

Complementary Methods for QoS Adaptation in Component-based Multi-Agent Systems

John Zinky, Richard Shapiro, Sarah Siracuse

BBN Technologies

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

http://Cougaar.org

Abstract

In the hostile computing environments typical of large, distributed, multi-agent systems, dynamic adaptation to changes in quality of service (QoS) is essential to the survival of the system. To some extent, a society of agents can control its own adaptation to changes in QoS at various levels - individual, community, society - by the introduction of a new class of QoS-managing agents. But without access to the system infrastructure, such adaptation is of only limited use. By implementing an agent system on a component-based middleware with awareness of resource constraints and effective mechanisms to work around those constraints, the survivability can be significantly enhanced. Cougaar [Cognitive Agent Architecture] [1] is an example of such a middleware. It offers several important robustness methodologies as part of its core structure (e.g., services and binders) and also enables others that would otherwise be difficult to implement in modular and maintainable way (e.g., open implementations and aspects). A suite of these methods that has proven to be effective at QoS management is described here, illustrating both the diversity of options as well as the power of combining them. These complementary methods can be configured to match any given fielded environment, trading off the overhead of each adaptation against its benefit. The overhead of these adaptations is discussed in the context of security, failure recovery, and Denial of Service (DoS) attacks.

Keywords

Quality of Service (QoS), Components, Multi-Agent Systems (MAS), Denial of Service (DoS) Defense, Middleware, Binders, Aspects

1. Introduction

In order for large-scale agent societies to be deployed effectively in real-world environments, they must be able to adapt to changes in their environment. The underlying computer and communication resources are constantly changing for a number of reasons, including equipment failures, competition from other consumers, and security attacks. In addition, the application's Quality of Service (QoS) requirements can change. QoS encompasses several distinct facets of system management, including robustness, load balancing and security. QoS adaptation enhances the functionality of agent-based infrastructure in order to deal

with the dynamic failures and constraints of the underlying resources. Wide ranges of adaptation techniques exist for the various facets of QoS, each with algorithms that work best under specific conditions. The goal of QoS-adaptation is to identify the current conditions and to apply the appropriate technique for that set of conditions. In order to support QoS-adaptation, an agent-based infrastructure therefore needs access to the system data that defines the conditions, as well as a rich set of adaptation techniques from which it can choose.

Effective QoS adaptation is implemented as layers of control loops at different time scales and system locales. Some parts work as sensors, others as actuators or controllers. Controllers at lower layers might summarize behavior for higher layer sensors, or might interpret higher layer actuator signals as additional sensor input for the guidance of the lower layer control algorithm. While on the surface this looks like a recursive structure that could reuse the same mechanisms at each layer, the system issues at each layer are dramatically different. Thus, the coarse-grained mechanisms for controlling communities of agents at a high layer are not practical for the control of quick, fine-grained actions of a single agent. The slower, more coarse-grained, distributed adaptations can best be implemented using QoS-managing agents, while the faster, more fine-grained, local adaptations can be implemented more effectively in middleware.

Flexible QoS-adaptation must allow a variety of adaptation implementations to be specified both statically, at configuration time, and dynamically, at runtime. This flexibility is needed for a number of reasons. First, QoS-adaptation consumes resources itself, which in some circumstances can be used more effectively by the application agents. So QoS-adaptations should only be enabled when necessary. Second, the application's QoS requirements change over time, as does the resource availability. Third, in large Multi-Agent Systems (MAS), some communities of agents might be using one set of QoS-adaptation policies while other communities are using completely different policies. A major theme of this paper is the issue of QoS configurability. We believe the approach described throughout this paper is flexible enough to provide efficient solutions in a wide range of contexts, from stable and resource-rich environments to chaotic and resource-contentious ones.

Agent societies could manage their own QoS by using the agent framework itself to create a community of agents whose specific purpose is the management of the runtime

framework. The advantage of using the agent framework is that the same techniques used to create application agents can be reused in the new domain of system management. While these QoS-managing agents could observe and control the underlying resource infrastructure directly, the principles of modularity suggest that an intervening middleware layer should be introduced instead. More specific rationales for middleware will become clear later in the paper.

In general, middleware is used to extend and integrate the services offered by low-level computing and networking resources into a unified run-time environment on which higher-level applications can be built. In the case of resource infrastructure, the middleware will sense the status of resources and allow them to be controlled. In an agent system, middleware can also monitor the agents themselves and determine how they are using the resources. The middleware could be implemented as agents, but can also be implemented using more traditional programming techniques, such as components or objects. The engineering choice of which technique to use to implement any given QoS-adaptation depends on the specific function the adaptation is trying to perform, as well as on the time scale and locale of the adaptation. A comprehensive agent-based infrastructure should allow adaptations both at agent layer, through QoS-managing agents, and also at the middleware layer.

Cougaar [1] is a comprehensive middleware infrastructure for developing societies of distributed agents. Cougaar is a large environment with over 1/2 million lines of Java code written by several organizations. Cougaar has shown that large societies with over a 1000 agents can be constructed for real world applications, such as military logistics. The DARPA sponsored Ultralog [13] program is extending Cougaar to handle chaotic environments in which substantial portions of the computer and networking resources are lost or compromised. As a result Cougaar needs to support a wide variety of schemes for handling QoS adaptation issues, such as security, persistence, dependability, performance tuning, and mobility. These QoS adaptation must be able to be developed independently, and must be configurable and dynamic at runtime. This paper introduces five methods for QoS adaptation supported by Cougaar. These methods, which have been used to develop dozens of QoS adaptive subsystems for the Ultralog program, show how different techniques are necessary at the agent and middleware layers.

To illustrate the issues involved with flexible QoS adaptation, this paper will use as a running example one of the QoS adaptive sub-system created with Cougaar technology for the Ultralog program. The Denial of Service (DoS)-defense subsystem is used to detect and defend against DoS attacks, in which an adversary consumes resources in order to stop the multi-agent application from performing its tasks. A DoS-defense subsystem uses sensors that detect when resource capacity has dropped below a threshold, an indication that the resource might be under

attack. In response, the DoS-defense will attempt to block the adversary from consuming further resources, or switch the application behaviors to a different mode that does not depend on the attacked resource. Nuances in DoS detectors help make it interesting as an example. First, the system itself puts load on resources, which must be accounted for when determining the expected resource capacity. Second, the fact that the system is under a DoS attack could be used to trigger other security defenses, such as the hiding of network traffic patterns.

The remainder of this paper discusses in more detail the important features of QoS-adaptation as it is implemented in Cougaar. Section 2 gives a brief overview of Cougaar, how it is used to create a multi-agent application and how to add additional agents and middleware components for QoS adaptations. The five subsections of Section 3 discuss complementary methods for QoS adaptation that are supported by Cougaar: (1) using distributed agents as sensors, actuators and controllers of a distributed QoS adaptation subsystem; (2) using Cougaar middleware to define new services that expose how the society is using resources; (3) opening the implementations of services to allow QoS sensing and control; (4) using Binders around components to add QoS features without changing the implementation of the component; and (5) using Aspect Oriented Programming techniques to added QoS adaptations that cross-cut multiple components. Section 4 shows how Cougaar societies can be flexibly configured with variable levels of QoS adaptation. It further discusses the overhead of this QoS adaptation and the impact on code size and runtime performance. Finally, section 5 closes with a brief conclusion and reference to future work.

2. Cougaar Multi-Agent Middleware

Cougaar is a product of a multi-year DARPA research project exploring large-scale, scalable, heterogeneous, distributed and survivable Multi-Agent Systems. Cougaar uses a component model to organize both its base functionality and its QoS adaptive features. Cougaar Agents are composed of Plugin components, which attach to an Agent-wide blackboard. Each Cougaar Agent can have a different set of Plugins, which define the Agent's function. Cougaar supports distributing the society over an arbitrarily large number of physical machines, but the society can also be configured to run on a single host. Plugins, Agents, and Blackboards interact in the following ways to implement a Multi-agent society

Plugin - A Plugin is typically a small to medium-sized codification of application business logic. Plugin code is usually used in more than one Agent in an application with instances performing similar functions in each location differing only by host agent and data flow. Within an Agent, Plugins communicate with each other and other components only via the host Agent's Blackboard. Plugin state is generally maintained on the Blackboard. Nearly all application programming of Cougaar is done at the Plugin

level. Plugins are logically independent entities; since they only communicate directly and asynchronously with the Blackboard, they may run in parallel. Plugins do not, however, have an independent, externally visible identity. That is, Plugins are never addressable at any level; they only interact with other entities by data exchange in the context of the Agent within which they exist.

Agent - Cougaar Agents are essentially a collection of Plugins, a Blackboard, a Messaging system, and a set of other support services. Each agent has a unique, addressable network identity and may bind a name to that identity with a high-level naming service. An Agent's application behavior is derived entirely from the set of Plugins from which it is composed. Indeed, early versions of Cougaar used the term *Cluster* rather than *Agent* to draw attention to this feature. Cougaar application developers generally do not program at the Agent level, but rather choose the set from off-the-shelf and custom Plugins to instantiate in each Agent.

Blackboard - Each Agent has at its core a membership-transactional Blackboard with predicate-based publish/subscribe semantics. Logically, the Blackboard is an arbitrary collection of objects. Each Plugin may publish without state, store their state on the Blackboard, or be prepared to reconstruct any state if lost due to Blackboard restore or move. There is also support for distributed (inter-agent) Blackboard reconciliation on rehydration when the application objects support it. The Blackboard is also the center of Cougaar persistence and mobility support: anything stored on the Blackboard will be retained in snapshots and will move when the agent moves. High-performance, general-purpose distributed blackboards are difficult to implement. It is legitimate to view Cougaar as just such an implementation of a distributed data space inhabited by Plugins. In this model, what we are calling *Agents* are merely the blackboard partitions and the specific configurations of the Plugin micro-agents.

The middleware layer of the Cougaar run-time offers services to configure and run distributed agents. A Cougaar Node is a Java process that encapsulates multiple agents and allows them to exchange blackboard objects among themselves and with other remote agents. Cougaar middleware implements a component model, in order to allow the easy addition of services for use by Agents and the infrastructure itself. The middleware layer has a rich offering of services which control basic resources, including threads, communications, and storage. Agents also can access high-level services like naming, web-publishing, and Agent mobility. Many of these services are implemented as components, which could, in turn, be composed of sub-components. Adding QoS adaptation to middleware itself is difficult because the adaptation tends to crosscut the dominant decomposition. This means the adaptive code must extract information from many components and coordinate the interactions between these components. While the basic component model can address some of these issues, it needs to be extended if the adaptations are going to be ef-

fective and maintainable. This requires complementary methods.

3. QoS Adaptation Methods

QoS adaptation is orthogonal to the core functionality of multi-agent applications. The QoS adaptation should work independently to meet the applications QoS requirements despite changes in the underlying environment. In general we believe that complete separation of the application and the QoS adaptation is impossible, but in many situations the interactions can be limited so that the application is only vaguely aware of the constraints imposed by the environment.

This section shows five methods of adding QoS adaptation that is independent of the multi-agent application. The first method uses the agent infrastructure itself to implement distributed QoS adaptation. But the agents need to sense and control how the society is executing in its environment. The remaining four methods show how QoS adaptation can reach into the middleware and expose these QoS properties to the agent-layer.

3.1 Agent-layer adaptation

To begin to address the problem of survivability in multi-agent systems without additional middleware, one approach is the familiar software paradigm of sensors, actuators and managers/controllers. Agents can be introduced into the society to play these three roles.

Sensor Agents monitor the underlying resources and signal the manager agent about their status. One of the main jobs of a sensor agent is to monitor infrastructure services and summarize the observations so that the manager agent can use them. This summary status is translated to a format that can be used for inter-agent communication.

Actuator Agents translate policies from the manager/control agent into a series of commands to the local infrastructure control services. These policies come from the manager agent via the inter-agent communication mechanism. The Actuator must handle both the proper sequencing of the control calls and recovery from errors. When control policies cannot be performed the manager/control agent must be informed of that fact and the reason for the failure.

Manager Agents integrate sensor data and choose an appropriate response. The desired response is translated into a policy, which is sent to the actuator agents for execution. The Manager needs to understand the system that it is controlling, and refrain from attempting to change policies faster than the actuators and sensors can respond.

Together, these three roles can provide a high-level form of QoS adaptation. For example, The DoS defense subsystem changes its behavior based on the intensity and location of DoS attacks. As the level of attack increases, Cougaar nodes will increase the effort used to hide its traf-

fic patterns, for example by encrypting inter-Node messages and generating fake traffic. In addition, Cougar nodes will use different algorithms that reduce the load on attacked resources at the cost of increasing the load on available/un-attacked resources, for example by using compression for inter-node communication, thereby using less bandwidth. Finally, different locations in the Cougar society can be under different levels of attack and will have different levels of response: for instance, one community might be hiding its traffic pattern while another community is acting normally.

In its basic structure, a DoS defense has three types of agents. *DoS Detector Agents* on each Cougar Node detect DoS attacks. Several types of attacks can be detected on different types of resources. A fuse detector monitors the amount of resource utilization, and when usage exceeds a threshold, the fuse detector will not allow the resource to be used again until the fuse is reset. Other DoS sensors detect the unaccounted-for resource load and will fire when the load is above a threshold. The current DoS sensors have an RMI fuse and detectors for attacks on CPU, sockets, and bandwidth. The detector agents publish their attack detections to a Robustness Management Agent.

The *Actuator Agents* enable different levels of defense based on the Attack Status level. Cougar nodes are grouped into robustness communities, each of which has its own robustness manager. Each robustness community has its own Attack Status, and hence its own level of response. DoS actuators convert the threat level into calls to low-level services in the infrastructure.

The *Management Agent* assesses the level of attack based on all the attack detections, and publishes an Attack Status back to actuator agents on each node in the community. In an early implementation, the control policy used by the Manager was a simple count of the number of DoS detections to set a simple threat level. The threat level was exported to other QoS managers, such as the Security Manager, and the actuator agents, which interpreted the threat level and enabled defenses based on the new environment. For example, the higher the threat-level, the more intense the defenses used to hide the society's network traffic pattern. Another feature of threat-management as implemented, was that the automatic control could raise the threat-level but not lower it. Thus, the system reacted quickly to DoS attacks, initiating some defense. But since the defenses shield the system from the DoS attack, the DoS detectors no longer detected an attack. If the raw detections were the only inputs to the control policy, then the controller would think the attack was over and lower the defense. Of course the system would then be under attack again and the defenses would oscillate. Thus, a higher-level policy was needed to reset the threat level and fuses. Figure 1 below shows an overview of the DoS defense.

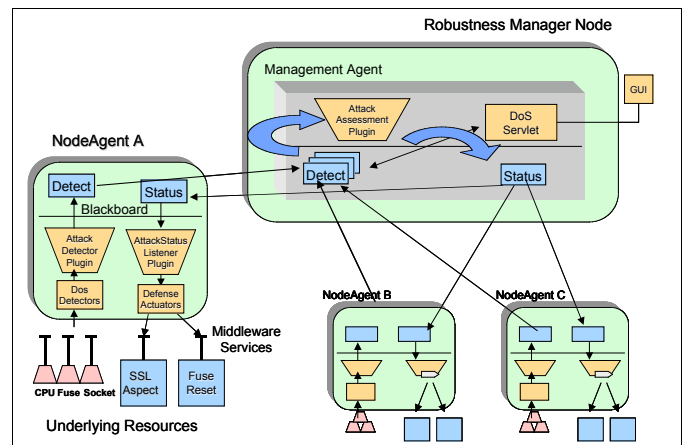


Figure 1. DoS defense

One significant advantage of these sorts of self-managing agents is that they can reuse the same infrastructure mechanisms used to implement application agents. The DoS defense infrastructure is configurable and extensible. The subsystem as a whole is optional, and components can be added or removed at Society configuration time. Each Robustness community within the larger society can have a different set of components, and hence a different protection for vulnerable external communities exposed outside a firewall versus safe enterprise communities. Finally, the architecture allows for adding new types of DoS attack detection and response.

However, the exclusive use of self-managing Agents in a multi-agent society has several important limitations. While these agents could observe and control the underlying resource infrastructure directly, the principles of modularity suggest that an intervening middleware layer of QoS management should be introduced instead.

Timing is another concern. The delay in exchanging information and the amount of time the agents need to sense, decide, and control determines the time horizon for the control loops. Trying to adapt to a phenomena that happens quicker than the control loop's cycle time will result in incorrect adaptation and may actually make the system behave worse. Therefore, the scope of the adaptation must be restricted to fit within the limitation of the cycle time. Groups of distributed agents have a longer cycle time than is appropriate for detailed control of the environment. Self-managing agents need to reason about real-time events, which is not a typical requirement for application agents.

Another consideration is that self-managing agents may take on multiple roles even for multiple QoS adaptations. From a software engineering standpoint, it would be nice to keep these roles and adaptations separate, since this makes them easier to develop and maintain. But separation also allows the QoS adaptation to be configured to match the environment. The agent-based infrastructure should support ways to partition the implementations and selectively enable the partitions at configuration time and run time.

model, which handles the translation and integration of raw data into higher-level data models. Redundant data is ignored and missing data can be inferred. The high-level models match requirements of the QoS adaptive policies, while the raw input data matches the constraints of the collection system.

Queries into the Metrics Service do full forward-chaining, requesting the latest sub-data required for the specific calculation. Callbacks from the Metrics Service do full backward-chaining, so that any change at any level is propagated as far as necessary. High-level data models are created and removed dynamically so that only the information that is being requested by QoS adaptive code is being processed.

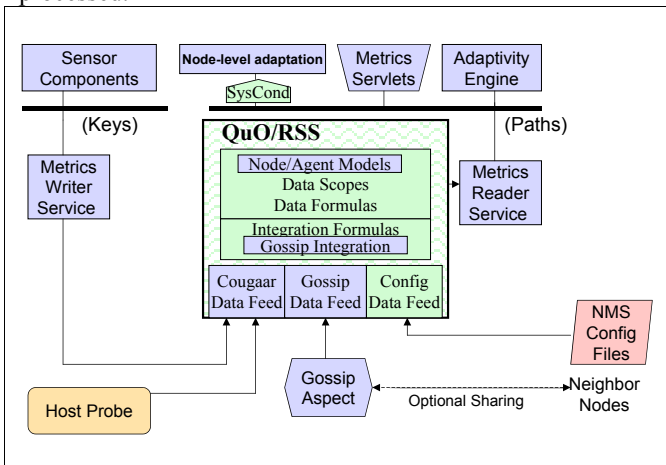


Figure 3. Metric Service for integrating instrumentation

The Cougaar Metrics Services are implemented using a QuO [4] runtime subsystem called the Resource Status Service (RSS). [15] While the original purpose of the RSS was to integrate information from several remote data feeds, the RSS works well as a Cougaar component. Raw sensor information can be fed into the RSS from the Metrics Service interface. The remote capabilities remain available and can be used to gather information from remote Cougaar Nodes or from external sources, for example network management systems. Figure 3 shows the QuO/RSS architecture.

3.3 Open Implementation of Services

Instead of adding dedicated new services to the component model, there are two common ways to add support for QoS to existing services.

One approach here called *in-band* is to add QoS management semantics to the service interface itself and re-implement the client and server components to take advantage of the new QoS management capabilities. The service interface modification can take the form of additional methods, additional parameters to the existing methods, or additional attributes for the data passed via the methods. The advantage of in-band QoS management is that it can be dynamically changed with every use of the service. The disadvantage is that a service interface must be modified for each QoS management facet.

Another approach is to augment a service with a pair of parallel services, one of which instruments the service for its resource load and QoS requirements, the other of which controls the QoS policies of the service. QoS Management components can observe the instrumentation, decide on adaptation policy, and manipulate the control services to activate that policy. In other words, QoS adaptation acts as a feedback control loop [6], with the QoS adaptation policies being part of a control plane for the base functional component. We call this *out-of-band* QoS management because it does not modify the base service, but adds additional services that “open up” its implementation.

The following figure illustrates these two QoS Management schemes. Out-of-band QoS is then described in further detail.

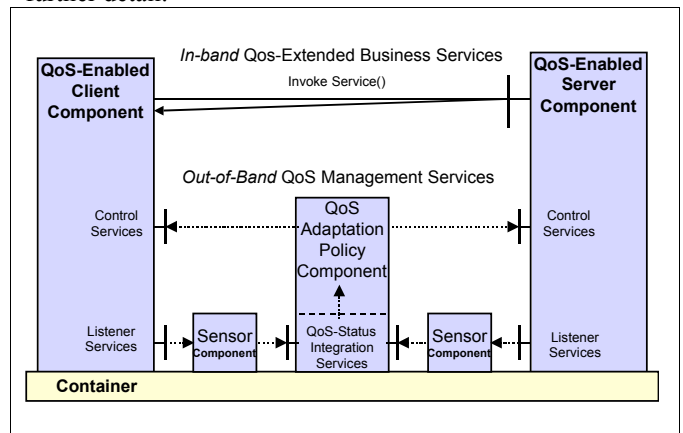


Figure 4. Adding QoS Management to Components

3.3.1 Listener Services

For functional components which implement services which are directly or indirectly related to QoS, a limited amount of information can be made available by invoking callbacks on subscribers when interesting events take place. Here the new implementation of the component is “opened up” to expose the internal events to the outside by offering a listener service. While the base component cannot be observed directly, the listener service allows outside observation. Multiple listeners can subscribe to the service and extract different information. This allows the separation of the functional behavior from the observation of that behavior. A limitation of this approach is that the listener service is fixed at design-time: only the event points exposed by the interface can be observed.

For example, Cougaar uses a Thread Listener Service at various levels of the component hierarchy to provide a listing of all Threads used by components at that level. Since the number of running and pending threads is of interest to QoS management tools, the Thread Services were modified to inform listeners of various events in the life of a thread: when it starts, when it suspends, when it stops, when it's queued or dequeued, etc. By monitoring such changes of state, a component that is willing to do its own bookkeeping will have a good idea of thread utilization at any given level. This information has been used in conjunction with load data to implement a simple but effective

form of thread prioritization. Other points in the containment hierarchy may want different threading policies based on different criteria. The listening component loaded at each level of the containment hierarchy matches the criteria for that place in the hierarchy.

The standard publish/subscribe pattern provides a good approach for designing the service listener mechanism, either by adding subscription calls to the definition of the service itself, or a defining a new parallel service (e.g. a Thread Listener Service). We took the latter approach in Cougar.

3.3.2 Control Services

The additional information provided by new services described above helps in the determination of a current QoS level, but not with reaction or adaptation. To adapt, the model will need new services that control the behavior of existing QoS-related services. In Cougar, a Thread Control Service was defined which allows a component to control various parts of the Thread Service: how many threads can be active at once; the order in which queued threads are dequeued; and the resource relationships between parent and child Thread Services.

Another approach is to use an interceptor pattern. In this style, specific points in the internal workflow of the component are exposed, via callbacks. The listener is allowed to change the raw data in the exposed interface. This allows for unanticipated control schemes. Multiple controllers can contribute modifications if some care is taken to avoid conflicts.

Control services generally require corresponding listener services to be used in a sensible way. This tends to make triads of QoS-related services: e.g., the Thread Service itself, Thread Listener Service, and Thread Control Service. The three services together are managed by a single Service Provider at any given layer in the containment hierarchy. In addition an external QoS adaptation policy component can be added which closes the feedback loop between the listener service and the control service.

3.4 Binders

When services are used to control middleware policies, security considerations require that access to such services be restricted appropriately. But static restrictions that have been compiled into a Service Provider are not compatible with dynamic adaptation. A more dynamic access-control methodology is required here.

For example, as part of responding to a Denial of Service attack, an adaptive service might be added that would refuse all incoming requests, at the TCP level, from a given originating host. This service needs to be protected from malicious abuse by unauthorized clients, where the exact meaning of ‘authorized’ might not be known until runtime.

Cougar added the concept of the *binder* for this purpose. Binders are entities that sit between a component and its container, which can replace the component’s inherited Service Broker with a modified one. A component is assigned a binder at creation time and is bound by that binder for the duration of its existence.

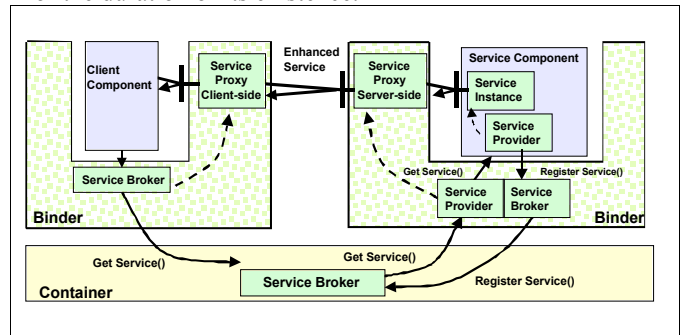


Figure 5. Binder for inserting QoS

Binders can therefore be used either to restrict the component's access to external services (since all service lookups by the component must go through the service broker supplied by the binder) or to restrict access to services the component itself provides (since service registrations made by the component must likewise go through the service broker supplied by the binder). See Figure 5 for a visual.

In the example above, binders would decide at runtime which components should have access to the TCP blocking service and would either deny or provide access.

A binder offers a form of service delegation to the component it binds. The Service Broker provided by a binder can transparently replace a service reference with a proxy that will delegate to that reference only under certain circumstances. Such proxies also provide a natural point at which QoS information can be gathered and QoS control effected. In theory, dynamic stacking of binders used in this way could implement a wide range of QoS adaptation, in a manner very similar to that of composition filters [7]. Figure 6 illustrates this stacking. But Cougar, as currently implemented, only uses them in simpler ways and only for security management.

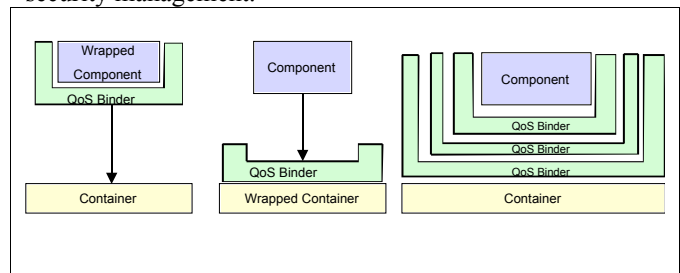


Figure 6. Binder for inserting QoS

3.5 Aspects

The advantages of explicit component models are well known, but they introduce one significant problem in the management of QoS. By its nature, quality-of-service concerns in general and adaptation in particular have a tendency to span multiple components: if one component alters its behavior for QoS reasons, other components with which it communicates also need to adapt at the same time. This kind of *crosscutting* strongly suggests the use of some form of Aspect Oriented Programming (AOP). [13]

In the context of Cougar, general-purpose AOP tools like AspectJ [8,9] weren't well suited, in part for pragmatic reasons (introducing a new programming language wasn't an option), in part because such tools are simultaneously more powerful overall than the requirements at hand demand, and too restrictive in some important ways (e.g., static weaving).

Rather than using a special purpose language we elected to implement a simplified Aspect "pattern" that was tailored for Cougar's needs and that could be written directly in Java. [10] The locus for the use of the Aspect pattern was in Cougar's Message Transport System (MTS), a set of services designed to allow Agents to communicate via message-passing. The MTS is structured as a predefined series of *stations* through which messages pass on their way from sender to receiver. Each station is defined by an explicit interface and instantiated by a service using a Factory pattern. [14]

The Aspect pattern allows Aspect instances to attach delegates to one or more station instances at runtime - effectively a simple form of runtime weaving. The Aspect instances themselves maintain the state of the collection of delegates it instantiates. This provides the equivalent of a point-cut. Finally, by adding meta-data to the messages being passed through the MTS, Aspect behavior can be shared across a distributed system, again at runtime. Aspects can implement the Observer Pattern [14] to gather statistics about the interaction between stations. Aspects also can be used like CORBA interceptors [16] to change the parameters in inter-station method call. Multiple Aspects can be added, each concerned about a different facet of QoS adaptivity. Usually, these Aspects are independent, but they can communicate via Cougar services or by the meta-data added to messages.

3.5.1 Aspect Implementation in Cougar

Each Aspect is represented explicitly as a component with its own state and access to services offered by its peers. Since Aspects are implemented as components, they inherit all the benefits of the component-model. An Aspect has the ability to create delegates for given interfaces when asked to do so. Interface Factories, which create default instances of particular interfaces, will request delegates for a particular set of Aspects, chaining the delegates together in series. The resulting wrapped instance will have behav-

ior (in the form of a delegate) given by multiple Aspects. Likewise, an Aspect will contribute behavior to multiple interface instances, see figure below. The choice of which types of Aspects to use is made dynamically and, when necessary, sent to remote processes.

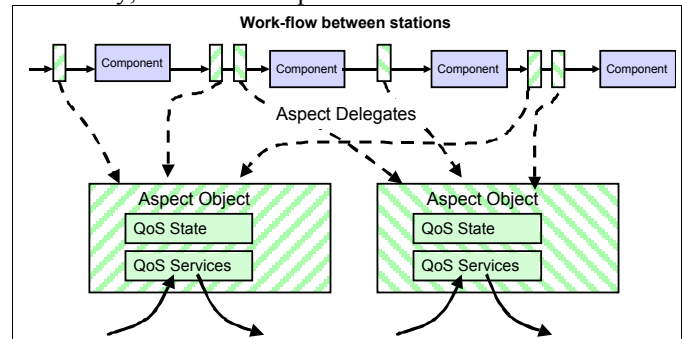


Figure 7. Aspects crosscut multiple service interfaces

This is a simple but powerful and dynamic form of AOP. The methods in the interfaces define collections of join-points while any specific Aspect instance implicitly defines a point-cut (depending on which interfaces it chooses to offer delegates for), as well as advice for each relevant join-point (the actual code in the delegate classes). The enclosing Aspect instance provides state as well as dynamic control over the delegation. When a call sequence crosses a host or virtual machine boundary the current list of relevant Aspects can be passed as meta-data to the remote process.

In common with other forms of AOP, the primary benefit provided by the Aspect Pattern is separation of concerns, the ability to encapsulate the implementations of distinct software concerns. This greatly improves the maintainability and extensibility of large software systems.

The Aspect Pattern offers three additional benefits vis-a-vis other approaches to AOP:

- *Dynamic code-weaving.* The implementations of the various concerns are selected and combined at runtime in the Aspect Pattern, not at compile or configuration time.
- *Dynamic meta-data.* The dynamic selection of concerns in the Aspect Pattern can be communicated between processes in a distributed system.
- *Ordinary programming languages.* The Aspect Pattern is written directly in ordinary program languages like Java and C++ and requires no third-party preprocessors or code generators.

On the other hand, the Aspect Pattern has two significant liabilities:

- *Delegation only.* Full AOP tools are more flexible and general than the Aspect Pattern, which is restricted to implementations that can be combined by delegation.
- *Modifications to existing code-base.* Full AOP tools do not require any modifications to an existing code-base. With the Aspect Pattern, factories must be modified to use the Aspect Attachment Service.

3.5.2 Examples

One example of the Aspect Pattern in Cougar occurs in a simple QoS adaptation case, the compression of messages on the fly if bandwidth drops. Compression is a two-part process: the sender will compress the data as it sends it, and the receiver will correspondingly decompress the data. The sender may choose any of a number of compression schemes. For the familiar separation-of-concerns reasons, we would like to keep the compression/decompression code in one place designed for that purpose, rather than spreading bits of it around. At the same time, the choice to add this behavior to standard message passing has to happen dynamically, not statically, a decision that must be shared among relevant components of a distributed system.

The Aspect pattern provides a clean solution to this set of problems. In the sender, an Aspect will insert a compression filter-stream delegate behind the raw RMI stream at runtime; the message passing system will notify the receiver via meta-data that this delegate is in place; the corresponding receiver Aspect will then use the meta-data to insert the right kind of decompression filter-stream delegate in front of its end of the RMI stream. The handling of compression is completely encapsulated in the Aspect, but unlike AspectJ, the “weaving” happens dynamically and can be shared across JVMs via meta-data.

4. Overhead of QoS Adaptation

QoS adaptations mitigate failures of the base-system to handle a varying environment. These adaptations cost system resources, such as CPU, memory, and bandwidth. It is important when designing methods for implementing QoS adaptation to be aware of system-resource cost. A relevant example of such cost-observation, as it applies to a previous method described above (Metrics Service and the component model), is consumption of network bandwidth vs. loaded components. The UltraLog program created a broad spectrum of QoS adaptations to categorically handle security, robustness, and adaptive logistics. Table 1 lists these differing configurations and their respective overhead in terms of code size. In reference, Metrics Service components comprise a node-level service for collecting distributed performance metrics. ‘Yellow Pages’ is Cougar’s *Yellow-Pages*-style service discovery subsystem. ‘Robustness’ stands for the Adaptive Robustness Defense Thread, which includes agent restarts, load balancing, and the DoS defense described in previous sections. ‘Security’ is the Adaptive Security Defense Thread handling access control, certificate management, and data protection in the system. ‘ALL’ includes all of these components. [2,3,13] Each of these subsystems illustrates some adaptivity, which can be enabled/disabled at configuration-time. As Table 1 reflects, the difference in code-size can be significant, which is interesting when trading off functionality for performance.

Adaptation	Number of Objects	Lines of Code
Metrics Service	36	3943
Yellow Pages	17	4570
Robustness	120	14856
Security	665	132505
ALL	838	155874

Table 1. QoS Adaptation Code Sizes

The benefit and relative overhead of an individual QoS adaptation is dependent upon the configuration of a Cougar society and the environment in which the society runs. For example, a QoS adaptation in one configuration could offer little benefit and have a high overhead relative to other components. This adaptation is a candidate for performance improvement, or even removal from the configuration. Conversely, in another configuration, the same QoS adaptation may have a high benefit, with only a small overhead relative to other components, making the service critical to society. Node-level QoS-adaptation services tend to have the following overhead behavior: high benefit in a certain situation, but not in others. Local vs. distributed configurations are classic examples of when QoS-adaptation service is needed, and when it becomes a performance liability, it can be removed. A major feature of Cougar is that QoS-adaptation can be removed for a local configuration and added back for distributed configurations. (Processing in a local situation does not usually need QoS-adaptation because the network environment is not changing.)

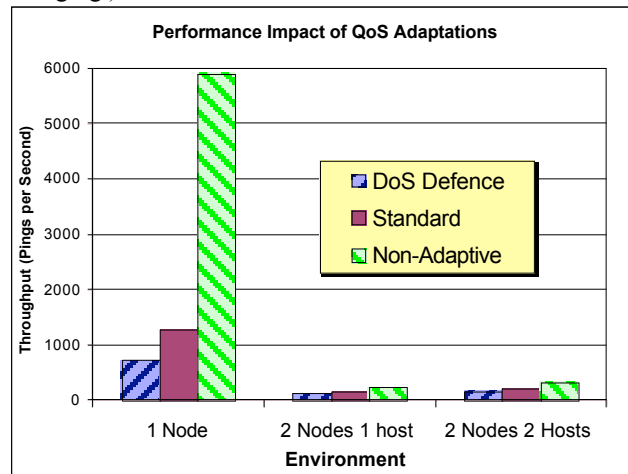


Figure 8. Adaptation overhead

Figure 8 illustrates the performance impact for different society configurations. The test application is a simple Ping society that publishes an object onto the source-agent’s blackboard, which is then relayed to the sink-agent’s blackboard. The performance of the society is measured by the number of objects that can be published per second, or pings per second. Figure 8 shows the performance of the Ping Society run with three configurations of QoS Adaptive

services over three environments. The ‘Standard’ configuration uses Cougar’s standard Node-level services, which have been implemented using the QoS adaptation mechanisms described in this paper. The ‘Non-Adaptive’ configuration replaces some of the Node-level services with alternative (trivial) implementations that cannot support QoS adaptation. These implementations are fast, but do not recover from common failures found in operational environments. Their purpose is to baseline Node-level services for their fastest possible performance. In this experiment, the Metrics, Threading, Servlet, Communities services, but not the Message Transport Service, were replaced with trivial implementations. The ‘DoS Defense’ extends the Standard configuration to add defenses against denial of service attacks discussed in this paper, such as DoS Attack Detection, DoS Attack Response, Compression, and SSL link protocol. For this experiment two 2.7 Ghz single processor Linux hosts were connected via 100Mbps Ethernet. As you will see, these configurations have different *relative* performance impacts and benefits, depending on the environment in which they are run.

In local configurations, the overhead for Standard QoS-adaptive services is relatively high. The ‘1 Node’ environment has both the source and sink agents on the same Node (Java VM), so messages between agents are not serialized. When the standard services are replaced with non-adaptive services the performance increases by a factor of five. The DoS defense, which has no benefit in this purely single host environment, reduces the performance even more. Cougar can be configured to remove the Standard and DoS device services, if they are not need, for example when debugging the functional behavior of a society or when deploying a society into an embedded environments, such as a hand-held computer.

In distributed configurations, message serialization dominates overall society performance. QoS-adaptation services become necessary to react to changes in the networked resources, such as device failure, competing network traffic, and security attacks. In addition to the benefit of such QoS services, their relative overhead is small. The ‘2 Node 1 Host’ environment shows a performance associated with serialization of messages and sending them over RMI. Notice, the standard configuration has almost a factor of 6 drop in performance when sending pings between node processes versus when the agents are on the same node. Despite this, the relative performance increase of replacing the standard services with non-adaptive services is only 40%, as opposed to 500% in the local environment. Since the benefit of these services is high and the relative overhead is low, this simple test suggests that these QoS-adaptive services should be installed in distributed configurations.

The ‘2 Node 2 Host’ environment of Figure 8 shows how the performance actually increases slightly when the nodes are run on two hosts connected by a high-speed network. The increase is due to the fact that some of the society processing can be done in parallel with the ping loop. In

this environment, the DoS Defense is necessary, but reduces the performance in half. The high cost of the DoS defense shows the need for dynamic adaptation. The DoS Defense can disable itself at runtime for environments where it is not needed, for example when the threat level is low or for communications between agents which are behind a firewall. So part of the society can be running the DoS defense, while other parts may not be running it.

The 2003 Ultralog assessment demonstrated that these techniques could scale to a moderate size logistics society, which ran on 69 dual-2.4GHz processor hosts, creating 131 nodes, and 765 agents. The security adaptation comprises roughly 19 nodes alone, totaling 63 agents.

5. Conclusion and Future Work

The Cougar component model has proven to be useful for creating both functional and QoS-adaptive components. In the UltraLog 2003 assessment over twenty different types QoS adaptive components were created and run on societies of seven hundred of agents. Note that these components are only the ones that are part of the Cougar middleware layer, and not those at the Cougar agent layer, which consists of 1000’s of Plugins instances. The components implemented QoS adaptation for security, dependability, performance tuning, and agent mobility. These components were designed to work in situations with chaotic resource bases and can be removed in other situations.

As the number of components increase and their implementations need to crosscut more places in the Cougar component hierarchy, the components begin to interact in unexpected ways. For example, two components may try to change the same control service, but for different and conflicting reasons. Detecting conflicts between components is currently an issue. Additional meta-data is necessary to assert the dependencies between components. More importantly, to understand how to compose multiple sets of QoS adaptive behavior, the QoS adaptive components must indicate explicitly the properties they supply and under what conditions they supply them.

6. Acknowledgments

The work described here was sponsored, in part, by the DARPA UltraLog contract #MDA972-01-C-0025. These ideas represent contributions by the many individuals who have participated in the DARPA ALP and UltraLog programs.

7. References

- [1] Cougar Home Page: <http://www.cougar.org>
- [2] Li B., Nahrstedt K. Qualprobes: Middleware QoS Profiling Services for Configuring Adaptive Applications, Proceedings of the IFIP/ACM Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware 2000), pp 256-272, Springer-Verlag, LNCS 1795.

- [3] D. C. Schmidt, D. L. Levine, and S. Mungee. The Design and Performance of Real-Time Object Request Brokers. *Computer Communications*, 21(4):294324, Apr. 1998.
- [4] Zinky JA, Bakken DE, Schantz RE. Architectural Support for Quality of Service for CORBA Objects. Theory and Practice of Object Systems, April 1997. <http://quo.bbn.com>
- [5] DeMichiel L, Yalcinalp L, Krishnan S. Enterprise Java Beans Specification Version 2.0, Sun Microsystems, August 2001.
- [6] Ashvin Goel, Molly H. Shor, Jonathan Walpole, David C. Steere, Calton Pu, "Using Feedback Control for a Network and CPU Resource Management Application", In Proceedings of the 2001 American Control Conference, Alexandria, Virginia, June 2001.
- [7] Bergmans L, Aksit M. "Composing Multiple Concerns Using Composition Filters," *Communications of the ACM*, special issue on AOP, October 2001.
- [8] Kiczales G, Hilsdale E, Hugunin J, Kersen M, Palm J, Griswold W. "An overview of AspectJ," Proceedings of the European Conference on Object-Oriented Programming (ECOOP), 2001.
- [9] Kiczales G, Hilsdale E, Hugunin J, Kersten M, Palm J and Grisold W, "Getting Started with AspectJ" *CACM* Oct 2001, page 59.
- [10] Richard Shapiro, John Zinky, Paul Rubel. The Aspect Pattern. OOPSLA 2002 Workshop - Patterns in Distributed Real-time and Embedded Systems, November 5, 2002, Seattle, Washington.
- [11] UltraLog Home Page: <http://www.ultralog.net>
- [12] Quo Home Page: <http://quo.bbn.com>
- [13] Aspect Oriented Programming Home Page: <http://aosd.net/>
- [14] Gamma, Helm, Johnson, Vlissides "Design Patterns: Elements of Reusable Object-Oriented Software" Addison-Westley, 1995 ISBN0-201-63361-2, pp 107-116
- [15] John Zinky, Joseph Loyall, and Richard Shapiro. "Runtime Performance Modeling and Measurement of Adaptive Distributed Object Applications" Proceeding of International Symposium on Distributed Object and Applications, DOA 2002, October 28-30 2002, University of California, Irvine CA USA
- [16] OMG CORBA Specifications "Common Object Request Broker Architecture (CORBA/IIOP)" Version 3.0.2 http://www.omg.org/technology/documents/formal/corba_iiop.htm