.core.thread.ThreadWellBehavedPlugin`

Loading this plugin into an Agent causes that Agent to use the WELL_BEHAVED_LANE as its default. The plugin also accepts a parameter "defaultLane" to set some other initial default. For now the parameter value should be given directly as an integer, not symbolically.

This page is blank intentionally.

3.2.15.6 Operation

Top Servlet



Threads

Node: NodeA
Date: Fri May 09 16:20:26 EDT 2003

6 threads: 0 queued, 6 running, 30 max running

State	Time(ms)	Level	Thread	Client
running	9	Agent_AgentC	org.cougaar.core.mobility.ping.PingTimerPlugin[10]	org.cougaar.core.mobility.ping.PingTimerPlugin[10]
running	7	Agent_AgentC	org.cougaar.community.CommunityPlugin	org.cougaar.community.CommunityPlugin
running	7	Agent_AgentC	org.cougaar.core.mobility.service.RedirectMovePlugin	org.cougaar.core.mobility.service.RedirectMovePlugin
running	6	Agent_AgentC	org.cougaar.community.CommunityServiceImpl$20@1429cb2	org.cougaar.community.CommunityServiceImpl$20@1429cb2
running	5	MTS	AgentB/DestQ	org.cougaar.core.mts.DestinationQueueImpl@6eb905
running	4	Agent_AgentA	org.cougaar.core.mobility.ping.PingTimerPlugin[10]	org.cougaar.core.mobility.ping.PingTimerPlugin[10]

Figure 3-25: Top Servlet

This page is blank intentionally.

3.2.16 PrototypeRegistryService

This service is an Agent level service that provides an API to register prototype and property providers as well as late property providers. This service provides Assets an API to activate the late binding of PropertyGroups as well as an API to request that a newly created Asset's PropertyGroups be filled.

PropertyProvider plugins should use this service to cache prototype assets or to check and see if a prototype is already cached. All plugins may use this service to get prototypes from the registry by providing a type name and an Asset class.

```
package org.cougaar.planning.ldm;
import org.cougaar.core.component.Service;
import org.cougaar.planning.ldm.PrototypeProvider;
import org.cougaar.planning.ldm.PropertyProvider;
import org.cougaar.planning.ldm.LatePropertyProvider;
import org.cougaar.planning.ldm.asset.Asset;
import org.cougaar.planning.ldm.asset.PropertyGroup;

public interface PrototypeRegistryService extends Service {
  void addPrototypeProvider(PrototypeProvider prov);
  void addPropertyProvider(PropertyProvider prov);
  void addLatePropertyProvider(LatePropertyProvider lpp);

  /** Request that a prototype be remembered by the LDM so that
   * getPrototype(aTypeName) is likely to return aPrototype
   * without having to make calls to
PrototypeProvider.getPrototype(aTypeName).
   * Note that the lifespan of a prototype in the prototype registry may
   * be finite (or even zero!).
   * Note: this method should be used only by PrototypeProvider LDM Plugins.
   **/
  void cachePrototype(String aTypeName, Asset aPrototype);
  /** is there a prototype with the specified name currently in
   * the prototype cache?
   **/
  boolean isPrototypeCached(String aTypeName);

  /** find the prototype Asset named by aTypeName.  This service
   * will actually be provided by a PrototypeProvider via a call to
   * getPrototype(aTypeName).
   * It will return null if no prototype is found or can be created
   * with that name.
   * There is no need for a client of this method to call cachePrototype
   * on the returned object (that task is left to whatever prototypeProvider
   * was responsible for generating the prototype).
   * The returned Asset will usually, but not always have a primary
   * type identifier that is equal to the aTypeName.  In cases where
   * it does not match, aTypeName must appear as one of the extra type
   * identifiers of the returned asset.  PrototypeProviders should cache
   * the prototype under both type identifiers in these cases.
   * @param aTypeName specifies an Asset description.
   * @param anAssetClassHint is an optional hint to LDM plugins
   * to reduce their potential work load.  If non-null, the returned asset
   * (if any) should be an instance the specified class or one of its
   * subclasses.  When null, each PrototypeProvider will attemt to decode
   * the aTypeName enough to determine if it can supply prototypes of that
   * type.
   **/
  Asset getPrototype(String aTypeName, Class anAssetClass);
/** equivalent to getPrototype(aTypeName, null);
   **/
  Asset getPrototype(String aTypeName);

  /** Notify LDM of a newly created asset.  This is generally for the use
   * of LDMPlugins, but others may use it to request that propertygroups
   * of the new Asset be filled in from various data sources.
   **/
  void fillProperties(Asset anAsset);

  /** Used by assets to activate LateBinding of PropertyGroups to Assets.
   * Called as late as possible when it is not yet known if there is
   * a PG for an asset.
   **/
  PropertyGroup lateFillPropertyGroup(Asset anAsset, Class pg, long time);

  // metrics service hooks
  /** @return int Count of Prototype Providers  **/
  int getPrototypeProviderCount();

  /** @return int Count of Property Providers **/
  int getPropertyProviderCount();

  /** @return int Count of Cached Prototypes **/
  int getCachedPrototypeCount();
}
```

3.2.17 DomainService

This service is an Agent level service that provides access to Domain object factories. Users can request a domain specific Factory which constructs the blackboard objects appropriate to that domain. For example, the following call returns the Factory for the planning domain:

```
(PlanningFactory)domainService.getFactory("planning");
```

For the planning domain, org.cougaar.planning.plugin.legacy.PluginAdapter includes the following call utility to return the PlanningFactory:

```
 getFactory()
```

```

package org.cougaar.core.service;

import java.util.List;
import org.cougaar.core.component.Service;
import org.cougaar.core.domain.Factory;

public interface DomainService extends Service {
  /** return a domain-specific factory **/
  Factory getFactory(String domainName);

  /** return a domain-specific factory **/
  Factory getFactory(Class domainClass);

  /** return a list of all domain-specific factories **/
  List getFactories();
}
```

3.2.17.1 Domain Configuration

When a node is started, the set of domains available to all agents in the node is specified as command line arguments or in the `LDMDomains.ini` configuration file. The arguments are of the form:

```
-Dorg.cougaar.planning.ldm.<suffix>=<classname>
```

The suffix must be distinct from the suffixes of all the other domains being used within the node. For example, if we wanted a domain called `my.domain.Library`, we might use the following option:

```
-Dorg.cougaar.planning.ldm.library=my.domain.Library
```

The LDMDomains.ini file has lines of the form:

```
<suffix>=<classname>
```

The interpretation is the same as for the command line arguments.

In addition, domains may be added to specific agents through the agent XML file. Domains loaded through the agent XML are only available within that agent. Agents which communicate with each other should load the same set of domains. Without a common set of domains, cross agent messages will not be correctly interpreted.

To add a domain to the "<agent ..>" XML tag, add the following line:

```
<component
  name="DOMAIN_NAME"
  insertionpoint="Node.AgentManager.Agent.DomainManager.Domain"
  class="DOMAIN_CLASS">
  <argument>DOMAIN_NAME</argument>
</component>
```

For example, to load the glm Domain, the following line should be added to the agent tags of all agents using the glm domain:

```
<agent ..>
  ..
  <component
    name="glm"
    insertionpoint="Node.AgentManager.Agent.DomainManager.Domain"
    class="Node.AgentManager.Agent.DomainManager.Domain">
    <argument>glm</argument>
  </component>
  ..
</agent>
```

It is important that the <domain name> argument match the String returned by Domain.getDomainName() exactly. Otherwise, the domain will not loaded.

3.2.17.2 Creating a New Domain

A domain provides the context within which a set of Plugins and logic providers operate. This context consists of the objects that are used within the domain, the values used within these objects, and the messages that are sent from one agent to another. A domain isavailable within the agent through the DomainService.

The Domain architecture has transitioned to a component model. Domains are Components and must implement the `org.cougaar.core.domain.DomainBase` interface. Domains will attach to Binders that allow them access to their parent component (DomainManager) and various services such as the Blackboard. This change should only affect those who are responsible for creating or maintaining Domains. The DomainService, which provides general access to the Domains within an agent, has not changed.

For more information about the Cougaar Component Model, please review the Cougaar Architecture Document.

The `org.cougaar.core.domain.Domain`Adapter class has been added to facilitate domain development. It implements the `org.cougaar.core.domain.DomainBase` interface and should be used as a base class whenever new domains are developed. Subclasses must implement 4 methods to specify the name, the `Factory`, the `XPlanServesBlackboard` and the logic providers for the domain. Each method is described below:

org.cougaar.core.domain.DomainAdapter.getDomainName()

This public method should return the domain name. The name will be used to uniquely identify the domain. For example, `DomainService.getFactory(String domainname)` uses `Domain.getDomainName()` to find the domain specified by the `domainname` argument

org.cougaar.core.domain.DomainAdapter.loadFactory()

This protected method is called within `org.cougaar.core.domain.DomainAdapter.load()` and loads the Factory for this Domain. The method should call `org.cougaar.core.domain.DomainAdapter.setFactory()` to set the factory for this Domain. The factory allows you to create objects that are not defined by other domains. The factory may do nothing. This is often the case when your domain shares the objects of another domain.

If you have your own kinds of objects that you want to manufacture, this is where they should be created. You must provide an implementation of `org.cougaar.core.domain.Factory` for your domain. The `org.cougaar.core.domain.Factory` is simply a marker interface. The actual methods are up to you, but they need to be known to your Plugins. The implementation of the `org.cougaar.planning.ldm.PlanningFactory` can be used as a model for your own factory.

org.cougaar.core.domain.DomainAdapter.loadXPlan()

This protected method is called within `org.cougaar.core.domain.DomainAdapter.load()` and loads the `XPlanServesBlackboard` for this Domain. . The method should call `org.cougaar.core.domain.DomainAdapter.setXPlan()` to set theXPlan for this Domain. An XPlan supports your logic providers. In particular, an XPlan has subscriptions that allow your logic providers to efficiently access the Blackboard (under the planning domain: LogPlan). For example, you might have a Book object and you might want subscriptions that permit you to find a particular Book by title (or author, *etc*.). Using subscriptions based on `HashMap` collections would allow your logic providers to efficiently locate the relevant object in your `BookPlan`.

Many of the methods declared in `XPlanServesBlackboard` can be implemented by simply delegating them to the underlying Blackboard. Most implementations of `XPlanServesBlackboard` should do exactly this.

You may not need your own XPlan. If this is the case, you can share the XPlan from some other Domain. For example, you might use the LogPlan XPlan from the PlanningDomain. The DomainBindingSet can be used to get the current set of Xplans from which you can select the one you want. If the one you want is not already there, then you must instantiate it yourself. Subsequent Domains may then share the XPlan you create. The following illustrates a version of this method that shares the LogPlan Xplan:

```
protected void loadXPlan() {
    DomainBindingSite bindingSite =  (DomainBindingSite)  getBindingSite();
    LogPlan logPlan = new LogPlan();
    setXPlan(logPlan);
  }
```

There should be at most one instance of any particular XPlan in an Agent. The order of creation of domains is not specified so this method cannot assume that the desired XPlan is already present or not.

org.cougaar.core.domain.DomainAdapter.loadLPs()

This protected method is called within `org.cougaar.core.domain.DomainAdapter.load()` and loads the logic providers for this Domain. . The method should call `org.cougaar.core.domain.DomainAdapter.addLogicProvider()` for each logic provider specific to this Domain. Logic providers primarily supply inter-agent communication. If you are using objects specific to this domain you will have to create logic providers so that your agents can communicate. A logic provider might process a particular class of object, create a Directive and send the directive to another agent where a complementary logic provider receives the Directive and adds, changes, or removes objects in that other agent. The logic providers that you write will be given the XPlan that you create so they can use the services of your XPlan.

3.2.18 UIDService

This service is an Agent level service that provides unique identifiers (UIDs) for blackboard objects that implement the UniqueObject interface.

```
Package org.cougaar.core.util;
Interface UniqueObject {
  UID getUID();
}
```

There is one UID server per Agent. The Cougaar object factories use this service when creating unique objects; however, if you are creating a new instance of a unique object without using a factory, this service will generate and set the UID for the object if registerUniqueObject(UniqueObject o) is called.

```

package org.cougaar.core.agent;
import org.cougaar.core.util.UID;
import org.cougaar.core.util.UniqueObject;
import org.cougaar.core.mts.MessageAddress;
import org.cougaar.core.component.Service;

/** For backwards compatability UIDServer contains the following. **/

public interface UIDService extends Service {
 /** MessageAddress of the proxy server.   **/
  MessageAddress getMessageAddress();

 /** get the next Unique ID for the Proxiable object registry
    * at this server.
    * It is better for Factories to use the registerUniqueObject method.
    **/
  UID nextUID();

  /** assign a new UID to a unique object.
    **/
  UID registerUniqueObject(UniqueObject o);

}
```

3.2.19 NodeControlService

NodeControlService is a Service which is available exclusively to subcomponents of NodeAgent. Furthermore, in societies with strict access control enables, access to NodeControlService will likely be extremely limited to avoid security issues. This Service is new for Cougaar 9.2.

```
package org.cougaar.core.node;
interface NodeControlService extends Service
{
  // return the ServiceBroker for the entire Node.
  // This allows the client of this service to offer
  // other services to all components in a node,
  // including to all Agents.
  ServiceBroker getRootServiceBroker();

  // Return the root contained of the Node.
  // This allows clients of this service to control
  // loading and unloading of agents, for instance.
  Container getRootContainer();
}
```

3.2.20 Quality Objects (QuO) Support

The Cougaar QuO module adds runtime support for Quality Objects (QuO) . QuO is used to add Quality of Service (QoS) adaption to object oriented software, such as Cougaars internal infrastructure.

This module adds support for the QuO kernel, which is a factory for QuO SystemConds and Contracts. The Metrics Service also uses QuO Resource Status Service (RSS) to implement the Metrics Service engine. This module implements helper functions which allow easy interfacing between the Cougaar Metrics service and the QuO Kernel. Specifically it adds:

- Metrics SysConds get their values from the Metrics Service
- A Syscond factory for creating Metrics Sysconds for different types of metrics and for reusing metrics.

QuO has been used with Cougaar to make QoS adaptive MTS Aspects for dynamically enabling compression and SSL. These MTS Aspects enable or disable compression and SSL based on the network conditions. The Ultralog society is the largest example of an application enhanced with QuO. The 2004 assessment had two types of QuO Qoskets, with 1000's of Contracts and Syscond instances running on around 100 hosts.

3.2.20.1 Configuration

The SyscondFactory is the only component supplied by this module. It needs to be loaded with the AgentHostUpdaterComponent. These modules are loaded as part of the basic Metric Service rules and will be part of the "Standard" node.xsl template.

The Acme rule is:

```
agentHostUpdaterComponent =
"org.cougaar.core.qos.rss.AgentHostUpdaterComponent"

syscondFactory = "org.cougaar.lib.mquo.SyscondFactory"
```

```
metrics_service_components_to_add = [agentHostUpdaterComponent,
syscondFactory]

society.each_node_agent() do |node_agent|
   metrics_service_components_to_add.each { |comp|
     node_agent.add_component do |c|
     c.classname = comp
     c.insertionpoint = "Node.AgentManager.Agent.MetricsServices.Component"
     end
    }
end
```

3.2.20.2 Operation

The QuO GUI can be enabled to view of status of QuO Sysconds, Contracts, and Metrics. The QuO GUI is a swing window so the `$DISPLAY` enviroment variable must be set. The GUI is enabled by loading the `org.cougaar.lib.mquo.QuoGuiComponent` into the `NodeAgent`.

Note: A QuO contract viewer servlet is available from Ultralog Robustness UC3. This allows viewing of QuO Contracts using the Cougaar Servlet interface.

3.2.21 ViewService

The ViewService is only service that is built into the component model itself. The ViewService can be used to see which components are loaded and which services they have advertised and obtained.

```
package org.cougaar.core.component;

import org.cougaar.core.component.Service;

public interface ViewService extends Service {
  ComponentView getComponentView(); // get the component view of itself
}

public interface ComponentView {
  int getId();  // unique identifier
  long getTimestamp();  // time when loaded
  ComponentDescription getComponentDescription(); // component class + args
  ContainerView getContainerView();  // parent component's view
  Map getAdvertisedServices(); // Map<Class, ServiceView> of added services
  Map getObtainedServices(); // Map<Class, ServiceView> of obtained services
}

public interface ContainerView extends ComponentView {
  // all of "ComponentView", plus:
  List getChildViews();  // List<ComponentView> of child components
}

public interface ServiceView {
  int getId();  // unique identifier
  long getTimestamp(); // time when advertised/obtained
  int getProviderId(); // identifier of component that advertised this service
  ComponentDescription getProviderComponentDescription();
  Map getIndirectlyAdvertisedServices();  // added via "get*ServiceBroker"
}
```

A servlet is provided to traverse the ViewService's view objects and display the result as XML. To load the servlet, add the following component to the node's XML file:

```
<component
  class="org.cougaar.core.util.ComponentViewServlet">
  <argument>/components</argument>
</component>
```

A trimmed XML example output of http://localhost:8800/components is provided below. The XML shows that the BlackboardService is advertised by the StandardBlackboard component, which in turn uses the MessageSwitchService advertised by another component with identifier "711":

```
<?xml version='1.0'?>
<component-model-view agent='NodeA'>
  <container-view id='1'>
    <component
      class='org.cougaar.core.agent.AgentManager' priority='HIGH'
      insertionpoint='Node.Component'/> ..
    <children> ..
        <component-view id='808'>
          <component
            class='org.cougaar.core.blackboard.StandardBlackboard'
            priority='BINDER'
            insertionpoint='Node.AgentManager.Agent.Component'/>
          <advertised-services> ..
            <service
              class='org.cougaar.core.service.BlackboardService'
              id='838'/>
          </advertised-services>
          <obtained-services>
            <service
              class='org.cougaar.core.agent.service.MessageSwitchService'
              id='809'
              sp-id='711'/> ..
          </obtained-services>
        </component-view> ..
```

Note that the ViewService displays both developer plugins and the infrastructure components specified in the XSL templates, as discussed in section 9.1.17. For additional details, see bug 1113 at http://bugs.cougaar.org.

3.3 How to Write a Service

3.3.1 Defining a Service API

In order to build a Service, a developer must define an interface that extends the Service API found in org.cougaar.core.component.Service. This interface should expose methods that allow other components to interact with the new service. The implementation of this interface can be a class or an inner class. Once the Service interface and implementation have been defined, a ServiceProvider should be developed.

3.3.2 Writing a Service Provider

ServiceProviders are called by a ServiceBroker whenever any of the ServiceProvider's services are requested by a component. A ServiceProvider is an object, usually part of a Component, which can construct service implementations for client components. A ServiceProvider can manage multiple services, however, each service (and its provider) should be registered with a component's ServiceBroker. Every ServiceProvider must implement the ServiceProvider interface. The ServiceProvider's getService method should create a new service implementation to return to the requestor or return a reference to an already existing implementation thate sha can be shared among clients. Note that these implementations should be cast to the service's interface, or a proxy can be returned. If a ServiceProvider is used to manage multiple services, a class check should be placed in the getService and releaseService methods to determine what class is being requested (use the serviceClass argument).

```
package org.cougaar.core.component;

/** Like BeanContextServiceProvider **/
public interface ServiceProvider
{
  /** @param serviceClass a Class, usually an interface, which extends
Service. **/
  Object getService(ServiceBroker sb, Object requestor, Class serviceClass);

  void releaseService(ServiceBroker sb, Object requestor, Class serviceClass,
Object service);
}
```

3.3.3 Adding the Service to a ServiceBroker

Once you have written a Service and a ServiceProvider, they must be registered with a Component ServiceBroker. Inside of a Component's initialize method, add code similar to the example below.

1. Get your parent container's service broker to allow other components at the same level to access your services (i.e. add the BlackboardService to Agent's ServiceBroker to allow PluginManager and below components to access the service).

```
ServiceBroker sb = bindingSite.getServiceBroker();
```

OR… just get your Container's ServiceBroker

```
ServiceBroker sb = getServiceBroker();
```

2. Create your ServiceProvider

```
BlackboardServiceProvider bbSP = new BlackboardServiceProvider(myargs);
```

3. Add the Service to the ServiceBroker linking the Service and the ServiceProvider

```
sb.addService(BlackboardService.class, bbSP);
```

4 Binders and Binding Sites

A *Component Model* describes how modular software objects (Components) relate to each other: How one Component may contain another, and how one Component may request services from another. The Cougaar Component model is comprised of four parts.

- The definition of what constitutes a Component: what it takes to define a Component, and how the system recognizes it.

- A Component Containment Model which describes how Components are created, how such components are arranged into a hierarchy of objects, and how the structure of the hierarchy structures interactions between any two Components.

- A Service Model which describes ways that Components, especially ones which are distant in terms of the Component Hierarchy, may discover and contact each other to develop dynamic relationships similar to traditional Client/Server Service boundaries.

- A Binder Model which allows each parent Component to manage effectively its children, including protecting itself (and the rest of the hierarchy) from the possibly less-trusted child Components.

For a more detailed discussion of the theory behind Components, Binders, and Services, see the Cougaar Architecture Document (CAD). This section recapitulates parts of the discussion from the CAD, while extending it with Binder Model details including the basics of writing a new Binder.

4.1 Binder Model and Implementation

One problem with most simple component models 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 layer 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.

Note that the Cougaar Component Model 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, such support is not part of the core Cougaar software base.

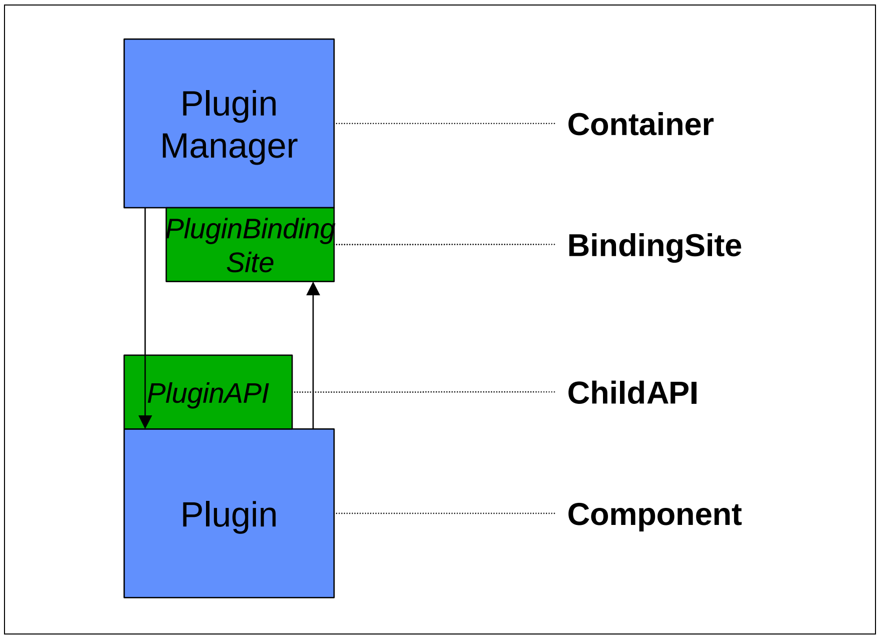


Figure 4-1: Component Containment

4.1.1 Component

A *Component* is a pluggable software entity which has an existence and identity separate from any other (Analagous 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. A Component may be named or described by a ComponentDescription instance (see section 4.1.2).

4.1.2 Component Description

A ComponentDescription object describes a Component in its uninstantiated form. ComponentDescription instances may be used with Container methods to add subcomponents. XML file descriptions of plugins, agents, and other components map directly to ComponentDescription objects which are then added to the appropriate parent objects.

```
package org.cougaar.core.component;
public final class ComponentDescription {
  /** Higher priority than internal components. **/
  public final static int PRIORITY_HIGH;

  /** Same priority as internal subcomponents. **/
  public final static int PRIORITY_INTERNAL;

  /** Binders are typically loaded before subcomponents **/
  public final static int PRIORITY_BINDER;

  /** Standard subcomponent (including plugin) priority **/
  public final static int PRIORITY_COMPONENT;

  /** Load after standard subcomponents **/
  public final static int PRIORITY_LOW;

  /**
   * The name of a particular component, used both as the displayable
identifier of
   * the Component and to disambiguate between multiple similar components
which might
   * otherwise appear equal.
   **/
  public String getName() { return name; }

  /**
   * The point in the component hierarchy where the component
   * should be inserted.  It is used by the component hierarchy to
   * determine the container component the plugin should be
   * added.  This point is interpreted individually by each
   * (parent) component as it propagates through the container
   * hierarchy - it may be interpreted any number of times along
   * the way to the final insertion point.
   */
  public String getInsertionPoint() { return insertionPoint; }

  /**
   * The name of the class to instantiate to construct the component.
   **/
  public String getClassname() { return classname; }

  /**
   * Where the code for classname should be loaded from.
   **/
  public URL getCodebase() { return codebase; }

  /**
   * A parameter supplied to the component immediately
   * after construction by calling instance.setParameter(param);
   * using reflection.
   **/
  public Object getParameter() { return parameter; }

  /** Load Priority of the component in it's Container relative to
   * other child components loaded at the same time at the same Containment
point.
   * Two components loaded at the same point and time with the same priority
   * will be loaded in the order they are specified in.
   **/
  public int getPriority() { return priority; }
}
```

4.1.3 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. The Collection members of a Container are usually ComponentDescription instances, which are instantiated and destroyed appropriately by the Container as the components are added and removed.

4.1.4 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.

```
package org.cougaar.core.component;
public interface BindingSite {
  //request access to the ServiceBroker layer of the parent.
  ServiceBroker getServiceBroker();
  //A Component may call this method to request that it be stopped.
  void requestStop();
}
```

Prior to Cougaar 10.0, the core Cougaar infrastructure often extended BindingSite for each Container class in the hierarchy to allow for simple access to the parent. While convenient for the Child component authors, this design pattern makes it more difficult to write multi-purpose Binders and Child components which may be inserted in multiple places in the Component tree. Starting with Cougaar 10.0, only plain BindingSite is used, relying on extra Services to give the same sort of information and control access as had been implemented with custom BindingSites. Both design patterns will continue to be supported for the foreseeable future, so developers should choose the one which makes the most sense for their application. We have left this chapter discussing PluginBindingSite as a more complete example of the flexibility of the Service model, even though the actual Plugin code now uses the plain BindingSite class instead.

4.1.5 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.

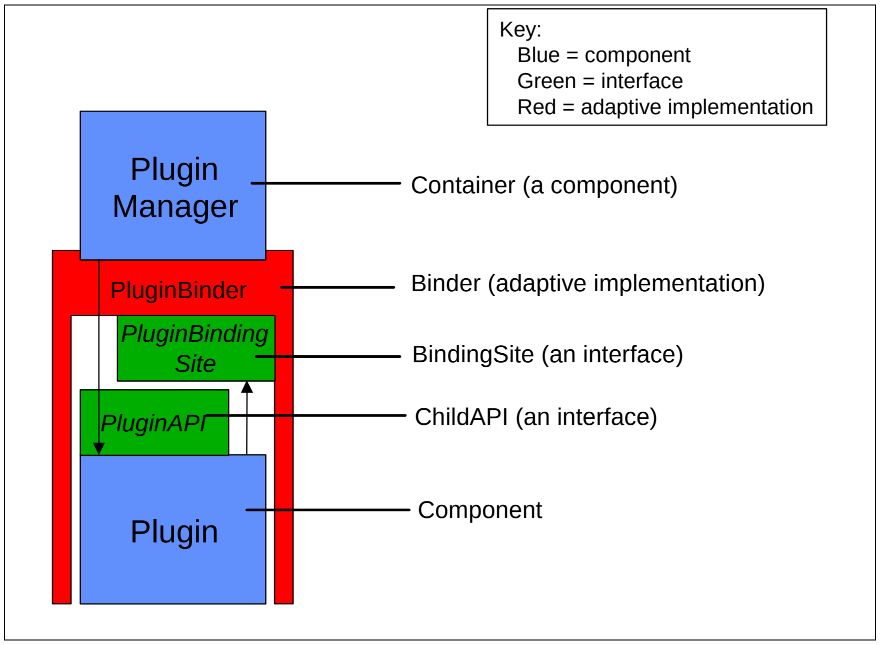


Figure 4-2: Component Binder

A BindingSite is usually defined as a very restricted interface, most often a vacuous extension of BindingSite, which offers only a hook into the Service Model (via getServiceBroker) and some basic ContainmentModel requests. A Binder typically implements the associated BindingSite via a private proxy object rather than directly implementing it so that the bound component cannot re-cast the BindingSite to a less restrictive class (see BinderSupport section below).

```
package org.cougaar.core.component;
public interface Binder {
  // Treat initialize() as an extended constructor.
  void initialize();
  // After initialize and before load, an object is notified about its
  // parents, services, etc.  After load, it should be ready to run (but not
  // actually running).
  void load();
  // Called object should start any threads it requires.
  void start();
  // Called object should pause operations in such a way that they may
  // be cleanly resumed or the object can be unloaded.
  void suspend();
  // Restore a suspended component back to active mode.
  void resume();
  // Fully stop an idle component.
  void stop() throws StateModelException;
  // stop a running component
  void halt() throws StateModelException;
  // unload a loaded (stopped, etc) component
  void unload() throws StateModelException;
  // return the current state of the component.
  int getModelState();
}
```

4.1.6 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.

In order to insert a new Binder class into the system, one must write a matching BinderFactory component that allows the appropriate parent component to figure out which Binder or Binders to use on activating a given component. A BinderFactory instance is a Component which plugs into the parent at the same level as the components it wants to Bind. Note that since BinderFactories are themselves components, they too are subject to binding, albeit usually by very different Binders (usually provided only by the core system).

```
package org.cougaar.core.component;
public interface BinderFactory {
  // get the priority (nesting level) of this binder factory
  int getPriority();
  // Get or Construct a binder for a child component.
  Binder getBinder(Object child);
  // Get the BinderFactory's ComponentFactory.
  ComponentFactory getComponentFactory();
}
```

4.1.7 BinderFactorySupport

BinderFactorySupport is a convenience base implementation for constructing new BinderFactory implementations.

```
package org.cougaar.core.component;
public abstract class BinderFactorySupport {
  //Override to choose the class of the Binder to use.
  protected Class getBinderClass(Object child);
  // override to set a higher priority.
  public int getPriority();
  // override to provide a non-default component factory.
  public ComponentFactory getComponentFactory();
}
```

4.1.8 BinderSupport

BinderSupport is a convenience base implementation of a Binder to ease development of a new Binder/BinderFactory combination.

```
package org.cougaar.core.component;
public abstract class BinderSupport {
  // Defines a pass-through insulation layer to ensure that the plugin cannot
  // downcast the BindingSite to the Binder.
  protected abstract BindingSite getBinderProxy();
}
```

4.1.9 WrappingBinderFactory

The Binder Model requires that a component be directly "bound" by exactly one Binder. Often, however, it is desirable to write Binders which only know about one particular type of security, policy or adaptivity. A WrappingBinderFactory is a BinderFactory which is allowed to bind other binder instances – effectively allowing arbitrary layering of Binder instances between the parent and child components.

4.2 Binder Support of 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.

In particular, the only access that a new Component has to the Service Model is through the BindingSite to its parent, which is usually actually a Binder implementation. A simplistic Binder implementation may simply pass through requests for the ServiceBroken whereas a Binder which wished to provide increased security will prefer to provide its own proxy to the "real" ServiceBroker of the parent component so that it may monitor, veto or modify all actual ServiceModel requests.

What follows is a recapitulation of the Cougaar Architecture Document (CAD) from the perspective of a Binder writer.

4.2.1 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. There are no specific requirements for how a Service is defined.

Binder implementations will often want to mediate client access to Services.

4.2.2 Service Implementation

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

Binders have no control over the ServiceImplementation. However, Binders may proxy ServiceImplementations by wrapping the implementation with an object that delegates some set of mediated functionality to the real ServiceImplementation.

4.2.3 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.

Binders may want to restrict bound components from offering new services by refusing to allow them to register new ServiceProvider implementations to the ServiceBroker. Alternatively, a Binder might allow a modified ServiceProvider to be registered, thereby allowing the Binder to mediate between any other component which requests such a service.

4.2.4 ServiceBroker

A *ServiceBroker* is essentially just 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 most common behavior of a Binder will be to provide its own custom ServiceBroker implementation when the bound Component requests. This effectively allows the Binder complete control of all Service Model related requests made by the bound component.

4.3 Implementing Adaptive Behavior with a Binder

There are a number of ways that Binders may be written so as to mediate, manage or insulate components from potential problem components. A fully operational Binder will be comprised of:

- A BinderFactory Component to create Binder instances on behalf of the parent to bind putative child Components.
- A Binder class to manage the child component (or, in the case of a WrappingBinding, the wrapped Binder) and to provide a managed ServiceBroker for the child.
- A ServiceBroker implementation to manage any or all service interactions of the child Component based whatever policies implemented by the associated Binder.

4.3.1 Binders

Binder implementations may implement their BindingSite methods in whatever way required, including implementing behavior tuned specifically to the bound Component, even by altering the behavior 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 must 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.

4.3.2 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.

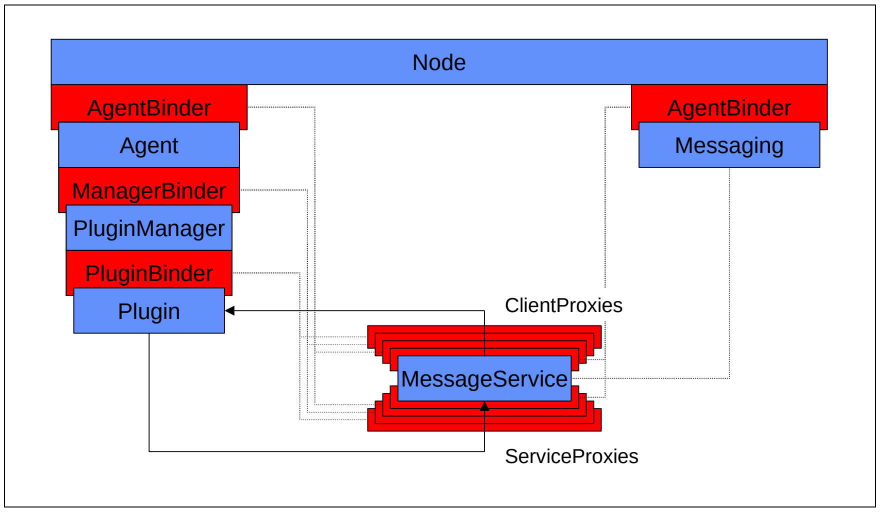


Figure 4-3: Complex Service Chain

4.3.3 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.

4.3.4 ServiceFilter

ServiceFilter is a convenience base class for implementing a Binder suite which proxies the ServiceBroker, providing the following functionality: Optionally veto any or all service requests, Optionally proxy the ServiceImplementation or the client for the Service for any service request.

An example of using ServiceFilter is in the examples of the core module development package as org.cougaar.core.examples.PluginServiceFilter, which audits the bound plugins use of new subscriptions.

```
package org.cougaar.core.component;
public class ServiceFilter {
  // override to select the right Binder class instead of ServiceFilterBinder
  protected Class getBinderClass(Object child);
}
public abstract class ServiceFilterBinder {
  // define to choose an extension of ServiceFilterBinderProxy
  protected abstract ContainerAPI createContainerProxy();

  // Base class for filtering/auditing/etc services.
  public class FilteringServiceBroker {
    // Override to control if a given service class is allowed or not.
    protected boolean allowService(Class serviceClass);
    // Override to specify an alternative instance to use as the client.
    protected Object getClientProxy(Object client, Class serviceClass);
    // Override to specify an alternative instance to use as the service
implementation
    protected Object getServiceProxy(Object service, Class serviceClass,
Object client);
    // Called to release a serviceProxy previously constructed by the binder.
    protected void releaseServiceProxy(Object serviceProxy, Object service,
Class serviceClass);
    //Called to release a clientProxy previously constructed by the binder.
    protected void releaseClientProxy(Object clientProxy, Object client, Class
serviceClass);
  }
}
```

4.4 Infrastructure implementation of Binders

The core infrastructure is, itself, a hierarchy of Components, all bound, with a network of Service relationships derived at runtime – there are no special "back door" mechanisms allows to the infrastructure which are unavailable to any other Component (plugin, etc). Figure 3-4 illustrates the actual core infrastructure indicating all binders and components (as of Cougaar 10.0).

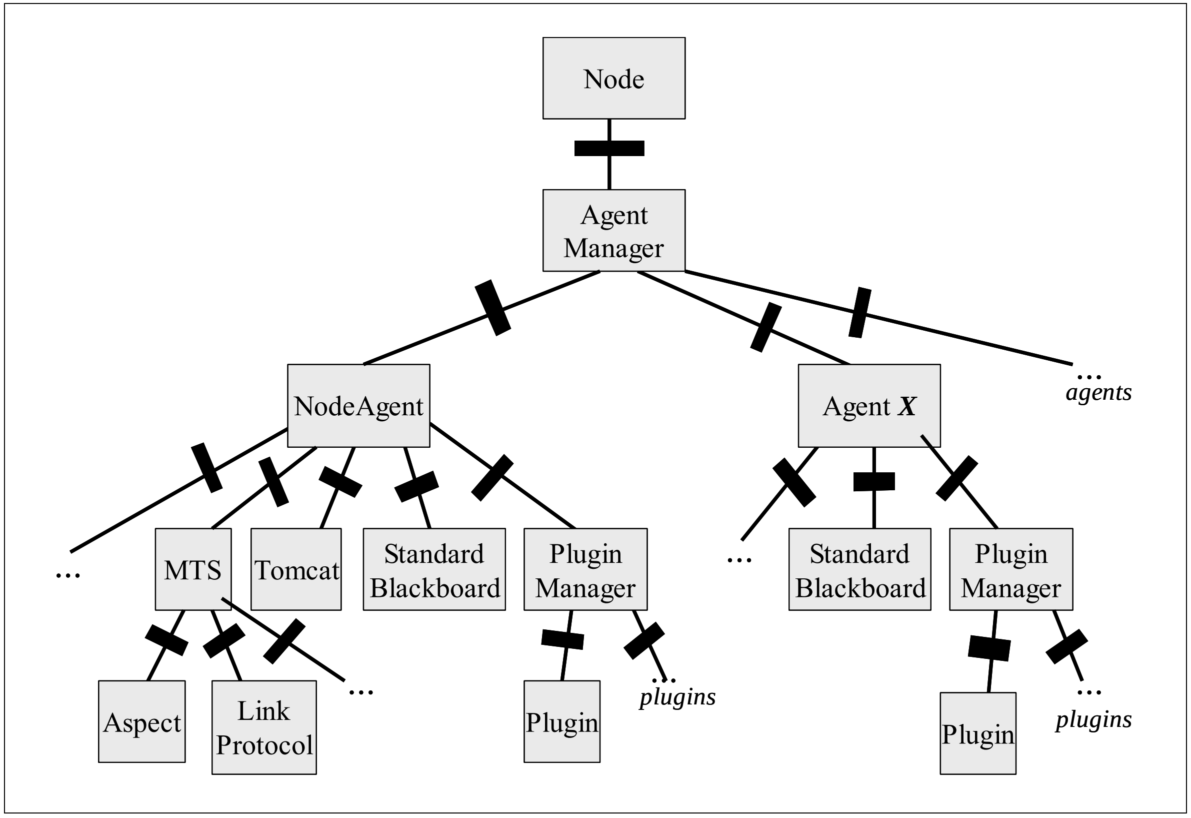


Figure 4-4: Implemented Components (gray boxes) and Binders (black solid rectangles)

5 Web-based User Interfaces

This section introduces Cougaar's support for Web-based "Servlet" interfaces. Section TWRIGHT presents configuration options, such as selecting an alternate HTTP port, enabling HTTPS, and enabling client-authentication.

5.1 Introduction

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 servlet server's white pages registration entry. 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.

5.2 Built-in Servlets

The servlet server includes many built-in servlets that are always loaded, providing basic support for listing servlet paths and locating agents. In the examples below, assume that:

- Host "localhost" is running Cougaar node "MyNode" with HTTP support on port 8800.
- Agent "foo" is on "MyNode"
- Agent "foo" has an internally registered (user-developed) servlet with path "/hello".
- Node "MyNode" has an internally registered (user-developed) servlet with path "/testme".
- A second cougaar node is on host "x.com" and HTTP port 9876.
- There is no agent named "junk" in the society.

Built-in servlets include:

- http://localhost:8800

  This URL generate a simple help page, plus links to some of the built-in servlets listed below. This is the best place to start.

- http://localhost:8800/agents

  This URL lists the name of all agents on the local node, in this example: "MyNode" and "foo". The generated page also includes an option to list all agents in the society, using the naming service to list the names. There are many URL options, including a format option for plain-text, limits on the number of agents listed or time spent searching for agents, and hierarchical name listings. See the javadocs for "org.cougaar.lib.web.service.AgentsServlet" for URL details.

- http://localhost:8800/$foo

  Generate a help page for agent "foo", with links to the other built-in servlets.

- http://localhost:8800/$foo/list

  List the servlet paths that are registered in agent "foo". In this example, both the built-in servlets will be listed ("/agents" and "/list") and the developer's "/hello" servlet.

- http://localhost:8800/$foo/agents

  Runs the "/agents" servlet on the node that is running agent "foo". This is handy for listing co-located agents.

- http://localhost:8800/testme

  Invokes node "MyNode"'s "/testme" servlet. This is how node-level servlets are invoked.

- http://localhost:8800/$foo/hello

  Invokes agent "foo"'s "/hello" servlet. This is how agent-level servlets are invoked.

- http://x.com:9876/$foo/hello

  Note that this URL is asking the wrong node for agent "foo", so a built-in redirect servlet will query the naming service for agent "foo"'s URL and redirect the client to http://localhost:8800/$foo/hello. In general, a client can use any node's server as a "gateway" to all the agents in the distributed society, relying upon the built-in redirection to guide the request to the correct host:port address.

- http://localhost:8800/$junk/hello

  In this example, there's no agent named "junk", so this will generate an HTML error page with all local agent names and the "/hello" path appended onto the end.

  By default, Internet Explorer hides this error page, so developer's should disable the "friendly HTML error page" option:

  tools->Internet Options->Advanced->Show Friendly HTTP error messages == UNCHECKED

- http://localhost:8800/$~foo/testme

  This accesses the node-agent for agent "foo", "MyNode", and invokes the "/testme" servlet. This can be handy if your society has a complex mapping of agents to nodes, or randomly generated node names.

5.3 Creating Custom Servlets

Servlets are very much like plugins; they are specified in the XML configuration, loaded into specific agents or node-agents, and can only access the information within that agent (e.g. the local blackboard). Each agent has its own set of servlets.

Servlets must register with their agent's "org.cougaar.core.service.ServletService". Developers can use the "org.cougaar.core.servlet.ComponentServlet" base class to handle this registration. Deprecated servlet base classes include "SimpleServletSupport" and "BaseServletComponent".

Here is an example servlet:

```
package foo;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.cougaar.core.component.ComponentServlet;

public class HelloServlet
  extends ComponentServlet  // similar to "HttpServlet" base class
{
  /** optional: get the default path for this servlet. */
  protected String getPath() {
    return "/hello";
  }

  /** handle typical browser requests */
  public void doGet(
     HttpServletRequest req,
     HttpServletResponse res) throws IOException {
    PrintWriter out = res.getWriter();
    out.print("Hello, world!");
    out.flush();
    out.close();
  }
}
```

Here is an example XML line for loading the above servlet:

```
<agent ..>
  ..
  <component class="foo.HelloServlet">
   <argument>/hello</argument>
  </component>
  ..
</agent>
```

Accessing "http://localhost:8800/hello" should generate the following text:

```
Hello, world!
```

The servlet can be modified to generate a proper HTML page, e.g.:

```
  ..
  out.println("<html>");
  out.println("<head><title>My Servlet</title></head>");
  out.println("<body>");
  // look for a parameter, e.g. "/hello?color=blue"
  String color = req.getParameter("color);
  if (color == null || color.equals("")) {
    color = "red";
  }
  out.println("<font color=\""+color+"\">Hello in "+color+"!</font>");
  out.println("</body>");
  out.println("</html>");
  ..
```

All the standard Servlet APIs are supported for accessing URL parameters, sending back raw data, and other options. See Sun's Servlet tutorial for details: http://java.sun.com/docs/books/tutorial/servlets

Note that, in HTML FORMs, developers should avoid HTTP POST ("doPost(..)"), since "/$" redirects will not work correctly. HTTP GET forms work fine, since redirected parameters will be preserved as URL parameters:

```
      <form method="GET" .. />  not POST!
      ... various "input" and "select" statements
      </form>
```

If POST is used, most browsers silently drop the POST data if the request is redirected to another host:port, so developers would need to make sure that the URL has the correct host:port for the target agent. For some clients, PUT ("doPut(..)") can be used instead of POST:

```
        UrlConnection uc = ...;
        ((HttpUrlConnection) uc).setRequestMethod("PUT");
        OutputStream os = uc.getOutputStream();
        ...  // usual "upload-data" code
```

The exact redirection rules are in the HTTP specification and built into the browser/URLConnection code itself and the HTTP/1.0 specification (http://www.w3.org/Protocols/rfc2616/rfc2616-sec9.html). For additional details, see http://ppewww.ph.gla.ac.uk/~flavell/www/post-redirect.html.

JSPs are supported, but must be precompiled to class files. An example JSP page is:

```
<html><body>
Hello, world!
</body></html>
```

To run, a Tomcat 4.0.3 JSP runtime jar is required, e.g.:

```
  cp $TOMCAT_HOME/lib/jasper-runtime.jar $COUGAAR_INSTALL_PATH/sys
```

To compile:

```
# compile jsp to java
$TOMCAT_HOME/bin/jspc.sh Hello.jsp
# compile java to classes
javac -classpath $COUGAAR_INSTALL_PATH/sys/tomcat_40.jar\
;$COUGAAR_INSTALL_PATH/sys/jasper-runtime.jar\
;$COUGAAR_INSTALL_PATH/sys/servlet.jar \
  Hello.java
# create jar
jar cvf $COUGAAR_INSTALL_PATH/lib/hello.jar Hello.class
```

The generated "Hello.java" can be saved for future compilations, or generated by ANT or Makefile build scripts.

The ComponentServlet base class includes utility methods to obtain the local agent's name, "getEncodedAgentName()", and a "getServiceBroker()" method that can be used in the "load()" method to obtain services. For example, this servlet uses the BlackboardQueryService to print the UIDs of all UniqueObjects on the local agent's blackboard:

```
package foo;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.Collection;
import java.util.Iterator;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.cougaar.core.component.ComponentServlet;
import org.cougaar.core.service.BlackboardQueryService;
import org.cougaar.core.util.UniqueObject;
import org.cougaar.util.UnaryPredicate;

public class UniqueObjectCountServlet extends ComponentServlet {

  private static final UnaryPredicate MY_PRED =
    new UnaryPredicate() {
      public boolean execute(Object o) {
        return (o instanceof UniqueObject);
      }
    };

  private BlackboardQueryService bqs;

  protected String getPath() {
    return "/uniqObjCount";
  }

  public void load() {
    super.load();
    bqs = (BlackboardQueryService)
       getServiceBroker().getService(
         this, BlackboardQueryService.class, null);
  }

  public void doGet(
     HttpServletRequest req,
     HttpServletResponse res) throws IOException {
    PrintWriter out = res.getWriter();
    out.println("<html><body>");
    Collection c = bqs.query(MY_PRED);
    out.println("UniqueObjects["+c.size()+"]: <ol>");
    for (Iterator itr = c.iterator(); itr.hasNext(); ) {
      UniqueObject uo = (UniqueObject) itr.next();
      out.println("  <li>"+uo.getUID()+"</li>");
    }
    out.println("</ol></body></html>");
    out.flush();
    out.close();
  }
}
```

To modify the blackboard, the full "BlackboardService" is required, requiring the client to implement the BlackboardClient API and open transactions around operations, e.g.:

```
package foo;

import java.io.IOException;
import java.io.PrintWriter;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.cougaar.core.blackboard.BlackboardClient;
import org.cougaar.core.component.ComponentServlet;
import org.cougaar.core.service.BlackboardService;

public class AddObjServlet
  extends ComponentServlet
  implements BlackboardClient
{
  private BlackboardService bb;

  protected String getPath() {
    return "/addObj";
  }

  public void load() {
    super.load();
    bb = (BlackboardService)
       getServiceBroker().getService(
         this, BlackboardService.class, null);
  }

  public void doGet(
     HttpServletRequest req,
     HttpServletResponse res) throws IOException {
    PrintWriter out = res.getWriter();
    out.println("<html><body>");
    try {
      bb.openTransaction();
      bb.publishAdd(SOME_OBJECT);
    } finally {
      bb.closeTransactionDontReset();
    }
    out.println("Added object!");
    out.println("</body></html>");
    out.flush();
    out.close();
  }

  // oddities of implementing BlackboardClient
  public String getBlackboardClientName() {
    return getPath();
  }
  public long currentTimeMillis() {
    throw new UnsupportedOperationException("bug 2515");
  }
}
```

Note that one servlet instance is loaded per agent, so multiple clients may access the servlet at the same time. This is fine as long as per-request fields are kept separate, e.g. by creating a "handler" class to hold per-request state:

```
  ..
  public void doGet(
     HttpServletRequest req,
     HttpServletResponse res) throws IOException {
    // create a temporary handler
    (new MyHandler(req, res)).execute();
  }
  private class MyHandler {
    private final HttpServletRequest req;
    private final HttpServletResponse res;
    private PrintWriter out;
    public MyHandler(HttpServletRequest req, HttpServletResponse req) {
      this.req = req;
      this.res = res;
    }
    public void execute() throws IOException {
      out = res.getWriter();
      sayHi();
      out.flush();
      out.close();
    }
    private void sayHi() {
      // okay to access “out” here, since we’re sure that its our
      // handler’s stream
      out.print(“Hello, world!”);
    }
  }
```

6 LDM Structure and Usage

6.1 Overview

The Cougaar Logical Data Model (LDM) allows Plugin Developers to define Blackboard objects that represent property-based assets.

As noted in the Cougaar Architecture Document, an asset is a property-based object design model, as opposed to an inheritance-based hierarchical taxonomy. Please see the Cougaar Architecture Document for a design overview.

Assets have PropertyGroups that define the distinguishing characteristics of that asset. PropertyGroups are essentially data structures that are attached to assets. An example PropertyGroup is an "ItemIdentificationPG", which can be used to tag an asset instance with an instance identifier (e.g. a car license plate, or a person's social security number). Property groups are also used to designate behavior characteristics, such as a "LandVehiclePG" for a vehicle that can drive over land (e.g. a truck).

Assets are divided into abstract prototypes and specific instances. A prototype often contains the majority of the PropertyGroup data, such as a "PhysicalPG" that defines the default physical characteristics of the asset. An asset instance points to its asset prototype and adds instance-specific PropertyGroup overrides, such as the instance "ItemIdentificationPG". The use of prototypes helps reduce the number of objects that are required at runtime, since a single prototype can hold all the common data for many asset instances.

For example, a Truck prototype would define the typical PropertyGroups for all Trucks (mass, volume, cargo capacity, vehicle properties, etc). An asset instance of the Truck prototype would add the specific Truck's license plate number and perhaps any non-typical Truck properties (e.g. color).

Developers often implement some level of inheritance on top of the property-based asset model. This provides regularity in the normal properties of related things and simplifies PropertyGroup access. For example, a Vehicle subclass of Asset could define utility methods for easy access to the PropertyGroups that are always present on vehicles, such as "getCargoCapacityPG()". Of course, excessive use of inheritance defeats some of the advantages of a property-based design.

BehaviorGroups are wrapper classes for PropertyGroups that add utility methods for object-orientated behavior.

Aggregates are short-hand for a number of equivalent asset instances (e.g. 9 trucks).

6.2 Using Assets and PropertyGroups

Using Assets and PropertyGroups is fairly simple once the LDM asset classes and data feeds have been defined. This section will illustrate how to use assets, and later sections will cover how to develop new Asset and PropertyGroups, and how to fill them with domain-specific data.

Assets are constructed by the "planning" domain's PlanningFactory:

```
  PlanningFactory ldmf = (PlanningFactory)
myDomainService.getFactory("planning");
```

The name "PlanningFactory" is a misnomer, since this can be used to create assets for non-planning applications. This may be refactored in the future.

The PlanningFactory has methods to create assets prototypes, instances, aggregates, and property groups. See the javadocs for "planning/org.cougaar.planning.ldm.PlanningFactory" for additional details.

The typically usage pattern is to create a prototype and bind its type-identifier in the LDM. The type-identifier can be any unique string that uniquely identifies the prototype asset, and can also distinguish between multiple prototypes of the same class, such as Truck prototypes for "Ford_F150" vs. "Chevy_S10". Asset instances of the prototype can later be created by specifying the prototype type-identifier and an instance-identifier (e.g. "Ford_F150", "XYZ-123"). Raw assets can also be constructed but typically only used to support the above prototype and instance operations.

Prototypes must be constructed and cached in the LDM before instances can be created. The PrototypeRegisteryService is also required (org.cougaar.planning.service.PrototypeRegistryService). Here's an example:

```
  Asset proto = ldmf.createPrototype(Truck.class, "Ford_F150");
  protoRegService.cachePrototype("Ford_F150", proto);
```

Instances are created using either of two equivalent methods. If you already have a prototype asset object, you can use:

```
  Asset inst = ldmf.createInstance(proto, "XYZ-123");
```

Or, if you don't have the prototype object on hand, you can specify the type-identifier:

```
 Asset inst = ldmf.createInstance("Ford_F150", "XYZ-123");
```

Aggregates are constructed like instances, since they also refer to a prototype. Here's a group of 9 trucks:

```
  Asset agg =ldmf.createAggregate("Ford_F150", 9);
```

Raw assets are created as follows:

```
  Asset raw = ldmf.createAsset(Whatever.class);
```

As noted above, subclasses of Asset can be machine-generated to always include certain PropertyGroups, which would add methods like:

```
  FooPG getFooPG();
```

Additional PropertyGroups can be found at runtime by using:

```
  PropertyGroup pg = inst.searchForPropertyGroup(somePG.class);
```

and can be added, replaced, and removed at runtime by using these methods:

```
inst.addOtherPropertyGroup(pg);
inst.replaceOtherPropertyGroup(pg);
inst.addOtherPropertyGroup(pg);
inst.removeOtherPropertyGroup(pg);
inst.removeOtherPropertyGroup(somePG.class);
```

PropertyGroups can also be time-phased, such as a “SkiResortPG” that is only active for the winter months. Cougaar includes support for schedules with either overlapping or non-overlapping segments.

6.3 Defining new Assets and PropertyGroups

PropertyGroups are always defined by three related classes. For example, a "Foo" property group is defined by these three classes:

FooPG	the reader interface, with "get*" methods
NewFooPG	the writer interface, which extends the reader interface and adds "set*" methods
FooPGImpl	the implementation which implements both interfaces

As noted above, PropertyGroups contain detailed information about the asset, such as its physical characteristic constants (e.g. mass=1209.37KG, volume=43.21SQ_FEET, etc). These constants are filled into the PropertyGroup by one or more plugin PropertyProviders, which are described later in this section.

Late-binding PropertyGroups are PropertyGroups that are loaded upon first request rather than at asset construction time. For example, an asset's property group may include data that is defined in an external database. Rather than accessing this data when the asset is constructed, a late-binding PropertyGroup is filled by its LatePropertyProvider upon the first request for that PropertyGroup.

PropertyGroups and Assets are machine generated from a ".def" to Java code at compilation time. This ".def" file is found by the Cougaar ANT build script and parsed by the Cougaar Build "DEF" parser:

```
org.cougaar.tools.build.DefRunner
```

The ".def" file must start with a line that defines the ".def" file parser Java class:

```
 ;!generate: CLASSNAME
```

Property group ".def" files, typically named "blahprops.def", specify the PG DEF writer:

```
 org.cougaar.tools.build.PGWriter
```

Asset ".def" files, typically named "blahassets.def", specify the Asset DEF writer:

```
 org.cougaar.tools.build.AssetWriter
```

Additional DEF writers can be used, but the above are the most common.

6.3.1 Using the PropertyGroup Writer

Here's a clip from the "alpprops.def" file for the GLM's definition of military property groups.

```
;!generate: org.cougaar.tools.build.PGWriter
;<copywrite> ... Cougaar License ... </copywrite>

includedefs=org/cougaar/planning/ldm/asset/properties.def

package=org.cougaar.glm.ldm.asset

import= org.cougaar.glm.ldm.plan.*,\
     org.cougaar.glm.ldm.oplan.*,\
     org.cougaar.glm.ldm.policy.*,\
              org.cougaar.core.mts.MessageAddress,\
     org.cougaar.glm.execution.common.InventoryReport

factoryExtends=org.cougaar.planning.ldm.asset.PropertyGroupFactory

[AmmunitionPropertyGroups]
abstract=true

[AmmunitionPG]
extends=AmmunitionPropertyGroups
slots=String DODIC, \
     String lotID, \
     String DODAAC, \
     long quantity_oh_serv, \
     long quantity_oh_unserv
DODIC.doc=A string of the form "A123"
lotID.doc=An identifier for the manufacturers lot

[PhysicalAssetPropertyGroups]
abstract=true

[PackagePG]
extends=PhysicalAssetPropertyGroups
slots=long count_per_pack, \
     String unit_of_issue, \
     Distance pack_length, \
     Distance pack_width, \
     Distance pack_height, \
     Area pack_footprint_area, \
     Volume pack_volume, \
     Mass pack_mass
```

The above ".def" defines abstract property groups for ammunition and pacakges. An AmmunitionPG is defined with fields for military identifiers (DODIC, lotID, DODAAC) and quantities. Javadoc documentation is defined for the DODIC and lotID. A PackagePG is also defined with fields for the physical characteristics of a package.

We'll revisit these PropertyGroup examples in the AssetWriter section below.

The above will machine generate Java code for these new files:

org/cougaar/glm/ldm/asset/AmmunitionPG.java

org/cougaar/glm/ldm/asset/NewAmmunitionPG.java

org/cougaar/glm/ldm/asset/AmmunitionPGImpl.java

org/cougaar/glm/ldm/asset/PackagePG.java org/cougaar/glm/ldm/asset/NewPackagePG.java

org/cougaar/glm/ldm/asset/PackagePGImpl.java

A new PropertyGroup factory will also be machine generated:

org/cougaar/glm/ldm/asset/PropertyGroupFactory.java

This factory will have static constructor methods for all the PropertyGroups, such as:

```
  // brand-new instance factory
  public static NewAmmunitionPG newAmmunitionPG() {
    return new AmmunitionPGImpl();
  }
  // instance from prototype factory
  public static NewAmmunitionPG newAmmunitionPG(AmmunitionPG prototype) {
    return new AmmunitionPGImpl(prototype);
  }
```

This PropertyGroupFactory is often wrapped by the domain's Factory class, which is registered in the DomainService.

Note that the abstract property groups do not generate Java source. If the abstract property groups defined slots, they would appear in all the property groups that refer to them with an "extends" line.

BehaviorGroups are defined by using the "delegate=" entry. The delegate specifies a class that implements the PGDelegate interface. See the PGWriter and example GLM "alpprops.def" for additional documentation on BehaviorGroups ("BG"s).

Note that the machine-generated classes replace the slot "_" field name with Java-style capitalization. For example, the above PackagePG will have a "long getCountPerPack()" method.

6.3.2 Using the Asset Writer

Here is a clip from the GLM's "alpassets.def":

```
;!generate: org.cougaar.tools.build.AssetWriter
;<copywrite> ... Cougaar License ... </copywrite>

package=org.cougaar.glm.ldm.asset

propertydefs = alpprops.def

[ClassVAmmunition PhysicalAsset]
slots=AmmunitionPG, \
     PackagePG

[Explosive ClassVAmmunition]
```

The above ".def" file defines a "ClassVAmmunition" asset as a subclass of PhysicalAsset that adds PropertyGroups for AmmunitionPG and PackagePG. An "Explosive" asset is a subclass of ClassVAmmunition but doesn't add additional PropertyGroups.

The above ".def" will machine generate Java code for:

org/cougaar/glm/ldm/asset/ClassVAmmunition.java

org/cougaar/glm/ldm/asset/Explosive.java

The "PhysicalAsset" class is defined in "org.cougaar.planning.ldm.asset", in the ".def" file for the planning domain.

The ClassVAmmunition Java source will include utility methods for obtaining the specified PGs:

```
  public AmmunitionPG getAmmunitionPG() {..}
  public void setAmmunitionPG(PropertyGroup arg_AmmunitionPG) {..}
```

All assets also support dynamic PropertyGroup attachment. In this case, "getAmmunitionPG()" is equivalent to "searchForPropertyGroup(AmmunitionPG.class)".

A new Asset factory will also be machine generated:

org/cougaar/glm/ldm/asset/AssetFactory.java

BehaviorGroups are also defined in the PropertyGroup ".def" file. Here's an example of the ".def" entry:

```
        ;!generate: org.cougaar.tools.build.PGWriter
        ;; my properties.def file

        [MyPG]
        doc=A property group with standard and active slots
        slots=int x, String y, double z
        delegates=MyDelegate thing
        thing.delegate=int computeW(int inc);\
                       String prettyString();\
                       double checkAsset(Asset a);
```

and here's the corresponding hand-written delegate:

```
        // expect it to be Serialized!
        class MyDelegate implements PGDelegate {
          private MyPG myPG;
          // this constructor is called by the constructor of MyPG
          public MyDelegate(MyPG pg) {
            myPG = pg;
          }

          // here are the delegated-to methods
          public int computeW(int i) { return myPG.getX()+i; }
          public String prettyString() { return
"("+myPG.getX()+","+myPG.getY()+")"; }
          public double checkAsset(Asset a) {
            return myPG.getZ()*a.getPhysicalPG().getMass().getKilograms();
          }

          // implement PGDelegate
          public PGDelegate copy(PropertyGroup pg) {
            return new MyDelegate((MyPG)pg);
          }
        }
```

6.4 External Asset Data Feeds

The above sections discussed how to create Assets and define new PropertyGroups. This section covers the data-feed part of the solution, where the constants in the PropertyGroups are filled from a back-end data source.

For example, a Truck asset may have a PhysicalPG attached which defines slots for mass, volume, width, height, etc. These constants are filled in by the data feeds (PropertyProviders), which can obtain their data from many possible sources.

The following figure illustrates the interaction between end-user plugins on the left, assets, the PrototypeRegistryService in the middle, and LDM Prototype/Property Provider plugins as external data feeds:

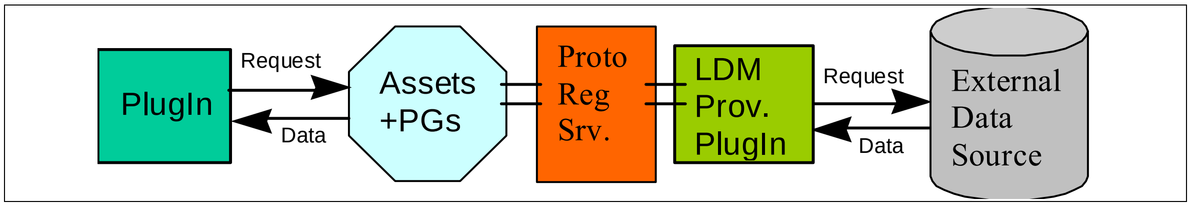


Figure 6-1: Logical Data Model and LDM Plugins

It is worth noting that the usual construction sequence of a prototypical asset follows this sequence:

1. A database Plugin (plugin1) decides it needs to create an "actual" asset of type T.

2. plugin1 calls LDM.getPrototype(T).

3. LDM looks in the prototype cache and fails to find anything matching T.

4. LDM invokes a PrototypeProviderPlugin (plugin2).

5. plugin2 constructs a new asset (proto1) of the right class.

6. plugin2 calls LDM.fillProperties(proto1).

7. LDM invokes a PropertyProviderPlugin (plugin3). In the figure above, plugin2 performs both roles.

8. plugin3 adds a PropertyGroup to the asset (proto1) and returns.

9. plugin2 calls LDM.cachePrototype(proto1).

10. plugin1 calls RootFactory.createInstance(proto1).

11. RootFactory creates an instance of asset (asset1) which delegates to proto1.

10. plugin1 adds asset1 to the Plan.

The prior sections covered steps (1), (2), (10), (11), and (12). The plugins covered in this section cover the remaining steps.

The PrototypeRegistryService allows a plugin to register as either a PrototypeProvider, PropertyProvider, or LatePropertyProvider:

```
  void addPrototypeProvider(PrototypeProvider prov);
  void addPropertyProvider(PropertyProvider prov);
  void addLatePropertyProvider(LatePropertyProvider lpp);
```

PrototypeProviders implement this method:

```
  // get a prototype by type-identifier
  // ignore the "asset-class-hint", which is null in practice
  // return null if this provider doesn't support this type-id
  Asset getPrototype(String aTypeName, Class anAssetClassHint);
```

PropertyProviders implement this method:

```
  // fill in all the PropertyGroup data
  // ignore if this provider doesn't support this type of asset
  void fillProperties(Asset anAsset);
```

LatePropertyProviders implement these methods:

```
  // get a list of the PropertyGroups that can be provided
  Collection getPropertyGroupsProvided();

  // fill a PropertyGroup that has been requested now
  // return null if this provider doesn't provide this pg
  PropertyGroup fillPropertyGroup(Asset anAsset, Class pg, long time);
```

A single class can implement all three interfaces and register with the three corresponding PrototypeRegistryService methods.

As noted above, LatePropertyProviders are just like regular PropertyProviders, except that the PropertyGroup is filled upon first demand instead of when the asset is first constructed. This allows a LatePropertyProvider to delay a potentially expensive data fetch until the data is actually required.

Future_PGs are used by PropertyProviders and are similar to LatePropertyProviders. Instead of returning a regular PropertyGroup when "fillProperties(..)" is called, the PropertyProvider can return a Future_PG. A Future_PG is a "promise" to provide a value by the time it is requested by a consumer. It is a statement that retrieval of the values are in-progress and any Plugins requesting the value will block until the value is actually available. LDM Plugins may emit Future_PGs instead of concrete PGs so that they can batch up many queries to a high-latency DB rather than executing many separate requests. An LDM Plugin of this sort would note each request from the LDM, filling in the Assets with Future_PGs, and adding the requests to a queue. Another thread would run, taking any unfulfilled requests from the job queue and executing a batch DB query. When the query returns, the results are used to replace the matching Futures with actual PropertyGroup instances.

Here's an example:

```
        // Plugin PropertyProvider code:
        PropertyGroup fpg = new PhysicalPG.Future();
        asset.setPhysicalPG(fpg);
        startQuery(asset,fpg);  // insert in a thread queue

        // when the query returns
        PropertyGroup rpg = new FooPG();
        // ... set the values
        asset.setFooPG(rpg);  // replace the future
        fpg.finalize(rpg);    // forward the future to the real PG
```

6.4.1 Using your own Data Feeds

The PrototypeRegistryService allows any plugin to register as either a PrototypeProvider, PropertyProvider, LatePropertyProvider, or all three. The LDM infrastructure will call back into the plugin code when it must obtain the necessary data to create an Asset or fill a PropertyGroup.

New Cougaar developers may find it easier to write their own LDM Data Feeds than to use the example SQL and XML data feeds.

6.4.2 SQL Database Data Feeds

The SQL Database Asset Data Plugins use SQL, defined in a ".q" data file, to communicate over JDBC to an external database. Java Reflection is used to construct classes and invoke PropertyGroup methods.

The LDMSQLPlugin supports connections to a variety of databases such as Oracle, MySQL and PostgreSQL via a JDBC interface. The database parameter values are defined in the cougaar.rc file as seen in the example below. During execution, the database driver that is listed in the database URL (e.g., jdbc:**mysql**://localhost/database01) will be loaded.

Example from a cougaar.rc file

```
#JDBC driver list - one or many may be listed
driver.mysql=org.gjt.mm.mysql.Driver
driver.oracle=oracle.jdbc.driver.OracleDriver
# Database connection parameters
org.cougaar.database=jdbc:mysql://localhost/database01
org.cougaar.database.user=user
org.cougaar.database.password=user password
```

The Plugin expects at least one Plugin parameter (via `PluginAdapter.getParameters`). The first parameter is interpreted as the name of a query file. We look for this using the ConfigFinder (see class for details). All other parameters are interpreted as query parameter settings.

```
Plugin=org.cougaar.mlm.plugin.ldm.LDMSQLPlugin(foo.q, NSN=12345669)
```

The query file contains the parameters for connecting to the database and a list of queries to execute on behalf of the Plugin. The values for the database parameters are substituted with values defined in the cougaar.rc file. Comment lines (lines starting with “#”) and empty lines are ignored. Lines starting with “%” indicate the start of a “query section” and the name of the class that will handle the results of the queries. All other lines are of the form “parameter=value” where the parameter values are made available to the current query. *See org.cougar.util.DBProperties.java and org.cougaar.util.Parameters for details on parameter substitution and mapping.*

Because database query languages can differ in syntax, the query file can be formatted to house multiple database-specific queries. You can define a default query "query = select …", intended to work with any database. If the query syntax is unique to a particular database, you must define the query as "query.*dbtype* = select." Please note that the dbtype value must match the database type reference defined in the database URL, "jdbc:**mysql**://localhost/database01." The database-specific query, if one exists, will be executed over the default query. For another query-based LDM Plugin example, see org.cougaar.glm.ldm.QueryLDMPlugin

Example of default query and database-specific queries in a " .q" file

```
query = select NSN, SUM(QTY_OH) AS "SUM(QTY_OH)", NOMENCLATURE from
jtav_equipment         where  UIC4 = substr(:uic, 1, 4) and
NSN in (:pacing)  group by NSN, NOMENCLATURE

query.mysql = select NSN, SUM(QTY_OH), NOMENCLATURE from jtav_equipment
 where  UIC4 = substring(:uic, 1, 4) and
 NSN in (:pacing)  group by NSN, NOMENCLATURE
```

Example ".q" File for Instance Creation:

```
Database = ${org.cougaar.database}
Username = ${org.cougaar.database.user}
Password = ${org.cougaar.database.password}

# First, get the personnel and generate an aggregate asset
%SQLAggregateAssetCreator
query = select 'Personnel' AS NSN, personnel AS QTY_OH, 'MilitaryPersonnel' AS
NOMENCLATURE \
     from ue_summary_mtmc \
     where uic = :uic

# Now, get 'eaches' for all 'pacing' assets from JTAV
%SQLAssetCreator
query = select NSN, SUM(QTY_OH) AS "SUM(QTY_OH)", NOMENCLATURE from
jtav_equipment \
where  UIC4 = substr(:uic, 1, 4) and \
  NSN in (:pacing) \
  group by NSN, NOMENCLATURE

query.mysql = select NSN, SUM(QTY_OH), NOMENCLATURE from jtav_equipment \
where  UIC4 = substring(:uic, 1, 4) and \
  NSN in (:pacing) \
  group by NSN, NOMENCLATURE
```

Example ".q" file using a QueryHandler class:

```
# get Oplan info
%OplanQueryHandler
OplanInfoQuery = select operation_name, priority, c0_date from oplan where
oplan_id = ':oplanid'
```

6.4.3 XML Data Feeds

The XML Asset Data Plugins read flat-files containing XML text. The XML is parsed using a DOM and Java Reflection is used to construct the Assets and PropertyGroups.

This LDM Plugin follows the same model as the LDMSQLPlugin, in that it is a one-time Plugin – it runs once, populating the Agent with LDMObjects, then is dormant. It gets the information for the LDMObjects from an XML file (*.ldm.xml). This Plugin expects only one parameter (via PluginAdapter.getParameters). This parameter is interpreted as the name of the .ldm.xml file. This file is found using ConfigFileFinder (see org.cougaar.core.util.ConfigFileFinder for details).

7 Notes on Persistence

7.1 Introduction

The Cougaar persistence infrastructure has been designed to have a minimal impact on Plugin developers. However, Plugin developers need to be aware of some aspects of persistence to insure correct behavior and they need to be aware of one additional aspect in order to avoid poor performance. Advice to plugin developers is given below.

Additionally, adding media-specific persistence plugins (not to be confused with domain plugins) can customize the persistence configuration. Persistence plugins adapt arbitrary storage media so it can be used for storing persistence data. Current media plugins include files and databases. The configuration of available persistence media is currently achieved through setting system properties. This is discussed below as is the persistence plugin API.

The operation of persistence is subject to control by the adaptivity engine through the `PersistenceControlService`. Most such controls apply to media plugins and are discussed below.

7.2 Avoid Redundant Initialization

Many Plugins make the mistake of assuming that when setupSubscriptions is invoked they must perform all sorts of initialization. Plugins may call `didRehydrate()` to determine if this is a cold start or a restart. Specifically, `didRehydrate()` returns true if the first transaction of (the Subscriber of) the plugin had been closed and the results persisted. It says absolutely nothing about objects that might have been published after the first transaction. It is often safest to simply query the blackboard to discover if certain objects are present and publishing only if they are not present. By the time a plugin's `setupSubscription()` method is called all rehydration has completed so this is perfectly safe thing to do.

A related issue concerns the private state of the Plugin. Plugins should be stateless – they should rely on the blackboard for state. Any state a Plugin maintains in variables should be regarded as a cached or memo-ized version of blackboard. This is the only way that the state can be preserved through a restart. In some cases, private state objects must be created and published to preserve state through a restart. Plugin developers must be parsimonious in creating these private state objects. Every object that is added, changed, or removed will require writing the new value to persistent storage. For example a hash table with a few thousand entries that is changed every execution cycle is going to bring the agent to its knees. On the flip side, if you must maintain private state, and that state object is modified, you must publish a change for it. If you don't, it *will* be incorrect after a restart.

7.3 Blackboard Objects Must Be Serializable

The information in the blackboard is saved in files by serializing the blackboard objects (and the objects they refer to). The most obvious way in which this rule is violated is to simply not declare an object class Serializable. Unfortunately, the check for serializability is a runtime check and cannot be performed by the compiler. Plugin developers should be careful to make all objects referenced by the blackboard serializable.

A common way for the rule to be violated is to create an object that is an inner class of a Plugin (or support class). Instances of inner classes have invisible references to the outer class and usually cannot be serialized because the outer class is not serializable. Even if the outer class implemented serializable, it would probably be a mistake to drag it along into the persistent state. This occurs most commonly in predicates that are defined using anonymous inner classes. It can also occur in creating triggers.

So, be wary of instances of inner classes that are used to create objects that are published or contained in objects that are published. They must always be "static" classes and can never be anonymous unless they are created in a static method or in the initialization of a static variable. If you see an anonymous class without the word "static" nearby, watch out!

7.4 Should Plugin Subscriptions Be Persisted?

There are two kinds of Plugins: Plugins that react to individual plan elements and those that react to a subset of the plan. For example, most expanders react to individual plan elements – a task appears and an expansion is created with a workflow and populated with subtasks. On the other hand, Plugins such as the debug UI display all plan elements of certain types.

The distinction becomes more obvious when we consider what happens if an Agent is restored from persisted state (a process that has come to be known as rehydration). For Plugins that react to individual plan elements, the objects that are restored to the plan during rehydration must not activate the Plugin – the Plugin has already reacted. On the other hand, Plugins that are dealing with a complete subset of plan objects do need to be activated otherwise they will never hear about these objects.

The distinction between these two cases is established during the setup of subscriptions. The default is for Subscribers to save the state of the subscriptions as part of the persistent state. After rehydration, the Plugin will not be informed about all the plan objects; it will only be informed about new objects or changes or removals. Plugins that want to be informed about all objects should call:

```
BlackboardService.setShouldBePersisted(false)
```

causing the subscriber state to not be saved persistently. If the subscriber state is not persisted, then all the existing objects in the plan will be added to the subscriber's subscriptions when the subscription is created and may be created during the first execution of the Plugin's execute method. Care should be taken to not process the objects twice. Immediately after calling subscribe(), the existing objects will have been added to the subscription. The subscription will be marked as having changes. If the transaction during which the subscription is created is not reset when closed, then the changes will still be present for the next transaction. This means that setupSubscriptions should *not* process the objects unless it also resets the subscription (resetChanges).

Plugins that seem to need both kinds of Subscriber may need to create a second Subscriber to handle the other type of subscription.

7.5 Should Blackboard Objects Be Persisted

The persistence of blackboard objects is optional. In some cases, it is far easier to recreate an object than to deal with a rehydrated one. There are two ways to mark an object so it won't be persisted: The object can implement the `NotPersistable` interface or it can implement the `Publishable` interface. The former is merely a marker interface; all such objects will not be persisted. The latter has a method returning a boolean that indicates the particular instance should not be persisted.

7.6 Naming Subscribers

Most Plugins don't need to worry about naming subscribers. The exceptions are Plugins with multiple subscribers and Plugins that are instantiated multiple times in a given agent. All Plugins have at least one subscriber. Subscribers have names that allow their persisted state to be matched up with Plugins after rehydration. The name consists of two parts: the Plugin name and the subscriber name. By default the Plugin is named with its class name and its parameter values and the subscriber is named with the empty string. This arrangement relieves most Plugins from the burden of creating unique names for subscribers. However, a Plugin requiring multiple subscribers must insure that its secondary subscribers have unique names that are not equal to the empty string. These names are only important for subscribers for which "shouldBePersisted()" is true.

In the very rare case that multiple instances of a particular Plugin class are instantiated in an Agent and the Plugins are not parameterized in a way that allows the instances to have distinct default names, the Plugin must take the responsibility of supplying the Plugin name. The Plugin should override the getSubscriptionClientName method in the SubscriptionClient interface. This method should return a name that is distinct from the names of all other Plugins in the agent. An alternative is to add a dummy parameter (in the Plugin line of the agent XML file) that is not otherwise used by the Plugin.

7.7 Performance Issues

Persistence exacts a performance penalty. The persistence mechanism insures that all the changes to an Agent are safely saved away before any other agent acts upon those changes. This saved state consists of all the changes to the plan as well as the state of communications. Most of this is out of the Plugin developer's hands – you have to do what you have to do. About the only things that a Plugin developer can do is to avoid private state that is saved in the plan and to be judicious in altering the plan.

7.8 Persistence Plugin API

Persistence data can be stored in various ways on different media. This is achieved by adding media-specific plugins to the generic persistence mechanism. The media plugins do not need to be aware of the intricacies of taking persistence snapshots or rehydrating an agent from persistence data, they need only be aware of how to store and retrieve the relevant snapshot data. The javadocs for the `PersistencePlugin` API give the details needed to write the interface. Here we describe the key concepts.

Each persistence plugin instance has a name assigned to it by the configuration data. The plugin is also supplied with initialization parameters from the configuration data that it may use as it sees fit. Obviously, the configuration data for a particular plugin must match what that plugin expects.

Each plugin is used exclusively for generating a "rehydration set" before persistence switches to a different medium. A rehydration set consists of a full snapshot followed by a series of incremental snapshots and can be used to rehydrate an agent to the state that it was in as of the last incremental snapshot in the rehydration set. Each set has a timestamp corresponding to the time of the last incremental snapshot. The media plugin must implement a mechanism to keep track of such rehydration sets and be able to return all available sets upon request.

The primary interface methods furnish input and output streams to be used to rehydrate or persist the agent's state.

The API also includes methods that the plugin can used to add additional adaptive controls. These methods furnish the names of the controls, the allowed values, and accept a new value for a particular control. These controls are in additional to the standard, per-medium, controls used by the base persistence code.

7.9 Persistence Configuration

The configuration of the media plugins is specified through system properties. The first property is:

```
-Dorg.cougaar.core.persistence.class=<item>,…<item>
```

Each item in the list gives the name of the class of the plugin, its name, and its initialization parameters all separated by semi-colons. The same class can be used for multiple plugins, but all names must be distinct. The parameters must match what the plugin's `init` method expects.

Examples:

-Dorg.cougaar.core.persist.class=org.cougaar.core.persist.FilePersistence\;P
-Dorg.cougaar.core.persist.class=org.cougaar.core.persist.DummyPersistence\;dummy

Multiple plugins can be specified with the "," separator, e.g.:

-Dorg.cougaar.core.persist.class=Alpha\;a1\;a2\;a3,Beta\;b1\;b2

where plugin Alpha is passed [a1, a2, a3] and Beta is passed [b1, b2].

7.10 Persistence Control Service

The persistence control service exposes the adaptive controls of the persistence mechanism and its plugins. Normally, this service is only used by the `PersistenceControlPlugin` which creates an OperatingMode for each control. Currently, there are 4 controls for each plugin. They are:

- interval – controls the average interval between uses of the medium
- encryption – enables encryption for this medium
- signing – enables data signing for this medium
- consolidationPeriod – specifies the number of incremental snapshots between full snapshots

7.11 Guidelines on Persistence and Impact on Agent Mobility

It's important for components to be persistable and to correctly rehydrate. This is crucial both for agent restarts after JVM crashes and for agent mobility.

Below is a summary of the issues and some simple tests that ***ALL*** Cougaar developers should run.

7.11.1 Definitions

- *Agent restart* is when a dead (not running) agent is recreated on either the same or a different node, using a prior state snapshot if it is available.
- *Persistence* is the infrastructure mechanism for taking periodic snapshots of the blackboard contents and saving these snapshots to the disk or other storage targets. All blackboard objects must be serializable, and plugins must save their state in the blackboard or be able to recreate it from scratch. **Note** that the plugin instances, services, and threads are not captured in the persisted state. Persistence must be enabled by a system property, and the default storage target is the local filesystem.

- *Rehydration* is the process of restarting an agent. It includes locating the most recent persistence snapshot, recreation of the blackboard, refilling the subscriptions, and reconciliation with other agents. Ideally the snapshot was taken just prior to the agent's death, but components must support partial or complete loss of these snapshots.

- *Lazy persistence* is the persistence mode that cougaar typically uses. It takes a snapshot every couple minutes or so, and doesn't block agent messaging. Lazy persistence risks some loss of state between snapshots, which means that some work may need to be reprocessed after restarts. However, given that crashes are typically rare, lazy persistence obtains better overall performance and is more robust to state loss.

- *Reconciliation* is the agent restart processing that is performed by lazy persistence to compensate for the periodic snapshots. Agents communicate between one another to fill in state that was lost between the time of the snapshot and the agent death. Reconciliation and agent restart detection is done in the infrastructure (`SimpleAgent`'s "`checkRestarts()`") and Logic Providers. Plugins and other components typically don't need to be involved, except for the persistence and rehydration requirements of their own agent, as described below.

- *Agent mobility* is the infrastructure mechanism to transfer a running agent from one node to another node. Mobility uses persistence to capture the state, but unlike lazy persistence, mobility captures the state just prior to the move to avoid loss of state and (in 9.4) reconciliation. Unlike regular persistence, components must unload and release all memory and threads to prevent resource leaks, since the node and JVM are not killed when an agent moves. Note that nodes and node-agents are not movable.

- *Forced move-based restart* is the use of agent mobility to restart an agent on its current node. Typically a move request to the current node would be a trivial success, but in this case the full restart is forced. As noted above, nodes and node-agents are not mobile, so node restart testing should use JVM-kills instead of mobility.

7.11.2 Failure Examples

Failure to support persistence or unloading can produce runtime bugs and a major or subtle loss of capabilities. Here are 5 examples that illustrate various failures:

1. Suppose a plugin keeps an instance variable that is a count of the number of Tasks it has processed. Since this variable is not in the blackboard, it won't be captured in the persistence snapshots. Upon agent restart the counter is lost, which might result in a runtime exception, or worse: a very subtle runtime bug.

   The fix is for the plugin to keep a blackboard object with this counter, and always "`publishChange`" this object when the counter is incremented, or to somehow recreate this counter based upon the existing blackboard contents (i.e. the counter is really a cache).

   This is the cause for most rehydration errors. The worst aspect of this bug is that it remains hidden until persistence or mobility are tested, at which time the agent is rehydrated but no longer functions.

   Cougaar developers should build persistence support into their plugins as early in their design as possible and run the tests that are listed later in this document.

   This bug is both a persistence bug, since the state was not captured, and a rehydration bug, since the plugin didn't attempt to recreate its cache within "`setupSubscriptions()`".

2. Suppose that a blackboard object is not serializable. Persistence will be unable to take a snapshot of the blackboard, resulting in a runtime exception. Note that the message transport doesn't serialize objects that are transferred between agents on the same node, so this problem can also creep in if formerly co-located agents are moved to separate nodes!

   All blackboard objects must be serializable, or marked as non-persistent by overriding the `Publishable` API's "`boolean isPersistable()`" method.

3. Suppose that a plugin supports persistence and rehydration, but upon rehydration it rushes to redo lots of processing. This may overload the system, consume too much memory, and accidentally kill the JVM.

   Agents should be careful not to overload the system when rehydrating. In some cases plugins may need to buffer the work and use alarms. Note that this is a general problem of work overload that may be exacerbated by rehydration.

4. Suppose that a plugin doesn't release all the resource in its "`unload()`" it has acquired in its "`load()`" or during execution. These resources include:

   • Services
   • Back-pointers from service providers (e.g. a blackboard subscription listener)
   • Objects referenced by static fields
   • Popup UIs
   • Threads and Timers
   • I/O and database connections

   When the agent is moved these resources will not be returned to the JVM, resulting in resource leaks.

   Also watch out for service revocation listeners (bugs 725 and 1549).

   All components must support the "`suspend()`" and "`unload()`" operations. In some cases a profiler (e.g. "OptimizeIt") must be used to check for resource leaks.

5. Suppose that a plugin communicates with a database to modify the database contents, or interacts with any other external system (writes files, RMI, etc). Persistence is only periodic and "lazy", so upon agent restart these external resources can be out of sync with the rehydrated agent's state.

   Cougaar blackboards include built-in reconciliation support, which is covered at the end of this section.

   If one is lucky and the persistence snapshots coincide with the external activities, nothing will be lost. Since a node may crash at any moment, due to power loss or other actions beyond the node's control, this clean external synchronization may not be possible.

   To fix these interactions with external systems, your code needs to perform its own reconciliation procedure. It must identify what was previously planned but will no longer be done, undo what will not be done, and recreate actions (e.g. new Tasks) to correspond with plans that should be kept but were lost. This can be very tricky to do correctly.

   Some implementations may discard all external plans to keep things simple, which may be wasteful.

7.11.3 Tests

Below are tests that developers can use to help verify that their components can support persistence and agent mobility. First, some Cougaar limitations to avoid:

- The node running the name server should not be killed, since the name server state will be lost and reconciliation will not occur. This will not be fixed in Cougaar 9.4, but in a later release TBA (10.0+).

- Not all agents and plugins support persistence. In particular, the inventory management agents are not persistable or movable, due to state-capture bugs.

If you're writing plugins that communicate between agents, make sure that the agents can run when split up between multiple nodes. This will detect message serialization bugs, memory usage bugs, and any dependencies upon local resource (files, system properties, statics, etc).

For example, given 2 nodes and 6 agents named "a" through "f", a full split requires `log2(6) =3` trials:

```
[[[a, b, c, d], [e, f]],  // first trial
 [[a, b, e, f], [c, d]],  // second trial
 [[a, c, e], [b, d, f]]]  // final trial
```

Here's a rough proof of how N nodes can split A agents using only `logN(A)` trials:

Assume N = 2, so we can use simple binary reasoning. Create a trial for each of the log2(A) bits and, for each trial *i*, split the list into one of two groups ("0" and "1") based upon the bit at index *i*. For any two indices in the list, their binary representation differs by at least one bit *x*, so they will be split into different groups on trial *x*. A similar proof can be made for any value of N by using base-N arithmetic.

Here's some example Java code that can be used to help create your XML files:

```
/** Given a number of n groups and a list of names,
 *  create a list of trials, where each trial is
 *  a split of the original list into n sublists.
 */
  public static List compute(int n, List a) {
    // validate
    if ((n < 2) || (a == null) || (a.isEmpty()))
      throw new IllegalArgumentException("n: "+n+", a: "+a);
    // compute max digits in base n
    int m = (int) Math.ceil(Math.log(a.size()) / Math.log(n));
    if (m == 0) m = 1;
    // alloc results list of size [m][m][m]
    List ret = new ArrayList(m);
    for (int i = 0; i < m; i++) {
      List x = new ArrayList(m);
      ret.add(x);
      for (int j = 0; j < m; j++)
        x.add(new ArrayList(m)); // average size is m, max is n
    }
    // split based upon values in base n
    for (int i = 0; i < a.size(); i++) {
      String ai = (String) a.get(i);
      for (int j = m - 1, t = i; j >= 0; j--, t /= n) {
        ((List) ((List) ret.get(j)).get(t % n)).add(ai);
      }
    }
    return ret;
  }
```

Persistence can be enabled with a system property:

```
  -Dorg.cougaar.core.persistence.enable=true
```

When running the node you'll see occasional "P" characters printed to standard-out, indicating that persistence is taking place.

To test, simply kill the node with a CTRL-C, and start the node again.

Note: Don't kill the node running the name server, since in Cougaar 9.X it does not support persistence. The name server is launched on the first node to start on the machine specified by the "configs/common/alpreg.ini" file (default is localhost), or as specified through the "-Dorg.cougaar.name.server=.." system property.

Use the "/tasks" servlet to make sure that the agents are recreated and the blackboard is rehydrated.

The real test is to submit additional work for your plugins to do, to make sure that they have restarted correctly -- recall the above "plugin counter" example listed above.

The first test should be:

1. send GLS
2. wait until processing is complete
3. wait for a bunch of "P"s
4. kill all nodes
5. restart all nodes
6. rescind GLS

Additional tests should include:

- Submit additional work for the agents to do after restarting. In some societies the GLMStimulator servlet can be used to inject new Tasks.
- Kill a subset of the nodes.
- Kill a node in between the "P"s, since a crash could happen at any time.
- Kill and restart a node while the GLS processing is in progress.
- Remove the persistence snapshots for one or more agents. The snapshots are typically saved in the "$COUGAAR_WORKSPACE/P", where COUGAAR_WORKSPACE defaults to "$COUGAAR_INSTALL_PATH/workspace". Restart the node and make sure that your agents have rehydrated even if state is lost.

Clearly, all permutations of the above tests would take a long time. Try a few tests at random to flush out the easier bugs. Focus on scenarios that your application might encounter.

Testing of state loss and reconciliation requires some level of JVM-kill testing. Mobility testing is somewhat easier, but mobility uses a clean persistence snapshot to avoid the reconciliation overhead. This means that some agent may be mobile but incapable of restarting after a node crash, so be sure to test both scenarios. Additionally, nodes and node-agents are not movable, so a JVM-kill must be used to test the persistence and rehydration capabilities of the node and node-agent.

Also note that reconciliation behaves differently for the sender and receiver of Tasks, which means that the restart of the root node of task processing (e.g. NCA) is different than the restart of a leaf agent (e.g. ConusGround).

Agent mobility supports an in-place "forced" move of an agent, which does:

- capture the agent state using an in-memory persistence snapshot
- unload and kill the agent
- rehydrate the agent from the in-memory snapshot

The restart can be requested from either the "/move" servlet or from a plugin.

Mobility support must be explicitly added to the agents XML configuration:

- Persistence does not need to be enabled.
- The mobility domain can be loaded by added this line to the "LDMDomains.ini" file, which is located in the "$COUGAAR_INSTALL_PATH/configs/common" directory:

  ```
  mobility=org.cougaar.core.mobility.ldm.MobilityDomain
  ```

- Add this plugin to all nodes:

  ```
  <component class="org.cougaar.core.mobility.service.RootMobilityPlugin"/>
  ```

- The following plugins should be loaded in the agent to be restarted, and can be safely loaded into all agents:

  ```
  <component class="org.cougaar.core.mobility.service.RedirectMovePlugin"/>
  <component class="org.cougaar.core.mobility.servlet.MoveAgentServlet"/>
  ```

  The logging level can be optionally be adjusted to capture additional mobility debugging output. This system property specifies the logging configuration file, which can be named "$COUGAAR_INSTALL_PATH/configs/common/log.props":

  ```
  -Dorg.cougaar.core.logging.config.filename=log.props
  ```

  and the file contents below will enable verbose output:

  ```
  log4j.rootCategory=WARN,A1
  log4j.appender.A1=org.apache.log4j.ConsoleAppender
  log4j.appender.A1.layout=org.apache.log4j.PatternLayout
  log4j.appender.A1.layout.ConversionPattern==%d{ABSOLUTE} %-5p - %c{1}
  - %m%n
  log4j.category.org.cougaar.core.mobility=DEBUG
  ```

  See the "$COUGAAR_INSTALL_PATH/doc/details/Logging.html" as well as the CDG sections 5.2.16 and 8.5.5 for additional logger configuration options.

Suppose that you want to test agent restart on agent "X". Here is the test procedure:

- Follow the above instructions to load the three plugins and domain.

- Start the society.
- Assuming that a node is running on localhost port 8800, use a browser to load this URL:

  http://localhost:8800/$X/topology

  The topology should list agent X on it's starting node. Write down the incarnation and move-id numbers.
- Access this URL:

  http://localhost:8800/$X/move

  An HTML form is presented to submit a new move request. Fill in the details as:

Mobile Agent	X
Origin Node	
Destination Node	
Force Restart	true

  Note the "true" selection for the force restart option. By default the restart will produce a trivial "success" response, since the agent is already at it's current node.

  Select "Add", and the move-based restart is submitted.
- The "/move" servlet will refresh and display a table of pending moves, where the current move request will have a *null* status. Select the "Refresh" button (***not*** Reload!) to update the page. Eventually the table will show the status as non-null, with either a success or failure value.
- Access the "/agents" servlet, and make sure that the agent is still listed in the "list all" view.
- Access the "/topology" servlet again to verify that the agent has been reloaded, that the incarnation number is the same, and that the move-id number is now larger than the prior move-id number.
- Submit additional work for the agent to process, to verify that the plugins have rehydrated correctly. The GLM-Stimulator-Servlet can be used to submit additional transportation tasks. Depending upon your application, a custom tool may be required.

Also try the above test from a third-party agent, which also requires the above plugins and mobility domain. For example, from agent X, request that agent Y be restarted in place.

In 9.4 there will be some mobility-related bug fixes and a scriptable mobility test rig. For 9.2 the above servlet can be used.

Also use the "/move" servlet to move multiple agents to and from nodes. The configuration requirements to load the "/move" servlet, mobility domain, and MoveAgentPlugin are the same as listed above.

Usage is similar to the above test. Fill in a non-blank destination node in the "/move" servlet form.

A good test is to create an empty node and move all agents from an existing node to that empty node. The "/agents" and "/topology" servlets should show the original node as empty (see doc/details/Servlet.html). A profiler, such as "OptimizeIt", should show the original node as mostly empty, with little memory and threads in use. Kill the original node, submit addition work to the agents, and verify that the moved agents still work.

7.11.4 Details

Here's an in-depth summary of how lazy persistence and reconciliation work:

- *Lazy Persistence*

  Cougaar persistence has been modified to operate in an optimistic or "lazy" mode. This mode is optimistic in that it assumes that restarts will not occur and that the persisted data is not absolutely essential. By making these two assumptions, the state of the agent can be saved much less frequently. This has the benefit that the objects to be saved will have had an opportunity to evolve through several stages and only their most recent stage will need to be saved. Lazy mode operates in the same way as non-lazy (or pessimistic) mode except the interactions with other agents are not rigidly locked into step with persistence. Instead, the states of an agent and the agents with which it interacts are allowed to progress without absolute assurance that the current state can be recovered in the event of a restart. The obvious upshot of this is that, if an agent is restarted, it will be in a state that is inconsistent with the states of the other agents with which it interacts. This inconsistency must be reconciled or else incorrect operation will ensue.

- *Reconciliation Rationale*

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

  These shared objects have a well-defined life cycle: they are created in an initial state, progress through a number of intermediate states and then are destroyed.

  The destruction of a Task actually can signify two things: the plan has changed and the Task is no longer desired *OR* and time has progressed and the Task is no longer relevant. In the latter case, the agents should have already permanently factored the effect of the deleted task into their planning. The Task deletion process marks Tasks to distinguish these cases.

  Note that there is *NO* semantic significance to the events that signal this progression, but the events do allow the agents to perform incremental updates. Since there is no significance to the path by which an object arrived in its current state (including its non-existence), the resynchronization process is not tasked with detailing these intermediate states; it need only insure that the slave object have the same state as the master.

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

  State changes sent just prior to the restart were not delivered.
  State changes received just prior to the restart were not persisted.
  The restarted agent has reverted to an earlier state of the object.

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

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

- *Reconciliation Procedure*

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

The incarnation numbers are saved in the topology service, which is backed by the global white pages service. Agents increment their incarnation number every time the agent is restarted, *except* for agent mobility, which keeps the prior incarnation number. A secondard "move number" is kept to track agent movement, which is initialized to the incarnation number when the agent is first created and incremented per move.

These incarnation numbers are available from the TopologyReaderService and the "/topology" servlet, but are primarily for infrastructure use only.

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

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

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

8 Adaptivity Engine

The adaptivity engine is responsible for setting the operating modes of a number of sub-components based on conditions within a component. For example, the adaptivity engine in an agent sets the operating modes of the plugins in the agent. The adaptivity engine of a community sets some of the operating modes of the agents in the community.

8.1 Terminology

We define a number of terms that will occur repeatedly throughout this section.

AEViewerServlet	A servlet that shows a user the Conditions, OperatingModes, and OperatingModePolicies in an agent. OperatingModes and OperatingModePolicies can be modified with this servlet, but Conditions cannot.
Adaptivity Engine	The term “adaptivity engine” is used both to refer to the ensemble of components, procedures, and data used to perform the adaptivity function as well as the specific plugin that responds to changes in conditions and the playbook and re-evaluates the plays in the playbook to set new operating modes.
Condition	A condition specifies a value that can be tested by the adaptivity engine. Conditions can arise in a number of ways: sensors can publish conditions to reflect a value that the sensor has measured, logic providers and plugins can publish conditions received from other agents, and conditions can be created by the adaptivity engine itself to hold intermediate values. In addition to its current value, a condition specifies a list of ranges of values that it will ever have.
Operating Mode	An operating mode is created and published by a component (for example, a plugin) so that the adaptivity engine can control the component. An operating mode has a list of ranges of values that it is allowed to have as well as a current value. Values must be numbers or strings.
Operating Mode Policy	A specific kind of policy that specifies constraints on operating modes. An operating mode policy behaves in much the same way as a play except that when there is a conflict between a policy and a play, the policy prevails. Operating mode policies can arise through the actions of managers of other kinds of policies or may originate in other agents (inter-agent operating mode policies). Higher-level adaptivity engines frequently produce the latter.
OperatingModePolicy Manager	A plugin that uses the PlaybookManager to constrain the Playbook with OperatingModePolicies. In some limited circumstances, the OMPM will set OperatingModes directly based on anOperatingModePolicy before calling the PlaybookManager.
Play	A play specifies restrictions or constraints on one or more operating modes and the conditions under which those constraints are to be applied.
Playbook	A playbook has a list of plays that are tested in succession for applicability to the current conditions. Plays that pass the test are then executed to affect the operating modes of sub-components or to establish conditions for later plays.
Playbook Manager	The playbook manager plugin maintains the playbook and provides the services needed to manipulate and use the playbook.
Sensor	A sensor creates and publishes sensor conditions. The term sensor may be used elsewhere with a different meaning, but this is the meaning used in this section.

8.2 Principles of Operation

An adaptivity engine monitors condition and continually updates the operating modes of plugins and other components as they change. Adaptivity engines can be run at various levels. The two most important adaptivity engines are community adaptivity engines and agent adaptivity engines. Community adaptivity engines set the operating modes of agents and agent adaptivity engines control the operating modes of their plugins and other components.

The values for these operating modes are specified by plays in a playbook. Each play has a Boolean expression written in terms of the conditions available to the adaptivity engine. When the expression evaluates "true", the play is selected to set one or more operating modes. The syntax of the plays is described in the javadoc for `org.cougaar.core.adaptivity.Parser`

A play does not necessarily specify a single value for an operating mode. Instead, it may specify an ordered list of ranges of values that specify constraints on the value of the operating mode. This allows multiple plays to jointly control an operating mode. This becomes particularly important when the constraints imposed by policies are considered. The constraints specified are combined by intersection. The order of the elements is governed by the first selected play. After the constraints from all the plays are combined, an effective value is computed from the combined constraint. This value is simply the minimum of the first range in the list.

Policy managers may wish to constrain the operating modes. Managers that are cognizant of the available operating modes can directly constrain those operating modes by publishing `OperatingModePolicies` to the blackboard. The operating mode policy manager will then apply the constraints specified by such policies to the playbook. The constraints are applied as follows:

1. Identify the plays that affect the same operating mode and split them into two plays: one for when the policy does not apply and one for when it does.

2. The condition of the play that applies when the policy does not apply is obtained as follows: if the condition of the operating mode policy is `ompc` and the condition is `playc`, then the new condition for the play is `(playc & !ompc)`. The operating mode constraints are copied from the original play.

3. The condition of the play that applies when the policy *does* apply is `(playc & ompc)`. The operating mode constraints that the play and the policy have in common are modified by intersecting the constraints (just as is done when multiple plays affect the same operating mode). In this case, however, if the intersection is empty, the constraint from the play is ignored and the constraint from the policy is used. This is an error because it means that the playwriter did not anticipate that such a policy might be imposed so it is logged as such. Constraints on operating modes that are not affected by the policy are cimply copied from the original play.

4. Finally, a play composed directly from the policy is added to the playbook in case there are no plays left that set the operating modes affected by the policy.

Often, operating mode policies will not directly affect the operating modes of components. This is particularly true when the policies are established at the community level because the detailed makeup of an individual agent is not (indeed, should not) be known to the community level adaptivity engine. Instead, these operating mode policies constraint (set) the value of a synthetic operating mode. This synthetic operating mode can then be referenced as a condition by plays in the agent's playbook.

As an example, consider the following operating mode policy:

```
True: DefensePosture=3;
```

and these plays:

```
DefensePosture>=3:Enc.kl=128;

DefensePosture>=3 | Hardware.SignerPresent=true
:Persistence.sign=true;
```

The combined effect of the policy and plays in the playbook is that when the given operating mode policy arrives in the agent, the operating modes of two components are affected. The `Enc` component has its `kl` mode set to 128 and the `Persistence` component has its `sign` mode set to true. But, notice that the `Persistence.sign` mode is also set if the `Hardware` sensor has specified that the `SignerPresent` condition is true.

The sections below give advice on using the adaptivity engine. There are three aspects to this: creating the conditions that the plays depend on, creating the operating modes that the plays affect, and writing the plays.

8.3 Writing an Adaptivity Playbook

Every play in a playbook should contribute toward establishing operating modes for various components such that the ensemble works efficiently according to a defined figure of merit. Every play represents a compromise on resource consumption to achieve different ends. A play that squanders all the available resources toward maximizing one particular measure of performance at the expense of all the other such measures *may not* be the best possible play.

There are a number of ways for a playbook to fall short. For example, there may be factors that affect performance that are not available as conditions for the playwright to use. Certain combinations of conditions may have been overlooked or assumed to not be distinctively different from some other combination. Another class of shortcoming is to incorrectly assume that the condition/operating space can be factored in certain ways; that is, to assume that certain conditions and operating modes are only related to each other and are independent of all other conditions and operating modes. If this degree of independence is false, then the playbook will be sub-optimal.

8.4 Writing an Adaptive Plugin

The hard task is to determine what degrees of freedom an adaptive plugin should have. The easy part is creating and publishing operating modes to control those degrees of freedom. The OperatingMode interface defines the API of an operating mode. Most plugins can use the standard OperatingModeImpl implementation. The operating modes should be created and published in the plugin's `setupSubscriptions` method. The plugin should retain each mode in a variable for later use.

There are two primary styles of using operating modes that a plugin may adopt. Most commonly, the plugin will simply reference the current values of the operating modes as it decides how to carry out a particular activity. A variation of this style is to record the values of the operating modes at the beginning of the `execute()` method and use the recorded values as above. The second style is for plugins that need to revise their earlier work when an operating mode changes. Such plugins should create a subscription for the operating mode so they can be wakened when it changes.

8.5 Writing a Sensor

The concept of a sensor has several meanings. In this context, we are referring to a component that supplies conditions to the adaptivity engine. Such a sensor might make use of various services that supply information useful to the adaptivity engine or perform measurements on the blackboard, or anything else. Whatever the source of the information that is to be put into a condition, the procedure for making this information available to the adaptivity engine is simple: create and publish a `Condition`. Most sensors will use an inner class extension of the `SensorCondition` class. The SensorCondition class has a protected setValue method so that only the sensor will be able to change the value of the condition. Refer to the javadocs for more detail.

9 Cougaar Administration and Programming Techniques

Having described the primary components of Cougaar from a technical perspective, this section attempts to back up a level. This section provides a basic introduction to installing and running Cougaar, and then a more detailed look at advanced topics in running and debugging Cougaar societies.

9.1 Cougaar Quick Start Guide

9.1.1 Introduction

The Cognitive Agent Architecture (Cougaar) is a Java-based architecture for the construction of large-scale distributed agent-based applications. It is a product of two consecutive, multi-year DARPA research programs into large-scale agent systems spanning eight years of effort.

Cougaar provides developers with a framework to implement large-scale distributed agent applications with minimal consideration for the underlying architecture and infrastructure. The Cougaar architecture uses the latest in agent-oriented component-based design and has a long list of powerful features.

For more information on Cougaar, see our open source site http://www.cougaar.org, and in particular the FAQ: http://cougaar.org/docman/view.php/17/55/FAQ.html

9.1.2 Cougaar Distribution

The latest version of Cougaar can be obtained at http://cougaar.org/projects/cougaar/. Cougaar can be installed both manually and through the use of a Windows or LINUX based installer. Both installation methods are described below.

9.1.3 Requirements

- Java 1.4.2+ installed and on the system path (see http://java.sun.com)
- Optional (required for some applications): MySQL version 3.23+ (see http://dev.mysql.org)

9.1.4 Installing Cougaar

9.1.4.1 Installer-based Installation

Ensure all the requirements above are satisfied, run the installer. Once Cougaar has been installed, you must define an environment variable: COUGAAR_INSTALL_PATH to point to the install location and ensure that COUGAAR_INSTALL_PATH/bin is on your system path.

Note: On Windows, the above steps are handled by the installer.

9.1.4.2 Custom Installation

This document outlines a basic custom installation. For a more detailed custom installation, see the CSMART install guide (COUGAAR_INSTALL_PATH/csmart/doc/InstallAndTest.html):

1) Ensure that you meet the JDK requirements above.

2) If you plan on using some CSMART functionality of the "logistics" project, then you will need to also install MySQL, as specified above. You can use the Installer, or install MySQL as described in the CSMART install guide.

3) Create a Cougaar directory and set a COUGAAR_INSTALL_PATH environment variable to point to this directory.

4) Recommended: Put COUGAAR_INSTLL_PATH/bin on your PATH environment variable.

5) Unzip the desired zip files.
   (*The following two are required*.)

   a. ***Cougaar.zip***, this contains the core Cougaar infrastructure.

   b. ***Cougaar-support.zip***, this contains all required 3rd party support jars.

   (The other jars, such as –src and –api are optional.)

6) You should now be able to run the pizza application, the tutorials, or develop your own configurations.

9.1.5 Configuring Cougaar

Cougaar societies are made of Nodes (JVMs), which contain Agents, which in turn contain components. All of this configuration data is preferably specified in an XML file. (Note that some existing societies may still use the deprecated CSMART database or "INI" files.)

The current method for configuring a Cougaar society is through the use of XML configuration files. The configuration file performs the mapping of Components to Agents, Agents to Nodes and Nodes to Hosts.

As an example, the basic installation of Cougaar contains a simple XML society configuration; Pizza Party located in `pizza/configs/pizzaparty` that can be run without modification. See pizza/docs/Readme.html for details.

9.1.6 Running Cougaar

To run Cougaar, you simply use the "c**ougaar[.bat]**" script. It takes as an argument, the name of the XML file specifying your society, and the name of the Node to run. See the script's help for details. Note in particular that you may specify various VM parameters on the command line, or in your society.xml. Note in particular that `COUGAAR_DEV_PATH` should point to your custom developed classes that over-ride or replace Cougaar classes. ***Do not use the `CLASSPATH` environment variable.***

For example, to run the Pizza society on a Windows machine, do:

```
$ cd %COUGAAR_INSTALL_PATH%\pizza\configs\pizzaparty
$ %COUGAAR_INSTALL_PATH%\bin\cougaar.bat SDPizzaNode1.xml
COUGAAR 11.4 built on Tue Dec 07 07:04:16 GMT 2004
Repository: HEAD on Tue Dec 07 07:00:08 GMT 2004
VM: JDK 1.4.2_06-b03 (mixed mode)
OS: Windows 2000 (5.0)
16:52:11,680 SHOUT - XMLComponentInitializerServiceProvider -
Initializing node "SDPizzaNode1" from XML file "SDPizzaNode1.xml"
...
```

Then start the second node in another window:

```
$ cd %COUGAAR_INSTALL_PATH%\pizza\configs\pizzaparty
$ %COUGAAR_INSTALL_PATH%\bin\cougaar.bat SDPizzaNode2.xml
COUGAAR 11.4 built on Tue Dec 07 07:04:16 GMT 2004
Repository: HEAD on Tue Dec 07 07:00:08 GMT 2004
VM: JDK 1.4.2_06-b03 (mixed mode)
OS: Windows 2000 (5.0)
16:52:11,680 SHOUT - XMLComponentInitializerServiceProvider -
Initializing node "SDPizzaNode2" from XML file "SDPizzaNode2.xml"
...
```

9.1.7 Configuring a Cougaar Society

Cougaar societies are made of Nodes (JVMs), which contain Agents, which in turn contain components. All of this configuration data is preferably specified in an XML file. Deprecated configuration options include "INI" files, which are similar to XML files, and the CSMART database, which is documented in the CSMART Install Guide and User's Guide.

The society XML file starts with a basic template:

```
<?xml version="1.0" encoding="UTF-8"?>
<society
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  name="SOCIETY"
  xsi:noNamespaceSchemaLocation="http://www.cougaar.org/2003/society.xsd">
  <!— host(s) go here →
</society>
```

The "<society>" tag can contain one or more "<host>" tags, which contain "<node>" tags, which contain "<agent>" tags, which in turn contain "<component>" tags. This maps of Nodes to Hosts, Agents to Nodes, and Components to Agents. Nodes may also contain "<component>" tags to define node-agent components. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<society
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  name="my_society"
  xsi:noNamespaceSchemaLocation="http://www.cougaar.org/2003/society.xsd">
  <host name="localhost">
    <component class="MyNodePlugin"/>
    <node name="localnode">
      <agent name="Foo">
        <component class="TestPlugin">
          <argument>x=y</argument>
        </component>
      </agent>
      <agent name="Bar">
        <component class="Alpha"/>
        <component class="Beta">
          <argument>one</argument>
          <argument>two</argument>
        </component>
      </agent>
    </node>
  </host>
 </society>
```

Nodes can also specify Java command line parameters, which are extracted by the bin/Cougaar script. This duplicates the "-D" system properties in the bin/Cougaar scripts, but allows easier control over per-node "-D"s, and consolidates configuration management in the XML file. For example:

```
<?xml version="1.0" encoding="UTF-8"?>
<society ..>
  <host name="localhost">
    <node name="localnode">
      <vm_parameter>
        -Dorg.cougaar.society.file=THIS_XML_FILENAME
      </vm_parameter>
      <vm_parameter>
        -Dorg.cougaar.node.name=localnode
      <vm_parameter>
      <vm_parameter>
        -Dorg.cougaar.install.path=$COUGAAR_INSTALL_PATH
      </vm_parameter>
      <vm_parameter>
        -Dorg.cougaar.core.node.InitializationComponent=XML
      </vm_parameter>
      <vm_parameter>-Xms100m</vm_parameter>
      <vm_parameter>-Xmx300m</vm_parameter>
      <!— the following is standard; to add jars, place them in
          $COUGAAR_INSTALL_PATH/lib →
      <vm_parameter>
        -Xbootclasspath/p:$COUGAAR_INSTALL_PATH/lib/javaiopatch.jar
      </vm_parameter>
      <vm_parameter>
        -Djava.class.path=$COUGAAR_INSTALL_PATH/lib/bootstrap.jar
      </vm_parameter>
      <class>org.cougaar.bootstrap.Bootstrapper</class>
      <prog_paramter>org.cougaar.core.node.Node</prog_parameter>

     <!— optional: node-agent component(s) →
     <!— optional: agent(s) →
    </node>
    <!— optional: more nodes →
  </host>
  <!— optional: more hosts →
 </society>
```

Within an "<agent>" or "<node>" tag, "<component>" tags are used to specify components. The format is:

```
<component
  class=CLASS
  [ insertionpoint=INSERTION_POINT ]
  [ name=NAME ]
  [ priority=PRIORITY ]>
  ( <argument>ARGUMENT</argument> )*
</component>
```

The default INSERTION_POINT is the standard plugin insertion point, "Node.AgentManager.Agent.PluginManager.Plugin".

The default NAME is the CLASS followed by parenthesis around a comma-separated list of ARGUMENTS, for example "AnyClass(A, B, C)". Component names are only used to distinguish two components with identical classnames and argument lists.

A component can specify zero or more "<argument>" tags, which are passed to the component at runtime through the "setParameter(Object o)" method as a List of Strings. If zero arguments are specified then the "setParameter(Object o)" method is not called.

The default PRIORITY is “COMPONENT”, which is used to order components relative to other priorities defined in the XSL template. Component priorities are essentially a static tool to order components into a flat list of components, which is done through the XSL template, are are ignored by the running Node. The standard priorities defined by the default agent templates are:

```
  HIGH > INTERNAL > BINDER > COMPONENT > LOW
```

These priorities are specified and sorted within the XSL template.

A couple example component tags are:

```
<society ..>
  <host ..>
    <node ..>
      <agent ..>
        <!-- typical plugin →
        <component class=”MyPlugin”/>

        <!—an agent-level component with parameters →
        <component
           class=”AnotherExample”
           insertionpoint=”Node.AgentManager.Agent.Component”
           priority=”HIGH”>
          <argument>a=b</argument>
          <argument>foo</argument>
        </component>
      </agent>
    </node>
  </host>
</society>
```

A “<node>” and “<agent>” tag can specify an alternate XSL template file, to control the default components loaded into the agent. For example, this can be used to create a minimal agent without a blackboard, or specify an alternate message transport service. Details can be found in section 9.1.17. An example is:

```
<society ..>
  <host ..>
    <node name=”Foo” template=”MyCustomNode.xsl”>
      <agent template=”BlahTempate.xsl”>
      </agent>
    </node>
  </host>
</society>
```

9.1.8 XSL Agent Templates

In Cougaar 11.2, the Cougaar core was refactored to move the hard-coded agent components, such as Blackboards and the MTS, out of the agent base classes (SimpleAgent and NodeAgent) and into easily modifiable XSL configuration files. While most Cougaar developers will continue to use the standard agent definitions, this core refactor greatly increased the configurability and flexability of the underlying architecture.

By default, an XML file that specifies an agent using the typical format:

```
    <agent name="ANY_NAME">..<agent>
```

will default to the "SimpleAgent.xsl" template, as if the XML line was:

```
    <agent name="ANY_NAME" template="SimpleAgent.xsl">..</agent>
```

Nodes have a similar "NodeAgent.xsl" template default:

```
  <node name="ANY_NAME" template="NodeAgent.xsl">..</node>
```

These template files are found by the Cougaar ConfigFinder, and are typically kept in $COUGAAR_INSTALL_PATH/configs/common. If an empty template is specified, then no template is used; this is useful for agents that already specify all necessary components. For future enhancements, if an agent or node specifies a "type" attribute, then the templates are also not applied.

The job of the XSL template is to take a developer's XML configuration and expand the node and agent sections to include the template's components. For example, the standard template expands this minimal XML configuration:

```
<?xml version="1.0" encoding="UTF-8"?>
<society
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  name="my_society"
  xsi:noNamespaceSchemaLocation="http://www.cougaar.org/2003/society.xsd">
  <host name="localhost">
    <node name="Foo" >
      <agent name="Bar">
        <component class="MyPlugin"/>
      </agent>
    </node>
  </host>
</society>
```

to include default components such as the node MTS, thread service, WP, blackboard:

```
<?xml version="1.0" encoding="UTF-8"?>
<society
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  name="my_society"
  xsi:noNamespaceSchemaLocation="http://www.cougaar.org/2003/society.xsd">
  <host name="localhost">
   <node name="Foo" template="">
    <component
     insertionpoint="Node.AgentManager.Agent.Component"
     priority="HIGH"
     class="org.cougaar.core.agent.service.event.EventServiceComponent"
     name="org.cougaar.core.agent.service.event.EventServiceComponent()"/>
   <component
     insertionpoint="Node.AgentManager.Agent.Component"
     priority="HIGH"
     class="org.cougaar.core.agent.service.uid.UIDServiceComponent"
     name="org.cougaar.core.agent.service.uid.UIDServiceComponent()"/>
   ..
   <agent name="Bar" template="">
     <component
       insertionpoint="Node.AgentManager.Agent.Component"
       priority="HIGH"
       class="org.cougaar.core.agent.service.event.EventServiceComponent"
       name="org.cougaar.core.agent.service.event.EventServiceComponent()"/>
     ..
     <component class="MyPlugin"/>
     ..
    </agent>
  </node>
 </host>
</society>
```

The full transform adds a total of 65 components. This can be run on the command line using:

```
cd $COUGAAR_INSTALL_PATH/configs/common
java org.apache.xalan.xslt.Process -in INPUT_XML -xsl society.xsl > OUTPUT_XML
```

If custom "template" attributes are specified, a two-step process used by the runtime Cougaar must be used:

```
cd $COUGAAR_INSTALL_PATH/configs/common
java org.apache.xalan.xslt.Process -in INPUT_XML \
  -xsl make_society.xsl > MY_SOCIETY_XSL
java org.apache.xalan.xslt.Process -in INPUT_XML \
  -xsl MY_SOCEITY_XSL > OUTPUT_XML
```

Modifying the XSL template will modify the transformed XML, resulting in custom node or agent types. For example, to disable Servlet support, one can modify the SimpleAgent.xsl and NodeAgent.xsl files (or make copies and specify "template" attributes), commenting out the components that load the Servlet server. For the servlet server, there are two components: an NodeAgent.xsl component:

```
  <component
    name="org.cougaar.lib.web.service.RootServletServiceComponent()"
    class="org.cougaar.lib.web.service.RootServletServiceComponent"
    priority="HIGH"
    insertionpoint="Node.AgentManager.Agent.Component"/>
```

and a SimpleAgent.xsl component :

```
  <component
    name="org.cougaar.lib.web.service.LeafServletServiceComponent()"
    class="org.cougaar.lib.web.service.LeafServletServiceComponent"
    priority="BINDER"
    insertionpoint="Node.AgentManager.Agent.Component"/>
```

In this case, there's already a system property that can be used to disable these two components:

```
    -Dorg.cougaar.society.xsl.param.servlets=false
```

This system property is only supported by the runtime Cougaar XSL parser, and not the command-line "org.apache.xalan.xslt.Process" parser.

As seen in the above example, system properties can be passed from Cougaar to the XSL parser. The properties must have the prefix "-Dorg.cougaar.society.xsl.param".

Alternate minimal implementations have been developed for some core components, such as a local-only MTS, minimal thread service, and trivial metrics service. These implementations are ideal for trimmed Cougaar configurations and performance testing. For details, see the comments in NodeAgent.xsl.

Many of the default SimpleAgent.xsl and NodeAgent.xsl components are loaded in a precise order to obtain services at just the right time, or take action at the right time in their "load()" or "start()" relative to the other components. Several new core services exist just to "glue" the standard components together, coordinating their actions. Many components require services to exist that they might do without, such as blackboard's requirement for persistence. In future releases of Cougaar, as the architecture is customized for new applications, these service dependencies will likely be minimized.

A small number of components are still hard-coded to load, since they are required to bootstrap the agent to the point where it can read its XSL and XML files. An agent starts with a Bootstrap component, which adds 7 more components, such as the persistence service and XML configuration reader. Additional details can be found in Cougaar enhancement request bugs 3653 and 3654.

Included in the core refactor was the cleanup of component priorities. Prior to Cougaar 11.2, component priorities were fixed string constants that specified points in between hard-coded components in SimpleAgent and NodeAgent. In the new XSL design, component priority strings are interpreted by the XSL file, allowing developers to modify the meaning of these priorities by modifying the XSL file. For example, a developer can add a new "MEDIUM" priority and modify their XSL to sort these components in the correct order relative to the standard HIGH/INTERNAL/BINDER/COMPONENT/LOW components. The developer can then specify this "MEDIUM" priority in their XML file, e.g.:

```
<component class="MyComponent" priority="MEDIUM"/>
```

The default XSL transform is to process "make_society.xsl" and then use the output XSL to process the XML file once again. The outer "make_society.xsl" transform can be bypassed by specifying a stylesheet in your XML file, e.g.:

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xml" href="society.xsl"?>
<society ..>
  ..
</society>
```

For details, see the javadocs for "org.cougaar.core.node.XMLComponentInitializerServiceProvider" and "org.cougaar.core.node.XMLConfigParser". Also note that the "make_society.xsl" supports embedded XSL templates, e.g.:

```
<?xml version="1.0" encoding="UTF-8"?>
<society ..>
  <xsl:stylesheet>
    <!—XSL templates here! →
  </xsl:stylesheet>
  ..
</society>
```

Although few developers will modify the default XSL templates to this extreme, these XSL options allow great flexability over the processing of the XML file, including arbitrary content transforms and dynamic node/agent generation.

9.1.9 Monitoring a Cougaar Society

Once a Cougaar society is created and run, there are various methods that can be used to monitor progress of the society.

When a node is started, various status messages are displayed in the console window indicating the current state of the society. The following is a summary of each of these message tokens:

'+'	A message has been received from another Agent
'-'	A message has been sent to another Agent
'.'	Node activity has "quiesced," meaning that no messages have been sent or received and no Plugins have been invoked in the last 5 seconds.
'P'	Persistence information has been saved
'R'	Rescind messages have been received from another Agent
'!'	This indicates that no name server has been contacted. If these messages continue, check that

	your alpreg.ini files are identical and point to a machine with a running Node.

Cougaar also contains an embedded servlet engine and some useful society monitoring servlets. These can be viewed, while the society is running, at http://localhost:8800. Two useful monitoring servlets are: `/agents` (to list the running agents on a Node) and `/tasks` (to display the objects on the Blackboard of an Agent).

Another useful tool that provides the ability to graph the society and task flows is the CSMART Society Monitor. For more information on using this tool, see the CSMART Install document.

9.1.10 Next Steps

Now that you have installed Cougaar, and run your first Cougaar society, you are ready for more complex societies.

For example, you could split up the Pizza configuration across more Nodes / hosts. Or change the set of components loaded in the agent.

9.1.11 Learning More and Getting Assistance

To learn more about Cougaar agents and societies, and examples on creating your own agents, see: http://cougaar.org/projects/tutorials/.

For more advanced discussions of developing Cougaar components, see the Cougaar Developers Guide, available on the Cougaar web site: http://cougaar.org/docman/?group_id=17.

There is a significant community of Cougaar developers, so there are a number of resources available to help you solve your problems.

- Bugs and Requests For Enhancements can be reported on the Cougaar Bugzilla web site at http://bugs.cougaar.org/
- Community support mailing lists. See http://www.cougaar.org/mail/ for information on subscribing and archives.
  - Cougaar-developers@cougaar.org is a community support mailing list. Many developers of Cougaar applications and infrastructure read and post here.
  - Cougaar-information@cougaar.org is a forum for announcing software updates, workshops, training sessions, and conferences.
  - Cougaar-help@cougaar.org is for licensing issues, requests for high-level information and web-site comments.
- Frequently Asked Questions (FAQ). See http://cougaar.org/docman/view.php/17/55/FAQ.html.
- Conferences and Workshops. Post a request to cougaar-help@cougaar.org if you are interested in attending a Cougaar conference, workshop or training. Watch for course or workshop announcements on the Cougaar web site, at http://cougaar.org.

9.2 Cougaar Advanced Configuration Guide

Cougaar Infrastructure Runtime Switches

There is a section in the "setarguments" or "boost" file that allows tailoring the properties of a Cougaar JVM by setting system properties. System properties are set to a particular value by setting "-Dproperty=value" in the setarguments file. There are many such properties; the following table specifies some of the most important parameters for tailoring the desired behavior of the Cougaar JVM. A complete list is included in the Javadoc file doc/api/Parameters.html.

Parameter Name	Semantics	Example Value
org.cougaar.install.path	System variable to fix the root of the Cougaar installation.	d:\opt\cougaar
user.timezone	Sets the timezone of time operations within the JVM	GMT
org.cougaar.core.persistence.path	Directory to which to write persistence data	persistDir
org.cougaar.core.persistence.enable	Read persistence data if available on startup and write persistence deltas and checkpoints.	true
org.cougaar.config.path	Series of URL search paths over which to search for config files. Note these could be URLs to remote or local file service.	http://alp-3/alp/configs/flag-config;http://alp-3/alp/conf
org.cougaar.config	Specifies the config directory from which files are served by default.	flag-config
org.cougaar.workspace	Specifies a directory where files may be written to.	d:\opt\cougaar\workspace
org.cougaar.core.logging.config.filename	Specifies the logging properties file. (See below)	logging.props

Cougaar Society Configuration Details

For an XML specified society, a separate XML file must be created for each Node that you wish to create and run (or you may include all Nodes within a single file and use it in multiple places). The XML file shows the Host, Node, and Agents to be included on that Node for that Host. The Components (Plugins and their arguments) are also included. Facets are an optional element which describe attributes of the particular agent. Shown below is an excerpt of the XML specified MiniTestConfig society.

```
<?xml version='1.0'?>
<society name='MINITEST-SOCIETY'
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'

xsi:noNamespaceSchemaLocation='http://www.cougaar.org/2003/society.xsd'>
  <host name='localhost'>
    <node name='localnode'>
      <class>
        org.cougaar.bootstrap.Bootstrapper
      </class>
      <vm_parameter>
        -Dorg.cougaar.node.name=localnode
      </vm_parameter>
      <agent name='1BDE' class='org.cougaar.core.agent.SimpleAgent'>
        <facet org_id='1BDE'  />
        <facet orig_org_id='1BDE'  />
        <facet superior_org_id='3ID'  />
        <facet subordinate_org_id='3-69-ARBN'  />
        <facet home_location='BBNV'  />
        <facet uic='WAQNA1'  />
        <facet combat_support='CMD'  />
        <facet echelon='BRIGADE'  />
        <facet echelon_group='DIVISION'  />
        <facet echelon_of_support='BRIGADE' role='Headquarters'  />
        <component

name='org.cougaar.core.servlet.SimpleServletComponent(org.cougaar.planni
ng.servlet.PlanViewServlet,/tasks)'
          class='org.cougaar.core.servlet.SimpleServletComponent'
          priority='COMPONENT'
          insertionpoint='Node.AgentManager.Agent.PluginManager.Plugin'>
          <argument>
            org.cougaar.planning.servlet.PlanViewServlet
          </argument>
          <argument>
            /tasks
          </argument>
        </component>
...etc.
```

Note that in order to run many of the supplied example military logistics societies, you must also configure a relational database for use by the included planning plugins. See the installation documentation elsewhere for details. In particular, be sure to correctly configure the required cougaar.rc file, which is used by running Agents to locate the database. (See doc/OnlineManual/DataAccess.html)

The pizza and MiniTestConfig examples do not rely on any data contained in the database nor require the existence of a valid database.

9.3 Embedded Cougaar Configuration

This section describes how to configure small footprint Cougaar societies that use minimal resources (cpu, memory, network) and have minimal survivability requirements. These configuration options can be used to deploy a Cougaar society in an embedded processing environment, for example a hand-held computer running Linux. Embedded Cougaar is a pruned version of the standard Cougaar installation and uses the standard Java VM (1.4.2). For a Cougaar version which uses a restricted Java VM, see Cougaar ME.

9.3.1 Background

The Cougaar infrastructure is highly configurable. Cougaar societies can be configured to operate in a wide range of operating environments, from a single stand-alone host to a survivable society with hundreds of hosts distributed over a potentially malicious wide-area network. Cougaar's default configuration is designed to deploy Cougaar societies in a relatively benign local-area network, such as an enterprise network behind a firewall. For deployments with higher survivability requirements, additional components and agents can be added, including the security and robustness defenses developed under the Ultralog program. As a result, the most common emphasis when configuring a Cougaar society is on adding components in support of the application's functionality and survivability requirements. But embedded deployments need to remove all but the most essential components.

Cougaar can be stripped down to deploy a society efficiently on an embedded computer. Cougaar infrastructure can even be configured without Agent support, that is, using only Cougaar's component model to make a Service Oriented Architecture (SOA) for deploying stand-alone Java applications that require rich plugin support. These embedded applications do not need the survivability support required by distributed societies.

When a society is deployed on a single node, performance can be increased by a factor of three by removing the unnecessary Node-level components that are designed to support survivability. Since this single-node configuration runs on a single host, an additional 7x performance increase can be realized because inter-agent communication stays within the Node, i.e. messages are not serialized or sent over sockets. The total performance increase of single-node configuration can therefore be more than a factor of 20, which is very useful for debugging a society's functionality before deployment or for deploying societies on embedded processors. The society's Agent plugins do not have to be modified for deployment in distributed environments; adding Node-level components to increase survivability is sufficient.

9.3.2 Procedure

Embedded Cougaar involves determining the minimum set of components needed to meet the society's survivability requirements. There are three main steps:

1. The minimal Cougaar installation must be identified and installed on the embedded device. Here the goal is to reduce the disk space to store the Cougaar society. For example, this step will prune the jars, configuration data, and shell scripts out of the standard Cougaar release.

2. The society XML configuration must be modified to remove unnecessary components. Here the goal is to remove components from the standard configuration, which are not required in the embedded deployment. For example, this step may involve adding new rules for generating the society or modifying the XSL Node definition template.

3. The society execution (run scripts) must use additional switches to execute the reduced society XML configuration. Here the goal is to get the Java VM configured to restrict the use of runtime resources, such as stack and heap size. As of Cougaar B11_2, these VM switches can be specified in the society XML. A modified run script is not needed.

The design of embedded Cougaar either removes or replaces (with a trivial implementation) the standard Node-level services. The choice of which services can be mixed and matched and depend on the survivability requirements of the society.

More details and new examples can be found under the online documentation as part of the Cougaar software release.

9.4 Cougaar Society Management

The management of a Cougaar society involves several pieces: starting and stopping of nodes, monitoring the state of nodes and Agents, and managing persistence and rehydration of Agents. The current state of Cougaar society management is largely manual interaction and is described in the following section. The Cougaar team anticipates tools that enable a much more automated, centralized approach to Cougaar society management. For a description of one approach to this problem, see the CSMART User's Guide. A script or command-line based approach is in the works.

9.4.1 Current Approach

The current approach to Cougaar society management is largely manual. Typically, you start a society by opening a separate command console window on each machine where a Node will be running. For XML societies, each node is started by executing the command "Cougaar.bat <Society Name> <Node Name>". Generally this command is issued from the directory where the XML society configuration files are located.

The output of the node back to the console window consists of developer status messages, error messages (which should be enabled and disabled by setting appropriate debug level flags), and stack-traces indicating errors or anomalous conditions. More significantly, there is the stream of status tokens indicating the flow of messages and other system activity (as described in the Quick Start Guide). By monitoring this activity, one can see if information is flowing properly, if persistence is working, or if the node activity has "quiesced" (no Plugin or Message activity in a period of time). Also note that many if not most of the status and error messages are output using the Cougaar Logging Service, which is described in the CDG. Note that the verbosity of these logs is highly configurable.

The monitoring of system performance is also currently manual: one uses the system Monitor tools on NT or the expert (or comparable) tools on LINUX/UNIX to monitor CPU and Memory consumption on a given host. If the memory level approaches or exceeds the limits of physical memory, system performance will degrade dramatically, and reliability of the JVM may degrade. Typically, running out of memory is an indication that Agents need to be reconfigured to other nodes, or that other hardware platforms need to be enlisted. CPU saturation should be monitored closely: the CPU may be entirely consumed from time to time. However, if one Agent's activity is precluding another's activity, it is likely a sign that the Agents need to be separated into different nodes. Additionally, developers may use various application-specific servlets to measure progress towards solving the application problem. These CompletionServlets are provided for the planning domain.

When Cougaar is running with persistence enabled, the persistence "delta" or snapshot is saved to a directory per Agent. By default, the persistence directory is $COUGAAR_INSTALL_PATH/workspace/P, so an Agent named Foo will have its persistence information stored to $COUGAAR_INSTALL_PATH/workspace/P/Foo. Every tenth snapshot is a consolidated record of the entire state (rather than a delta from the prior snapshot). The deltas embodied by the consolidated snapshot are discarded, but all consolidated snapshots are retained by default.

An Agent can be configured to save persistence information using various media plugins. Plugins to save to files and to databases are available. The media plugins can be configured to store persistence information in locations accessible to multiple hosts (e.g. shared filesystems) so that an agent can be reincarnated on a different host from that on which it was originally running.

Stopping a node is simply a matter of killing the process by killing the window containing the JVM process, or "control-C-ing" the process in the window. The node can then be restarted manually and will rejoin the society in progress. Alternatively, the entire society can be brought down and restarted, though there is no requirement to do so since dynamic Agent and Plugin operations are supported.

9.5 Servlet Server Install Guide

Servlet development is covered in section 4. This section addresses configuration options, such as control over the HTTP port, HTTPS, and client-authentication.

9.5.1 Standard Options

By default, the servlet server is enabled to run HTTP on port 8800. New users are encouraged to first try a standard Cougaar HTTP configuration before trying to modify the configuration.

To disable the server, add this Java property:

```
-Dorg.cougaar.core.servlet.enable=false
```

Also see section 9.1.7 to disable servlets by modifying the XSL templates.

System properties that can be used to configure the server include:

```
-Dorg.cougaar.lib.web.scanRange=INT(defaults to 100)
-Dorg.cougaar.lib.web.http.port=INT(defaults to 8800, -1 disables HTTP)
-Dorg.cougaar.lib.web.https.port=INT  (defaults to -1, -1 disables HTTPS, the
typical value is 8400)
-Dorg.cougaar.lib.web.https.clientAuth=BOOL  (defaults to false)
```

The “scanRange” sets a limit for the server's search for an open port. First the given HTTP/HTTPS ports are tried, then they are both incremented by one, tried again, etc. If more than “scanRange” ports are already in use then the server throws a BindException and quits. Setting the “scanRange” to 1 will force exact port addresses.

To optionally disable the server printing of stack-traces to clients, such as when a servlet throws an exception, modify:

```
webtomcat/data/conf/server.xml
```

to specify:

```
<ContextManager .. showDebugInfo="false" >
```

Note that this will make servlet-debugging more difficult, since users can no longer cut-&-paste exceptions and report them back to the developers. However, it can be seen as a minor security enhancement, since it hides the code details.

9.5.2 HTTPS

To enable HTTPS, set the Java property to specify the HTTPS port:

```
-Dorg.cougaar.web.https.port=8400
```

Port “8400” is the typical Cougaar HTTPS port.

Create a keystore to hold the server's private RSA certificate. There are many 3rd-party utilities that can be used to create a server certificate, but here we'll use the Sun “keytool” that is included with the JDK: http://java.sun.com/products/jdk/1.2/docs/tooldocs/solaris/keytool.html

The keystore file must be placed in the $COUGAAR_INSTALL_PATH or a subpath -- here we'll name the file “$COUGAAR_INSTALL_PATH/bin/certs”. Note that the servlet server does not assume the typical JSSE/JDK's “${user.home}/.keystore” and “${java.home}/jre/lib/security/.keystore”.

The certificate alias must be “tomcat” to work with Cougaar's Tomcat 4.0.3 servlet server implementation, and must be an RSA certificate. Additionally the certificate's “name” must be the host's name, such as “foo.com” . Here is the full keytool command:

```
cd $COUGAAR_INSTALL_PATH/bin
keytool -genkey -keyalg rsa -alias tomcat \
  -keystore certs -dname “o=cougaar, cn=HOSTNAME ” \
  -keypass PASSWORD -storepass PASSWORD
```

where, for example, HOSTNAME is “foo.com” and the password is “changeit”.

The above keystore will allow server-only SSL authentication, which will provide an encrypted channel for client-server communications. See below for two-way client-server authentication.

Ideally all machines in a multi-host society would have different keystores and certificates. For testing one can create a single keystore and copy it amongst the machines, then ignore the “hostname doesn't match certificate's hostname” warnings when using HTTPS.

The keystore filename and password are kept within Cougaar's “org.cougaar.util.Parameters” table, as “org.cougaar.web.keystore” for the keystore file name, and “org.cougaar.web.keypass” for the password to the keystore

In this example we want the values to be:

```
org.cougaar.web.keystore=bin/certs
org.cougaar.web.keypass=changeit
```

Edit the “cougaar.rc” file to add the above two lines. The “cougaar.rc” is typically kept as:

```
$COUGAAR_INSTALL_PATH/configs/common/cougaar.rc
```

See the javadocs for “Parameters” for additional options, such as passing these values as “-D” system properties.

To test your HTTPS configuration, use a browser to access a Cougaar node, e.g. https://foo.com.8400. Your browser will be prompted to accept the server's certificate. If you specify localhost instead of the full host name, your browser may generate a warning.

9.5.3 HTTPS Client Authentication

To additionally enable (optional) HTTPS client authentication:

Enable the “clientAuth” system property.

```
-Dorg.cougaar.lib.web.https.clientAuth=true
```

The client's public signer's certificate (“certificate-authority cert”) must be imported into the server's “bin/certs” keystore file. This is a Tomcat 4.0.3 requirement.

Unfortunately Keytool can not be used to create a client certificate. There are other 3rd-party tools and companies that will generate a client certificate and manage the certificate-authority work, such as OpenSSL (http://www.openssl.org/).

With client authentication enabled the server will only accept HTTPS requests from trusted clients. A trusted client is a client that provides a client-certificate (via SSL) that has been signed by a public certificate-authorities certificate that's already in the server's keystore. For example, if the server's keystore includes the public Netscape certificate, then all clients that present certificates that are signed by Netscape will be accepted. The client (browser, application, whatever) must be separately configured to pass the client certificate as part of the HTTPS request.

A special-case for client authentication is to let the client use the server's certificate, in which case the server's keystore only needs to contain the server's certificate.

A user-developed servlet can check for "HttpServletRequest.isSecure()" to see if a particular request used HTTPS security. For details, see:

http://java.sun.com/products/servlet/2.2/javadoc/javax/servlet/ServletRequest.html#isSecure()

9.5.4 Username and Password Authentication

As an example of username / password authentication, the Tomcat 4.0.3 server configuration included with Cougaar is configured to prompt for a login for all requests with paths that end in ".secure". This is specified in:

```
$COUGAAR_INSTALL_PATH/webtomcat/data/webapps/ROOT/WEB-INF/web.xml
```

The pattern can be modified to another path or multiple "<url-pattern>" entries for testing.

The usernames and passwords are stored in:

```
webtomcat/data/conf/tomcat-users.xml
```

By default (for testing only) there is one user:

```
username: test
password: test
```

To test the login support you should either modify the "web.xml" and/or define a servlet with a "*.secure" path within your XML files (e.g. "/tasks.secure").

9.5.5 Additional Configuration Options

Cougaar uses Tomcat 4.0.3, so many of the security options available within Tomcat are also available within Cougaar. For example, Tomcat Realms and form-based authentication can be enabled by altering the Tomcat XML configuration files in the Cougaar "webtomcat/data" directory.

For additional details, see:

- Tomcat 4.0 documentation page: http://jakarta.apache.org/tomcat/tomcat-4.0-doc/
- How-to doc on Tomcat Realms: http://www.onjava.com/pub/a/onjava/2001/07/24/tomcat.html
- Tomcat security overview: http://www.cafesoft.com/products/cams/tomcat-security.html

9.6 Cougaar Security Management

Cougaar provides interfaces for developers to integrate standard practice security measures at a number of points in the system operation. These include verifying jar files, encrypting and signing messages between agents, and using cryptographic user or agent identification for access control.

Current Cougaar development and research is extending the range of security possible in Cougaar. Research efforts include using hardware tokens for cryptographic identity, complex authorization control and management engines, complete operations auditing, blackboard access control, and many other areas. Here we describe those capabilities currently integrated into core Cougaar in some form.

The Cougaar build process takes all "jar" files delivered with the installation and signs them with the private key of the validated Cougaar configuration authority. If the contents of a jar file are modified, the class loader will refuse to load from the modified jar file.

Some deployments of Cougaar enhance its security by installing and running the Agents on a VPN or other IPSec tunneling configuration. In this way, messages within enclaves may be "in the clear," but between enclaves through the open internet are encrypted and impervious to interception and falsification. The management of a VPN or IPSec tunneling configuration is beyond the scope of this document.

- Cougaar uses the Tomcat servlet engine, which supports SSL security and additional security options. See section 9.4 for details.

9.6.1 Message Transport Security

All inter-Agent messages may be signed by the private key of the sending Agent. When the message is received on the remote side, the Agent validates the message against the public key of the sending Agent. If the message is unsigned, or signed by a different Agent than the one indicated, or if the message is different than the information indicated in the certificate, Cougaar will indicate the insecure condition and will refuse to receive and process the message.

The Cougaar Message Transport Service (MTS) provides a hook for encrypting and signing messages on a per-message basis. The approach is to add a `FilterOutputStream` on the message serialization on the sending side, and a `FilterInputStream` on the message de-serialization on the receiving side. The relevant classes are found in the core module, the package `org.cougaar.core.mts`:

- `MessageProtectionAspect`: When loaded, this aspect will call the `MessageProtectionService` at the right times to encrypt the message. If the `MessageProtectionService` is not available, a default (no nothing) service is used.

- `MessageProtectionService` defines the interface for protecting messages. The service can specify the type of encryption used, based on the source and destination of the message, and the type of `LinkProtocol` (transmission mechanism, ie RMI) being used.

The MTS uses various "LinkProtocols" to communicate between Nodes (SSL-RMI, CORBA, Email, NNTP, and UDP). Of these, only SSL-RMI offers session-based encryption using Node-to-Node keys.

Note: an implementation of the MessageProtectionService is not included in the Cougaar distribution.

The SSL session keys are supplied via the standard Java Security mechanisms, i.e. keys are loaded into a keystore in the users home directory. For details on setting up this keystore, see the Javadoc for `org.cougaar.core.mts.SocketFactory.`

To enable SSL-RMI, replace the `RMILinkProtocol` with the component `SSLRMILinkProtocol` in the XML configuration file. `SSLRMILinkProtocol` is part of the core module and is in the package `org.cougaar.core.mts`. Note, when changing the LinkProtocols, all default LinkProtocols are removed and must be explicitly specified. So you should at least specify both the `SSLRMILinkProtocol` and the `LoopbackLinkProtocol`.

9.6.2 DataProtectionService

The `DataProtectionService` signs and encrypts data streams as specified by security policies. The primary use of the service is to protect persistence snapshots, but other uses are possible. The data protection service comprises four interfaces described below:

Note: an implementation of the DataProtectionService is not included in the Cougaar distribution.

`DataProtectionService` is the primary interface through which data protection is obtained. The interface has two methods for wrapping input and output streams respectively. Both these methods have a `DataProtectionKeyEnvelope` argument allowing the encryption key and ancillary information needed to recover the streams to be stored and managed by the client.

`DataProtectionServiceClient` is an interface that must be implemented by the client. This interface serves two purposes. It allows the service to identify the client so that the appropriate protection policy can be used and it allows the service to spontaneously access the saved encryption keys. The latter is necessary when an agent's keys are being changed. The keys must be re-encrypted with the new key.

`DataProtectionKey` is a marker interface used to identify legitimate key material and to insure that it is `Serializable` so it can be stored and managed by the client.

`DataProtectionKeyEnvelope` is a container to allow access to the `DataProtectionKey` by the service. When the service is used for output, the service stores the key in the envelope and when the service is used for input, the service retrieves the key from envelope. It is the responsibility of the client save the key with the data in such a way that the two are intimately associated and can be retrieved together.

There are no particular restrictions on the timing of the use of the envelope. The client should be prepared for the service to use an envelope at any time. In practice, such accesses will occur before or during the first read from an input stream, before or during the closing of an output stream and soon after the `DataProtectionServiceClient` provides an envelope through its iterator.

These interfaces are used as follows: The process starts with the client requesting the service from its `ServiceBroker`. Then, to protect an output stream, an output stream wrapper is requested from the service. The service returns a wrapper that the client can use as an ordinary `OutputStream`. The service also places the key being used to protect the stream in the envelope provided by the client. The client stores the key along with the data. For example, the data might be written to "delta_00001" and the key might be written to "key_00001".

On the input side, the client requests an input stream wrapper providing the key previously used to protect the stream being wrapped. The service takes the key from the envelope and returns an input stream wrapper for the client to use. The client should be aware that the service is not required to validate the signature of the stream until all the data has been delivered to the client. This restriction is made so the protection service need not buffer the entire stream. Since the data has not been validated, the client must take care to not perform any risky actions with the data. If doubt exists, the client should buffer the entire byte stream before processing it. If the signature is not verified, an exception will be thrown when the last byte is read.

The third use is for the service to get an iterator for the envelopes the client is managing. As each envelope is returned, the service will extract the key, decrypt it with the old agent key, re-encrypt it with the new agent key, and stuff it back in the envelope.

9.7 Debugging Methodologies

This section describes some of the unique problems and solutions associated with debugging and troubleshooting a Cougaar application. It is not intended as a standalone document: it should be read with the Cougaar Architecture Document (CAD) and the rest of the Cougaar Developers Guide (CDG) close at hand. Often, the sources for the Cougaar infrastructure will be helpful, both from the perspective of gaining additional insight into how the infrastructure works and as significant body of working sample code.

The intended audience is both Plugin (and, more generically, Component) developers, and Society integrators/testers. Plugin debugging/testing is often in the context of a running society, so the larger view is important even to those testing small components. Since there is so much overlap between the various viewpoints, we will not even attempt to categorize the techniques.

9.7.1 Concepts

The Cougaar infrastructure represents a radical departure from traditional planning systems. While none of the following features are unique to the Cougaar design, the combination is sufficiently unusual to state up front. Also note that these are infrastructure design goals – few Cougaar applications will make full use of these features, even when the infrastructure is able to fully realize them.

- Continuous operation. Cougaar is designed to operate without system-wide resets. In a system that may have thousands or even millions of peer agents, it is extremely undesirable to design it in such a way as to require a full restart on every software upgrade (for instance). Different agents may move, may be intermittently connected, and may run on undependable hardware, itself subject to both planned and unplanned downtime. In practice, few Cougaar applications run for days – only a couple run for weeks and actually support partial shutdown and recovery.

- Distributed system. Cougaar societies are often distributed across multiple CPUs, over both LANs and wide-area networks. Getting the "big picture" of a distributed application can be extremely difficult, especially when the observation process may compete with the application for resources (e.g. network bandwidth).

- Parallel operation. Cougaar entities are designed to run in parallel at all levels – Nodes, Agents, Plugins, etc. There is no option for forcing serial operation. Infrastructure parallelism notwithstanding, planning problems are rarely, if ever, fully parallelizable – while the observed behavior of a Cougaar application may seem ordered and serial most of the time, it is important to realize that if the problem allows parallelism, the infrastructure will do its best to take advantage of it.

- Fully asynchronous operation. Cougaar was designed to avoid any critical point-of-failure: that is, the architecture specifies no centralized facilities at the agent (or above) level. The implication is that the agents are naturally independent of each other and will run not just in parallel, but without any synchronization other than that specified by the application. For example, one application agent may be working far ahead of another, or may receive a command far in advance of another. Inter-agent communications are accomplished with asynchronous messages (your reply may not return at a predictable interval). Even Plugins, in general, execute independently and compete for resources.

- Convergence. Cougaar processes should tend to converge upon a single answer, rather than alternate/cycle between answers or (worse yet) diverge into chaos. The infrastructure has some functions in place to act as chaos dampers, e.g. to minimize **infrastructure** effects of reverberation or "ringing," but quite often these sorts of effects are due to implied cycles and feedback loops in the application's business logic.

- Partial answers. Complex application processes can often benefit from partial or low-quality (but fast) estimated answers up front, followed by a deeper analysis later. This allows more parallelism, and can lead to a higher-quality (global) solution as more agents can make progress without having to wait for a final answer. Without estimates, the system often loses most or all parallelism.

- Cooperative and Emergent behavior. Many effects (good and bad) arise as a result of cooperation (or competition, etc) between multiple agents.

- Non-deterministic behavior. Asynchronous processing, particularly in the presence of partial results, often lead to non-deterministic behavior. The solution space of real-world problems is usually vast, and non-trivial planning systems will find different solutions in the space at different times. The goal of the Cougaar infrastructure was to support predictably **high-quality** solutions, not predictable solutions (of any quality).

9.7.2 Problems and Issues

There are a number of related reasons to be analyzing the behavior of a Cougaar application, each with slightly different goals and so a different but overlapping set of techniques.

- Compile-time errors. For the most part, compile-time errors are out of the scope of this document: However, it is important to be aware that Cougaar uses a number of code generators which are driven by non-java files (e.g. ".def" files).

- Load-time errors. Sometimes you will see classloader and/or introspection errors at load time. This is usually due either to a JDK installation problem (e.g. Required "standard" extensions not installed properly) or jar-file version mismatches. For most purposes, CLASSPATH need not be set to run the Node script and the scripts usually do not need to be modified to include additional jars. See the discussion on the Bootstrapping classloader below.

- Debugging obviously broken behavior. Thrown exceptions – usually runtime exceptions – indicate serious problems. Not all exceptions are fatal, but ones that are reported (for instance to the background stream) most often indicate permanent failure.

- Stack traces. Occasionally, diagnostics will print a stack trace that is not actually an exception to the background stream. There is a significant amount of latent debugging code in both the Cougaar infrastructure and in application code that may become active when possible errors are detected. Bug reports should include this sort of information whenever possible.

- Debugging bad behavior (e.g. incorrect application results). The application fails to give the correct result (possibly no result at all), but does not actually error. This is probably the most difficult debugging task of all and will be discussed at length below.

- Optimization/performance analysis. Applications will often exhibit poor performance either by running too slowly, or by scaling unacceptably. While not necessarily a debugging task in the usual sense, this sort of analysis shares much of the same techniques as most.

- Smoke and regression testing. Cougaar infrastructure includes both point regression tests and a few standard test suites for release smoke testing. Development of these tests is an important part of Cougaar application development as well.

- Stressing the system (including artificial stressors). Related to scalability testing and optimization, stressor development requires predicting application behavior under duress and then development of tests that look for good (and bad) performance.

- Analysis of deployed systems. It can be challenging to determine if any non-trivial running system is performing correctly. Especially a system as complex as a widely distributed Cougaar application. Again, many of the same techniques apply here, though minimization of the impact of the analysis itself on the running system is critical.

9.7.3 Debugging and Analysis Techniques

This section is a list of techniques that may be useful for debugging and analysis of a Cougaar application. It is by no means complete, but focuses on approaches that are especially useful for Cougaar.

- Use debugging/testing GUIs. There are a variety of GUI applications that are useful to determine what is going on inside the system, including multiple servlets and several applications. These include the /tasks servlet, and the CSMART Society Monitor, both documented elsewhere.

- Enable or add extended logging. There are several ways to enable additional built-in logging and progress notifications, and you can add your own which rely on the same core facilities. See the System Properties catalog, and the discussion of the LoggingService facility below.

- Enable/add extended assertions. Assertions are generally relatively inexpensive and can be invaluable in debugging. Incorporation of assertions in your original application code can save a great deal of time later when something goes wrong. Log4j includes a basic assertion package similar to the JDK 1.4 package.

- Enable alternate modes of operation. There are often several ways to accomplish a given application task. Maintaining several options allows you the freedom of selecting from a suite of approximately equivalent options at various points to result in a predictable effect on your problem. For instance, a good trick is to have both a general-purpose plugin and one programmed to do only the minimal job required by your regression/smoke tests. Replacing the generic business logic with a hard-coded module should give you the same answer – if it doesn't, you've narrowed the scope of the problem.

- Simplify (society, node, agent). If the problem is too complex or takes too long to run, consider simplifying it. It is often important to simplify in several different ways to narrow down to a single error: for instance, you might need to cut down on the number of agents, reduce the number of different types of tasks each agent is doing, and reduce the total problem size in order to make the analysis tractable. As your testing proceeds, you can add pieces back into the problem a little at a time until you are back up to the full problem size.

- Node/agent organization. Sometimes problems only happen when two agents are in the same node or running on the same machine. Sometimes only when two agents are on different machines or VMs. Try distributing your agents differently to elicit different behavior. This technique can help debug serialization problems and issues where there are objects shared between multiple agents in the same VM (e.g. static variables, etc).

- Special purpose plugins. Write your own debugging plugins, loading them at startup or even dynamically. Special-purpose plugins can be used to gather additional information, watch for bad behavior or inject stressors into your system. Such plugins can be then require far more resources than you would want to allow for a "real" plugin, but may be able to detect problems that would otherwise go unnoticed. You can also incorporate this sort of functionality into your real software, but make certain that it is disabled by default.

- Patch/restart cycles. Once you've narrowed down your problem to a minimal set, you may end up resorting to patch-debug cycles. Since startup and setup time is often significant, you will certainly want to minimize this, so use debugging tools (e.g. jdb or your favorite IDE) and the other techniques wherever appropriate.

- Contract testing. Consider wrapping questionable components in "test jigs": when possible, test a component with a tool that looks (to the component being tested) like the rest of the society. Also possible is an "in-vivo" jig, which acts as a proxy between a component and the rest of the system, testing inputs and outputs as it functions and reporting on any suspect behavior.

- Standard test configurations. Develop standard test suites and application configurations. This covers both synthetic data (for reproducibility), and a range of application configuration sizes and purposes. This allows analysis along more than one axis and makes narrowing the scope of problems easier. These configurations can form the basis of your regression tests.

- "Full thread dump." Most Java Virtual Machines allow a user to signal the VM (even without a debugging session) to list the current stacks of all active threads. This can be critical to resolving java-thread-level deadlocks. Solaris and Linux use the "Control-\" keystroke or the SIGQUIT signal (e.g. "kill –QUIT <vm process id>"), and Windows uses "Control-Break". For a guide to figuring out stack traces, see the following document: http://developer.java.sun.com/developer/technicalArticles/Programming/Stacktrace/ .

- "/tasks" servlet for blackboard object viewing

- CSMART Society Monitor tools for blackboard object viewing (see the CSMART User's Guide).

- Host system analysis tools. The host OS will usually have a variety of tools available to analyze running applications.

  – Win32 has "system monitor," "resource meter," and "net watcher." Wintop and Process Explorer as well as a variety of other third-party tools are available (*e.g.*, Norton utilities, *etc*.).

  – Unix has ps and top/gtop, perfmeter, netstat, etc as well as very low-level tools for doing things like network traffic analysis.

- Name threads. If your code creates threads (and most Cougaar components should instead use the Cougaar ThreadService), make sure to give them names that are unique and meaningful to people debugging the code. Note, however, that many parts of the infrastructure implement their own schedulers and thread/resource pools, so this information may not be quite as available as you might hope.

- Traditional debugging methodologies. In particular, "Extreme Programming" (XP) techniques seem to be well suited for Cougaar development. There is a great deal of ground to be covered here: an especially useful guideline is to not even attempt to fix a problem until you can implement a regression test that identifies it.

9.7.4 System Properties

System properties control a range of infrastructure settings useful for debugging a Cougaar system. Most attributes of Cougaar can be tuned by either adding or removing components in the agents, or by setting different –D System properties. For a complete listing of these properties, see doc/api/Parameters.html (when the cougaar-api.zip package has been installed) which is built as part of the standard javadoc of the core infrastructure package. The full catalog includes hotlinks to the full API (javadoc) documentation of each class or method.

9.7.5 Logging Facilities

Cougaar provides a LoggingService (see section **Error! Reference source not found.**) that provides a generic logging API for components to use. The logger is preferred to simple "System.out" calls and custom log implementations for several reasons:

- The LoggingService can be shared and managed across all Cougaar components

- System.out calls clutter the screen, are not saved or forwarded to the UIs, are are difficult to disable. System.out calls are not formatted for an external tool to parse (e.g. no timestamps, no debug/error indication, etc)

- The LoggingService may be wrapped by Cougaar binders for future enhancements.

New components are strongly encouraged to use the new LoggingService API. Core infrastructure and sample plugins will be modified over time to add use of the LoggingService.

The section on the LoggingService covers the basic usage and configuration details.

Here are some recommendations for use of the LoggingService when focused on debugging:

- Carefully distinguish between the choices of logging levels (DEBUG, INFO, etc). Incorrect use of logging levels can make the log files difficult to understand.

- Make certain that the logged messages are informative.

- In some situations you may need to add logging calls to focus on a very specific, possibly hard-coded context. After debugging this code, make sure to clean up the logging calls to leave behind only the useful logging calls and (optionally) assertions.

- Excessive logging can impact performance. Sometimes logging alters timing enough to change the behavior of the system. This can be true when logging to a terminal screen, a file, or even with logging disabled.

- A common mistake when writing debugging messages is to construct large string arguments (e.g. "informative messages") even when the logging level is set to drop the message. This can cause serious performance problems. The LogService now contains a flag to enable checking for such “unwrapped” log calls. See org.cougaar.util.log.log4j.LoggerImpl. Be sure to wrap logging call with "log.is*Enabled()" calls, such as:

```
    if (log.isDebugEnabled()) {
      log.debug("your message");}
```

- Another similar problem is that over-use of logging facilities can force an essentially parallel system into serial behavior, in effect synchronizing on an output stream.

- Shipping code should, by default, be quiet when working correctly. Extraneous progress reports and "expected warnings" often hide real problems.

9.7.6 Guidelines for Developing Debuggable and Testable Code

There are a number of programming practices that increase the likelihood that your code is maintainable, debuggable and understandable by others. There are innumerable well-documented human and machine processes designed to further this goal. Rather than supporting any one process, we will point out a few specific practices that have proven their worth over the course of developing the Cougaar infrastructure.

- Regression tests. Whenever possible, develop unit/point/regression tests, especially for the foundation classes of your code. This is probably impossible (or at least impractical) for complex software like Cougaar applications and GUIs, but the more testable a piece of software is, the more likely it is to be maintainable and usable.

- Documentation. Both javadoc and in-line (descriptive) comments about how the code works. Also high-level documentation on the organization of your code.

- Examples of the use of your code, especially utility and base classes. Sample code should always be compilable.

- Source/Version Control systems are essential. It is critical to tell what has changed and who changed it. The Cougaar infrastructure team uses CVS, but there are plenty of other options available. (All Cougaar CVS trees are publicly readable at http://www.cougaar.org)

- Automatic build support is nearly essential. For instance, the Cougaar build and release process does nightly javac builds from source control, with including the running of all regresssion tests, compiling of examples, generating javadoc and the building and publishing of the complete release package.

9.7.7 Sources of Assistance.

- There is a significant community of Cougaar developers, so there are a number of resources available to help you solve your problems. As listed above, these include the Cougaar Bugzilla (http://bugs.cougaar.org) and the various mailing lists, such as cougaar-developers@cougaar.org

9.8 Standard Error / Exception Handling

9.8.1 Introduction

The Cougaar architecture utilizes many java exceptions from the java.lang, java.io, java.util. java.net, rmi and swing packages as well as others. The architecture also provides and utilizes Cougaar specific exception classes. This document will attempt to describe these exceptions and their general usage patterns as found in the architecture. The document will also discuss when exceptions should be thrown and caught.

9.8.2 Brief Overview of Java Exceptions

In the Java language, exceptions in a program can be thrown by the Java Virtual Machine (JVM) if a semantic constraint is violated or an exception can be thrown explicitly in program code. All exceptions should extend from the java.lang.Throwable class. Two common subclasses of Throwable are Error and Exception. Error classes are used to signal unrecoverable system conditions whereas Exception classes are used to signal program problems that are generally recoverable and should be handled somewhere in the system. However, an exception to that rule is the use of a RuntimeException, which generally falls under the Error category even though it is a type of Exception. The java language provides a hierarchy of predefined Errors and Exceptions to be used by developers. All errors and exceptions should be instances of a class so that the object can maintain information about the error or exception that has occurred. This information is important to the code that actually handles or catches the exception. In addition to using the Java exception classes, the Cougaar architecture has defined some of its own Exception classes.

9.8.3 Cougaar-specific Exception Classes

Cougaar has defined some of its own exception classes that are used throughout the architecture. These exceptions allow the architecture to provide specific information about a given problem in the system. A handful of these exceptions extend RuntimeException, which signals a significant problem in the running society that cannot be handled or remedied by exception handling code. In these cases, the user or the system administrator must review the error and remedy the problem, which is most likely a setup problem. However, most Cougaar exceptions are non fatal and they provide state and condition information to the calling code. This information should allow the calling code to handle the exception properly. Exception handling code in Cougaar should be unique to each Component. Only the Component has the knowledge to understand the impact of the exception and make a decision about how to remedy the problem. The goal of an exception handling block of code should be to allow the system to try to recover from the problem and continue processing. This is important to the continuous distributed processing of a Cougaar application.

Some components of the Cougaar architecture utilize standard out (System.err.println and System.out.println) and logging facilities to provide various levels of debugging output and user information. Many Plugins use Java arguments to activate their debugging output. This output can range from simple informational output to a very detailed description of what's being processed in a given Plugin. Detailed output is often accompanied by a Java stack trace to aid in debugging. Unlike a formal Error being thrown, these outputs allow the system to continue processing while communicated important runtime issues to the developer and the user.

Cougaar developers are encouraged to utilize Java's Exception handling mechanisms and create their own subclasses of Error and Exception when appropriate. The use of Exception and Error handling will increase the overall readability, maintainability and survivability of Cougaar applications.

9.8.4 Guidelines for Throwing Exceptions and Errors

When an unexpected or problematic condition occurs within a Cougaar application an Exception or an Error should be thrown using a 'throw' statement. Any method that may cause an Exception to be thrown must either handle that Exception using the 'catch' mechanism, or the Exception must be declared in the method's throws clause. Note that as mentioned above, Errors and `RuntimeExceptions` are exempt from being defined in `throws` clauses and are exempt from being handled.

All Exceptions and Errors should be subclasses of `Throwable`. The `Throwable` class provides the following methods that are useful for exception handlers and general debugging output to gather useful information in the context of the Exception.

Once an Exception is thrown, the exception propagates up the call stack at run time until it reaches an exception handler for that particular class of the exception (or its superclass). Unhandled exceptions will eventually reach the `uncaughtException` method invoked for the ThreadGroup that is the parent of the current Java thread.

Cougaar developers are encouraged to throw meaningful exceptions where necessary to provide callers as much information as possibly to help rectify the problem. For example, Services should freely throw exceptions when clients pass in unexpected or illegal arguments to defined methods. This lets the client Component know that there is a specific problem with their request rather than simply receiving a null in return. Developers should carefully evaluate the severity of a problem when determining what kind of exception to throw. For example, if the problem is not fatal to the society run, the exception should not extend `RuntimeException`. Developers should also include as much useful information about the exception as possible by defining a string descriptor of the exception circumstances along with a stack trace. The developer should also make sure to dump the stack trace information in order to effectively aid in debugging the problem especially when utilizing the logging or standard out mechanisms.

Errors should be thrown when a Cougaar Component reaches an unrecoverable state. Given the size and the distributed nature of Cougaar societies every effort should be made to avoid unrecoverable errors especially since causing the JVM itself to die will cause the death of other components and more importantly other Agents running in the same Cougaar Node.

9.8.5 Guidelines for Catching Exceptions

In Cougaar, components should strive to handle exceptions locally when possible. For example, when a Plugin incurs an exception while executing, it is likely that the Plugin itself is the only piece of code that contains the knowledge to determine the severity of the exception and remedy the problem. If the Exception continues up the call stack, the program will loose the ability to handle the exception in a meaningful way. As Components make requests to the infrastructure or other components and services, they should encase their requests in try/catch blocks to immediately resolve problems stemming from illegal arguments, `ClassCast` and `NullPointer` exceptions among others.

All Java Exception classes are formed from a hierarchical relationship. A single exception handler can be generated to catch all exceptions from a class and its subclasses. Alternatively, a series of exception handlers can be used, each handling exceptions for individual exception subclasses as shown in the code example below (however note that users should generally be using the logging service, as in `log.error("I got an exception", ce);`).

```
try {
       // use the LDM Classloader
       Class myClass = loadClass(className);
       myPol = (Policy) myClass.newInstance();
     } catch ( ClassNotFoundException ce ) {
      ce.printStackTrace();
     } catch ( InstantiationException ie ) {
      ie.printStackTrace();
     } catch ( IllegalAccessException iae ) {
      iae.printStackTrace();
     }
```

10 References

10.1 Related Cougaar Documents

Cougaar Architecture Document (ref. www.cougaar.org)

Introduction to Cougaar Programming (Cougaar Tutorial, ref. www.cougaar.org)

10.2 Mobility / Dynamic Components

Library of Journal Papers on Mobile Agents: http://citeseer.nj.nec.com/Agents/MobileAgents/

Recommended Papers:
 Pros/Cons of Agent Mobility: http://citeseer.nj.nec.com/chess95mobile.html
 Overview of Mobile Agent Security issues: http://citeseer.nj.nec.com/jansen99mobile.html

Aglets, IBM's Mobile Agent architecture: http://www.trl.ibm.com/aglets/index.html

10.3 Java Style Guide

Draft Java Coding Standard, Doug Lea: http://g.oswego.edu/dl/html/javaCodingStd.html

10.4 Java Beans

Presenting JavaBeans, Michael Morrison, Sams.Net Publishing, 1997, ISBN 1-57521-287-0.

JavaBeans web site, including BDK's: http://java.sun.com/beans/

10.5 Error and Exception Handling

Mastering Java 1.2, John Zukowski, Sybex, 1998, ISBN:0-7821-2180-2

The Java Language Specification, James Gosling, Bill Joy, Guy Steele, published by Addison-Wesley, 1996 Sun Microsystems, Inc,, ISBN:0-201-63451-1

Java Performance Tuning, Jack Shirazi, O'Reilly and Associates, Inc., 2000, ISBN: 0-596-00015-4

10.6 Object Database Management Group's OODBMS API

The Object Database Standard: ODMG 2.0, R.G.G. Cattell, Douglas Barry, Dirk Bartels, Mark Berler, Jeff Eastman, Sophie Gamerman, David Jordan, Adam Springer, Henry Strickland, and Drew Wade, published as part of The Morgan Kaufmann Series in Data Management Systems, Jim Gray, Series Editor, 1997, ISBN 1-55860-463-4. (Contains the definition and detailed description of the programming interface to object databases, including the Java and C++ language bindings.)

Interactive Object Databases: The ODMG Approach, Richard Cooper, published by International Thomson Computer, ISBN 1-85032-294-5. (Provides a tutorial teaching use of the ODMG API. The included CD-ROM contains the software needed to run the book's examples.)

10.7 Object-Oriented Design

[L86] Henry Lieberman. *Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems.* OOPSLA '86 Proceedings. ACM Press, September 1986.

Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1988.

Grady Booch. *Object Oriented Design with Applications.* The Benjamin/Cummings Publishing Company, Inc., 1991.

James Rumbaugh, et al. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

I. Jacobsen et al. *Object-Oriented Software Engineering.* ACM Press, 1992.

S. Shlaer and S. Mellor. *Object Life Cycles: Modeling the World in States.* Prentice Hall, 1992.

Robert Orfali, Dan Harkey, and Jeri Edwards. *The Essential Distributed Objects Survival Guide.* John Wiley & Sons, Inc., 1996.

Desmond D' Souza and Alan Wills. "Catalysis - Practical Rigor and Refinement Extending OMT, Fusion, and Objectory". ICON Computing, Inc., 1996.
http://www.iconcomp.com/papers/catalysis/catalysis.frm.html

Erich Gamma, et al. "Design Patterns, Elements of Reusable Object-Oriented Software", Addison Wesley, 1995.

11 Glossary

Agent		► Society, Node, Plugin
A software entity that communicates with other Agents to model domain- specific functionality. Agents are components of Nodes, and all Agents in the Society have a unique name in the global NameService.		
Agent Message		► Agent, Message Transport
A Message sent between Agents, such as the transfer of Tasks from one Agent to another.		
Aggregation		► PlanElement
An Aggregation is a type of PlanElement that contains a Composition that defines how multiple input tasks are to be satisfied. An Aggregation is a subclass of PlanElement so that it can be recorded in the Blackboard to facilitate tracing multiple Tasks to a single combined Task. Aggregation is also used to describe the process of combining the AllocationResult of the several subtasks of an Expansion to compute the AllocationResult of the Expansion itself.		
Alert		► Blackboard, LDM
An LDM object intended to capture a need for a human user's intervention with a running Cougaar system. A Plugin is expected to generate an alert when it needs information or approval from a user, or when it needs to inform a user of a condition within the society.		
Allocation		► PlanElement, Task
A plan element representing the act of associating (allocating) a task to an asset, either a physical asset or an organizational asset. In the case of allocating to an organizational asset, the task along with its preferences is forwarded by the infrastructure to the associated agent for further processing. The allocation structure contains a "reported" result and an "estimated" result. When the "estimated" result is filled in on the tasked agent, the infrastructure automatically forwards this value as the "reported" value to the tasking agent which can subscribe and react to this report.		
AllocationResult		► Aspect, Preferences
A structure representing the real, estimated or predicted results of an allocation. The result contains details on all aspect dimensions specified in the associated preferences on the allocated task, as well as information about the success/failure of the allocation itself. An Allocation contains an AllocationResult that specifies exactly how a specific task will be accomplished. It specifies measures computed with respect to the requirements defined in a task (specifically, the task preferences).		
Allocator		► Plugin, Task, Asset
A type of Plugin that matches a Task to an Asset which will complete the Task. Inputs to this Plugin are undisposed Tasks and ouputs are in the form of Allocation PlanElements. Allocators should strive to find the optimal usage of Assets as the Tasks evolve and change over time.		
ALP		► Ultra*Log, Cougaar
Advanced Logistics Program, a DARPA sponsored 5-year effort to design, develop and demonstrate an end-to-end DoD prototype logistics system based on a distributed intelligent architecture. See http://www.alpine.bbn.com and http://www.cougaar.org		

Aspect		► Task, Message Transport
A measure of a given task along one of several dimensions. Aspects can be independent, that is, directly settable in a task, e.g. START_TIME, END_TIME, COST, QUANTITY, or dependant, that is, based on aggregate or derived behaviors, e.g. RISK, DANGER, CUSTOMER_SATISFACTION, INTEGRITY. Also used in the context of Aspect-oriented Programming (AOP), see http://www.parc.xerox.com/csl/projects/aop/		
Assessor		► Plugin, Blackboard
A type of Plugin that monitors the Blackboard and external resources such as databases that reflect the status of the real world to watch for changes that may affect the plan, inconsistencies in the plan, scoring threshold violations, etc. This plugin may initiate system actions when it detects a need for replanning.		
AssetAssignment		► Asset, Directive
An AssetAssignment is a specialization of Directive used to assign an asset to an Agent. AssetAssignment directives are sent to other agents; they are not blackboard objects.		
AssetRescind		► TaskRescind, Directive
Rescinds are a specialization of Directive used to terminate activity associated with a specific Directive. Internal processing by an Agent in response to a Rescind may result in the creation of new Rescind orders, reflecting the termination of all Directives associated with the rescinded Directive.		
AssetTransfer		► PlanElement
An AssetTransfer is a type of PlanElement that triggers the sending of an AssetAssignment (a specialization of Directive) to assign Assets to other Agents. An AssetTransfer is a subclass of PlanElement so that it can be recorded in the Blackboard to facilitate tracing a Task to the assignment of an Asset to an Agent.		
Binder		► BindingSite
A Binder is an object that 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.		
BindingSite		► Binder
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.		
Blackboard		► LogPlan, Agent, Plugin
The part of an Agent that holds elements like Tasks, Assets, and PlanElements. It's 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. All plugins of an Agent share the Blackboard.		
Community		► Society, Agent
A set of Agents in the same Society with shared properties or goals. Some examples of these defining characteristics are shared functionality (e.g. transportation modelling) or system properties (e.g. trust or authority).		
Component		► Binder, JavaBean
A Component is a plugable software entity that has an existence and identity separate from any other. A Component may be required to implement a specific interface (ChildAPI) so that the Parent container and/or the Binder can invoke the Child's methods.		
Component Message		► Node, MessageTransport

A Message sent to a Node telling it to add or remove a particular Component, such as an Agent.		
Composition		► Task, PlanElement
A structure representing the joining or aggregating of several tasks into a single task for processing as a single entity. A Composition contains a single Task that satisfies multiple Tasks and their Aggregation PlanElements.		
Confidence		► AllocationResult
A value between 0-1 representing the expected closeness between a given allocation result and the result that would have been given had the allocation proceeded all the way to full decomposition to physical assets.		
CONOPS		► ALP
Concept of Operations		
Constraint		► Workflow, Task
A structure on a workflow representing constraints on the allocation results on subtasks of that workflow. These constraints may be expressed as absolute (e.g., a given aspect value may not exceed a particular level), or relative (e.g., a given aspect value of one task may not exceed an aspect value of some other task). Support exists for expressing constraints on arbitrary aspects on tasks, and for detecting violations of those constraints, but the resolution of the constraint violations is the responsibility of the plugins.		
Directive		► DirectiveMessage
Directives are contained by a DirectiveMessage. The distinguishing feature (or extension) of a Directive (as compared to a Message) is that a Directive is related to a Plan. All Directives are related to one and only one Plan.		
DirectiveMessage		► Directive, AgentMessage
DirectiveMessage extends AgentMessage. A DirectiveMessage provides a way to package up Directives and send them to another Agent.		
Disposition		► PlanElement
A Disposition PlanElement indicates that a task has been disposed of in a non-standard way. There are several common uses of Disposition. The most common is to indicate a failure to allocate the task in a way that meets the minimum requirements as specified by the task. This is accomplished by creating a Disposition with an AllocationResult with the attribute isSuccess = false. The other significant use of Disposition is for when the task is trivially accomplishable. For instance, if the task is by definition successful (“Fix all your broken trucks” when all your trucks are working), already accomplished (“Deploy to X” when you are already there), or otherwise requires no resources be consumed or allocated against.		
Distributor		► Blackboard
The software in the Blackboard that manages the Subscriptions and distributes Blackboard modifications to the Subscriptions.		
Domain		► ALP, Planning
A set of data types that define the language for Agents to model particular problems. For example, the “planning” Domain defines “Task” and other data types for the modelling of generic planning problems.		

Estimate		► AllocationResult, Task
An estimate is an AllocationResult that is generated by the society without any actual allocation being made. This is a "no commitment" estimate that may be used for planning, but must be understood to be transient. A predictor (below) is an example of a mechanism for generating an estimate.		
Expander		► Plugin, Workflow, Task
A type of Plugin that expands Tasks into implied sub-tasks necessary to complete the Task. Expanders contain specific knowledge regarding task expansion within a specific domain area. Expanders create a Workflow that contains the parent Task (incoming) and the new sub-tasks (Tasks) as well as Constraints between the sub-tasks. The parent Task and the Workflow are used to create an Expansion PlanElement which is published to the Blackboard (the Expansion is this Plugin's output).		
Expansion		► PlanElement
An Expansion is a type of PlanElement that contains a Workflow of related subtasks and contraints that define how the input is to be accomplished. An Expansion is a subclass of PlanElement so that it can be recorded in the Blackboard to facilitate tracing a Task to its expanded Tasks.		
GLM		► LDM, Domain
"Generic Logistics Model", a Domain that includes data types and utilities for the creation of basic logistics modelling.		
Host		► Node
A computer capable of network communication. A host can used run one or more Nodes and/or provide other services, such as database access.		
LDM		► Domain, ALP, Assets
"Logical Data Model", a set of Object Factories, including Asset primitives and basic data types. Domains use LDMs.		
LogPlan		► Blackboard
A Logistics specific Blackboard. A LogPlan usually consists of tasks, oplans and policies that contain the logistics requirements, assets that complete those tasks and schedules and scores for completing the tasks.		
LogicProviders (LPs)		► Blackboard, MessageTransport
LogicProviders are infrastructure Plugins that forward information and interact between the Blackboard, the MessageTransport, and the infrastructure.		
Metrics		► Survivability, Assessment
Metrics are statistics about components within a Cougaar society. For example, how many Tasks are in the Blackboard or how many messages have been sent from a given Agent.		
Node		► Host, Agent,
The basic Cougaar process, running on its own JVM, which can run Agents and provide communication support for those Agents to the rest of the Society.		
Notification		► Directive
A Notification is a specialization of Directive that is used for synchronous event messages from one Agent to another.		
Plan		► PlanElement, Blackboard
A Plan (also called a logistics plan or a LogPlan) contains one or more PlanElements.		

PlanElement		► Aggregation, Allocation, AssetTransfer, Expansion, Plan
A PlanElement references a Task and specifies how that Task is to be accomplished. A PlanElement is an Allocation, an Expansion, an Aggregation, or an AssetTransfer.		
Plugin		► Agent, Component
A Component of an Agent that uses Services to perform a portion of the basic domain functionality for that Agent. Most Cougaar developers will create Plugins to capture the "business logic" of their Agent.		
Policy		► Directive, LDM
A specialization of Directive used to communicate policies or guidance, generally from outside the Cougaar Agents, to Agents. An LDM object containing a series of parameters and rules guiding the behavior of a particular Plugin or agent. A policy can be a local object, or received from a superior and dynamically updated.		
Predictor		► Asset, AllocationResult
A structure on an organizational asset enabling a local computation of an allocation result on a task representing how an associated agent would compute that allocation. It is expected that the Predictor will generate allocation results containing confidences reflecting the expected fidelity of the prediction relative to actual allocation. An Agent provides a predictor when it passes a copy of itself as an organizational asset at hookup time, though this value can be updated dynamically through the course of the running society. (Note: this concept was previously referred to as an "Estimator.")		
Preference		► Task
A structure on a task specifying the desired handling of a task as expressed by the tasking organization. A task may contain multiple, one or no preferences. An individual preference describes a single aspect, a mapping of aspect space into score space, a weight allowing for the comparison of multiple preferences and 'threshold' information indicating values in score space beyond which the tasking organization would rather the allocation not take place.		
PropertyGroup		► Property, LDM
An element of an Asset that defines a property of that Asset, such as "MobilityPG". Assets can a set of PropertyGroups to define both the data type and it's current status.		
PropertyProvider		► LDM, Property
An Object capable of creating PropertyGroups, used in Asset creation by an LDM Factory.		
PrototypeProvider		► LDM, Prototyping
An Object capable of creating Asset Prototypes, used in Asset creation by an LDM Factory.		
Prototype		► LDM, PrototypeProvider
Prototypes are "templates" for Asset instances that define the default Asset PropertyGroups. For example, a "Truck" is a Prototype that defines basic Truck properties (e.g. "CargoVehicle"), and a specific Truck instance, "Truck123", can override non-standard PropertyGroups of it's basic Truck definition.		
PSP		► UI, Plugin, Servlet
"Plan Service Provider", the old concept of an HTTP listener loaded by the PlanServer that responds to web requests with HTML or serialized data. The PSP concept has been replaced by Servlets (cf. section 6).		

QuO, QoS		► Message Transport
Quality Objects (QuO) is a framework for providing quality of service (QoS) to software applications composed of objects (especially CORBA-based objects) that are distributed over wide-area networks (WANs). Such networks include the Internet and many military data networks. http:// http://www.dist-systems.bbn.com/tech/QuO/		
Rescind		► Task
The act of removing a Task from the tasked organization, causing all derived Tasks to automatically recursively be rescinded as well.		
RootFactory		► LDM
An object creation factory used to create basic Cougaar LDM objects.		
Servlet		► ServletService, UI, PSP
A class thet handles HTTP(S) requests. Servlets are registered by the ServletService and used to create web-based User Iterfaces.		
ServletService		► Servlet, Service, UI
A Service within an agent that allows a component to register a servlet with a URL path.		
ScenarioTime		► ALP, What If
The forward-moving sense of time within a given executing ALP scenario. The time is maintained globally throughout a society. In an ALP/Ultra*Log demonstration or "What-If" scenario, the scenario time is maintained by a distributed server and driven by UI; in a real-time ALP/Ultra*Log society, scenario time is driven from real clock time.		
Score		► Preferences, Task
A unitless dimension in which all preferences map represent their desired aspect values.		
Service		► Service Model, Component Model
Service is an API for a facility that may be requested by a Component. Services are always Java interface classes - that is, a Service is named by its Java interface. There are no specific requirements for how a Service is defined.		
ServiceBroker		► Service Model, Component Model
A ServiceBroker is essentially just an object that 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.		
ServiceImplementation		► Service Model, Component Model
A Service Implementation is an implementation of a Service class constructed by a ServiceProvider, usually for a specific client component.		
ServiceProvider		► Service Model, Component Model
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.		

Society		► Community, Agent
The set of all Agents that can talk to one another, as defined by a shared set of contact addresses (host and port or URLs).		
Society Awareness		► Society
The ability of Agents in a Society to monitor both internal and external activity and dynamically respond to that activity.		
Task		► Interactive, Agent, Blackboard
A Task is a specialization of Directive. Task is the essential "execute" directive, instructing a subordinate or service provider to plan and eventually accomplish a logistics activity. An MPTask is a task that has multiple parent Tasks.		
TaskRescind		► Task, Directive
Rescinds are a specialization of Directive used to terminate activity associated with a specific Directive. Internal processing by an Agent in response to a Rescind may result in the creation of new Rescind orders, reflecting the termination of all Directives associated with the rescinded Directive.		
Trigger		► AssetRescind, LDM
An LDM object representing an action to be taken when a particular condition on a specified set of LDM objects is detected. For example, an alert may be generated when a particular threshold of some sort is detected, or replanning/rescinding a plan element as a result of some inconsistency being detected or some assumption being changed.		
Ultra*Log		► ALP, Cougaar
Ultra*Log is a Defense Advanced Research Projects Agency (DARPA) sponsored research project focused on creating survivable large-scale distributed agent systems capable of operating effectively in very chaotic environments. https://www.ultralog.net		
White Pages		► Yellow Pages, Naming Service
White Pages (WP) refers to accessing objects in the naming service by "name".		
Workflow		► Task, PlanElement
A Workflow is set of Tasks (also called subtasks) that must be completed in order to accomplish a higher-level Task (also called parent task) and the temporal set or causal relationships (constraints) between the subtasks. A Workflow is part of an Expansion and holds the subtasks of that Expansion and the constraints on those tasks.		
Yellow Pages		► White Pages, Naming Service
Yellow Pages (YP) refers to the attachment of attributes to named objects (cf. WP) and the ability to search for objects by filtering on the attributes.		