Meta-Programming Techniques for Distributed Real-time and Embedded Systems

Joseph K. Cross
Lockheed Martin Tactical Systems
P.O. Box 64525, M.S. U2X26
St. Paul, MN 55164-0525, USA
(651) 456-7316
joseph.k.cross@lmco.com

Douglas C. Schmidt
Electrical & Computer Engineering Dept.
University of California, Irvine
Irvine, CA 92697-2625, USA
schmidt@uci.edu

Abstract

Commercial off-the-shelf (COTS) middleware increasingly offers not only functional support for standard interfaces, but also the ability to optimize their resource consumption patterns. For example, a COTS real-time object request broker (ORB) may permit users to configure its server-side thread pooling policies. On one hand, this flexibility makes it possible to use standard functional interfaces in applications where they were not applicable previously. On the other hand, the non-standard nature of the optimization mechanisms – i.e., the "knobs and dials" – acts against the very product-independence that standardized COTS interfaces are intended to provide.

This paper provides two contributions to the study of mechanisms for reducing the life-cycle costs of distributed real-time and embedded (DRE) systems. First, we present a mechanism–called a Quality Connector–that enables applications to specify the qualities of service that they require from their infrastructure, and then manages the operations that optimize the middleware to implement those requirements. Second, we show how Quality Connectors are being applied in practice to allocate communication resources automatically for real-time CORBA event propagation. Although middleware that configures itself in response to quality of service (QoS) requests has been investigated and applied in general-purpose computing contexts, we believe that the present work is among the first to put such capabilities into mission-critical DRE systems with stringent QoS requirements.

1 Introduction

1.1 Emerging Trends

New and planned commercial and military distributed real-time and embedded (DRE) systems take input from many remote sensors, and provide geographically-dispersed operators with (1) the ability to interact with the collected information and (2) to control remote effectors. In circumstances where the presence of a human in the loop is either too expensive or too slow, these systems must respond autonomously and flexibly to unanticipated combinations of events at run-time. Moreover, these systems are increasingly networked to form long-lived "systems of systems" that must run unobtrusively and autonomously, shielding operators from unnecessary details, while simultaneously communicating and responding to mission-critical information at heretofore infeasible rates. In such environments, it is hard to enumerate, even approximately, all possible physical system configurations or workload mixes *a priori*.

It is possible *in theory* to develop these types of complex DRE systems from scratch. However, contemporary economic and organizational constraints, as well as increasingly complex requirements and competitive pressures, make it infeasible to do so *in practice*. The proportion of DRE systems made up of "commercial-off-the-shelf" (COTS) hardware and middleware has therefore increased dramatically, which helps reduce the initial non-recurring cost of these systems. In the context of this paper, middleware is software that functionally bridges the gap between application programs and the lower-level underlying operating systems and network protocol stacks. It therefore provides services whose qualities are critical to DRE systems [5].

1.2 Problems: COTS Refresh Costs and Lack of Standard Configuration/Control Interfaces

In many commercial application domains, such as e-commerce or consumer electronics, application software evolves faster than middleware. As a result, most mainstream COTS middleware products focus on presenting a powerful set of services that are attractive to new applications, so that existing applications can evolve freely. Long-lived DRE systems, however, often have the reverse problem, *i.e.,* how to write applications that can remain stable, while permitting and exploiting the relatively rapid evolution of the underlying infrastructure.

In the DRE domain, applications are often maintained over long periods, e.g., 20 to 30 years. When combined with free-market economics, this simple fact has far-reaching technical consequences. For example, consider the Theater Air Planner (TAP), which is the air tasking order generation function of the US Department of Defense (DoD) Theater Battle Management Core Systems (TBMCS). TAP is currently using version 7 of a popular COTS database product, which is the same version that was used when TAP was first written in 1995. Since then, there have been two major releases of this database product – version 8 in 1998 and recently version 9 – and these revisions provide functionality that would significantly enhance TAP.

Unfortunately, TAP cannot be upgraded to use these newer products cost-effectively due to a complex web of dependencies among its infrastructure components:

- The database
- The OS it runs on
- The implementation of the display widgets and
- The supporting Government-standard product set defined by the Defense Information Infrastructure Common Operating Environment (DII COE).

Notice that these dependencies were present *even though the system is architected with open standard interfaces and components*. When the consequences of all of these dependencies are taken into account, what might seem to be a simple version replacement actually requires a large-scale, prohibitively expensive effort.[1]

Moreover, if COTS components are available only through proprietary interfaces, DRE application developers system will be locked into using a particular set of COTS products. While proprietary COTS may still decrease initial system costs, it can increase maintenance and evolution costs. These costs can be non-trivial for long-lived systems since the typical cost to maintain a software product is from 60% to 80% of total life cycle costs [1]. Using COTS products via vendor-specific interfaces is therefore not generally in the long-term best interest of DRE system owners.[2]

Fortunately, the powerful new capabilities of COTS components are increasingly available to DRE applications through open standard interfaces, such as Real-time CORBA [11], Real-time Java, and Real-time POSIX. These standards enable system integrators to choose among various COTS implementations, which can reduce the on-going, recurring cost of these systems. Regrettably, however, the capabilities of COTS components to *optimize* their performance and resource consumption are *not* generally available through open standard interfaces. Instead, these capabilities are provided via *ad hoc* proprietary configuration and control parameters. This situation results in DRE systems that are once again locked in to using a single product, which significantly weakens the recurring cost advantage of COTS, often to the point where life-cycle system costs actually *increase* by using COTS.

[1] Not surprisingly, these types of problems are also found in long-lived commercial systems, such as complex telecom switches, as well as military systems.

[2] The use of proprietary interfaces in commercial products impose significant additional costs on product vendors compared with similar products with an open interface, since proprietary vendors are responsible for the expenses of requirements analysis, documentation, and training, which cannot be amortized by other companies or individuals participating in broader open standardization efforts.

1.3 Solution: Meta-Programming Techniques for DRE Middleware

Recent advances in fundamental software technologies, such as aspect-weaving software [12] and adaptive and reflective middleware, are beginning to address the problems outlined above. *Adaptive* middleware [6, 7, 20] is software whose functional and/or quality of service (QoS)-related properties can be modified either:

- *Statically*, *e.g.,* to reduce footprint or to use and configure resources that can optimized *a priori* in deeply embedded systems; or
- *Dynamically*, *e.g.,* in response to changes in environmental conditions or requirements, such as changing component interconnection topologies; component failure or degradation; changing power levels; changing CPU demands; changing network bandwidth and latencies; and changing priority, security, and dependability needs.

In DRE systems, adaptive middleware is required to make such modifications while still meeting stringent end-to-end QoS requirements.

Reflective middleware [21, 22, 23, 24, 25] permits programmatic examination of the capabilities it offers, and then permits programmatic adjustment of those capabilities. Reflective middleware supports a more advanced form of adaptive behavior, in that the necessary adaptations can be performed autonomously (or semi-autonomously) based on conditions within the system, in the system's environment, or in the doctrine defined by system operators and/or administrators. Such automatic adaptations must be implemented carefully to ensure that distributed optimizations retain system stability and converge rapidly.

This paper describes the goals, technical approach, and initial results of an ongoing research project called MINERS (Meta-INterface for Real-time Embedded Systems). MINERS is investigating the use of *meta-programming techniques* to provide DRE applications with an *open* interface through which they can configure and control the underlying middleware as they require. This goal is achieved in MINERS as follows:

- DRE applications are built to use open, standard COTS interfaces. In addition to the functional software that uses these interfaces, applications specify their required qualities of service (QoS), such as latency of event delivery and capacity of a wireless link. These QoS requirements are stated in a declarative form and cannot depend on middleware implementation details, *e.g.,* they cannot assume that inter-process communication is implemented by sockets.

- A new meta-interface mechanism, operating automatically during system development and at run-time, uses the configuration/control interfaces of the (necessarily adaptive and reflective) middleware to monitor and enforce the qualities of service specified by DRE applications.

In the MINERS project, we call this meta-interface mechanism a *Quality Connector*.

The rest of this paper is organized as follows: Section 2 describes the motivation for–and an overview of–Quality Connectors; Section 3 provides a detailed description of this concept and illustrates how we are applying it to DRE middleware and applications; Section 4 compares the MINERS project with related work; and Section 5 presents concluding remarks and a synopsis of the current and future directions of the MINERS project.

2 Applying Quality Connectors to Optimize DRE Middleware Declaratively

2.1 Background and Motivation

As commercial and military information technology users transition from a platform-centric to network-centric paradigm, an important challenge for researchers is to develop and evolve assurable, adaptable, and affordable standards-based DRE middleware that can be configured to implement required end-to-end QoS properties, such as predictable latency/jitter/throughput, scalability, dependability, and security. The *functional* interface to DRE middleware products can be–and increasingly is–standardized, yielding well recognized benefits. In addition, many middleware products that implement standard functional interfaces are adaptive and reflective in the sense described in Section 1, *i.e.*, they permit their qualities of service to be manipulated programmatically. However, the interface through which such reflection and adaptation is accomplished, namely, the *quality* interface, is *not* yet standardized. Consequently, any system that uses the quality interface–as DRE systems in general must–loses its infrastructure independence.

Before describing how we're addressing this problem on the MINERS project, we first describe the two levels at which an application can lose its infrastructure independence.

2.1.1 Primary Dependency of DRE Applications on Middleware

Primary dependency of DRE applications on middleware arises when applications are designed and written to use a single infrastructure product, as shown in Figure 1. Traditionally, such unique infrastructure products were created as part of the same effort that produced the applications. Two (historically valid) reasons have been used to justify the development of custom application infrastructure:

1. The system required qualities of service (*e.g.*, latency or reliability) that were not available from any existing functionally appropriate COTS infrastructure component; and
2. No existing functionally appropriate COTS infrastructure components would execute on the lower levels of infrastructure.

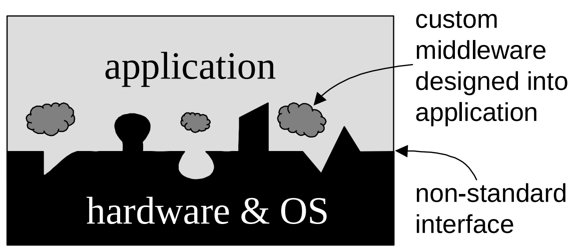


Figure 1. Historical Primary Dependencies

An example, taken from a production DRE system development effort, is as follows:

- A custom-built database was required because the operating system was custom-built and no existing database would run on it,
- Likewise, the operating system was custom-built because the hardware was custom-built and no existing operating system would run on it, and
- Likewise, the hardware was custom-built because, among other reasons, no existing hardware could provide the required I/O throughput.

Although the initial, non-recurring costs of such systems were high, the maintenance costs could be low, simply because little maintenance was required: if no enhancements to such a system were needed, then it could continue to run for many years, subject only to the availability of replacement hardware. Unfortunately, these systems were often *brittle*, in the sense that a small modification to the software, or a small modification to the function of the hardware, would require large-scale software changes. Moreover, these systems could not be evolved to leverage rapid improvements in COTS hardware and infrastructure software.

Today, the procurement costs of such systems—particularly mission-critical DRE systems—are often unacceptable due to budgetary constraints. Moreover, brittle end products are often unacceptable due to

1. The rapidly changing nature of mission-critical requirements and
2. The expanding universe of what is possible. In particular, if DRE systems can now support rapid response to an international humanitarian crisis, commercial aviation free-flight, and coordination of autonomous entities to clean up environmentally toxic situations, then those possibilities must not be foreclosed by the high cost of evolving software.

Fortunately, the primary dependence of applications on middleware can largely be avoided today by adopting open standard interfaces for DRE middleware, such as Real-time CORBA [11]. For example, Real-time CORBA implementations [19] can now be selected and configured such that their resource-consumption overheads are low enough and their qualities of service are high enough for all but the most demanding DRE applications.

2.1.2 Secondary Dependency of DRE Applications on Middleware

Secondary dependency of applications on middleware arises precisely from the process of optimizing the middleware by selecting implementation and configuration options for open standard DRE middleware, as illustrated in Figure 2.

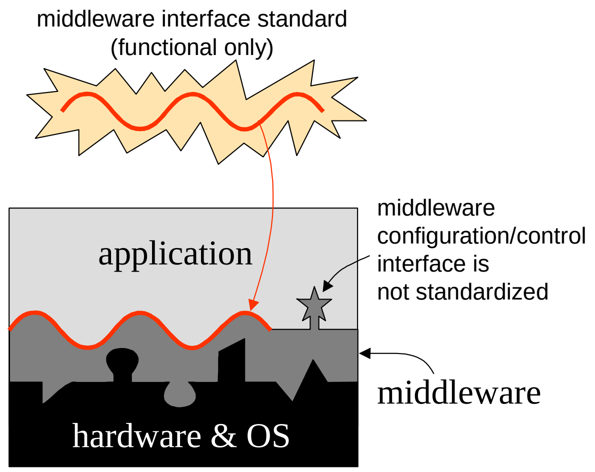

middleware interface standard
(functional only)
application
middleware configuration/control interface is not standardized
middleware
hardware & OS

Figure 2. Secondary Dependencies

In this paper, we call these user-selectable values the *properties* of middleware services. For example, consider a distributed application program that is designed to use the CORBA Event Service [2] for data distribution. This program has avoided the primary dependency problem, since there are many products available on the open market that implement the CORBA Event Service. However, these products differ in their properties, such as

- Transports and protocols supported
- Support for fault tolerance
- ORB initialization options
- Efficiency of marshalling and de-marshalling event parameters
- Efficiency of de-multiplexing incoming method calls
- Thread and thread priority utilization
- Buffer sizes, flow control, and buffer overflow handling

Note that these properties may be critical to the correct end-to-end behavior of the DRE system in which the middleware is embedded.

Moreover, for certain CORBA ORBs, some of these properties will be controllable by the application through idiosyncratic mechanisms, such as compilation options, link options, run-time environment variables, parameters passed to the ORB at initialization, and run-time interfaces for property value alteration. For example, consider the large-scale, HLA/RTI distributed interactive simulation environment described in [2]. In that work, numerous critical event-distribution optimizations are defined, and the mechanisms by which they were implemented are described. Examples of these optimizations include

1. Sophisticated event filtering to limit execution overhead and unnecessary data traffic
2. Selectable locking strategies for use when the implementation is iterating over a set of consumers that are to receive an event and
3. Selectable strategies for the choice of thread that is to dispatch an event to a consumer.

Although these optimizations may be critical to the performance of an end system, they are not controllable through *open standard* interfaces. Consequently, DRE applications that require specific qualities of services—even through open standard interfaces—must still be built to use specific products, thereby reducing the recurring cost savings from using COTS.

In general, the process of tuning middleware components to provide specified qualities of service is hard. The more flexibility that a middleware component or framework provides, the higher the level of skill required to configure its properties. The difficulty of obtaining the required QoS for applications in mission-critical DRE systems is compounded by the fact that the association of required qualities with services may change dynamically when the system mode changes, *i.e.*, when some set of events has caused a significant change in the operational characteristics of the system.

In DRE systems the time allotted to respond to mode changes may be very short. In fact, this requirement is one of the key technical differences between mission-critical DRE applications and mainstream commercial business applications. This issue is discussed further in Section 3.1.2, System Modes, below.

2.2 Overview of the Quality Connector

To address the primary and secondary dependency challenges described above, the MINERS project is developing a meta-programming mechanism called a *Quality Connector*. This mechanism allows applications to specify the qualities of service they require from their middleware. In this way, applications behave analogously to an executive who gives a package to his staff with direction that it must be delivered within a specified time. The Quality Connector acts, analogously to the staff, by selecting mechanisms for transport and setting the controllable parameters of those mechanisms. The position of the Quality Connector in a DRE system is illustrated in Figure 3.

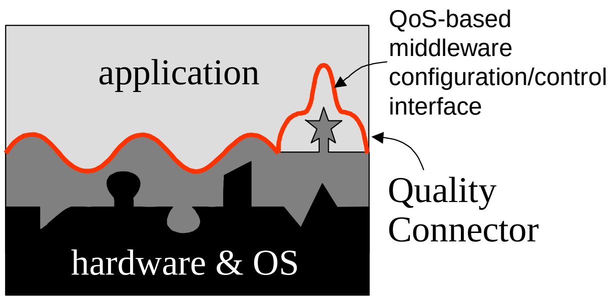


Figure 3. Role of the Quality Connector in DRE Systems

Figure 4 illustrates the means by which a Quality Connector fulfills its role. That figure shows application programs with associated specifications of their QoS requirements, the ingestion of those requirements by the Quality Connector, and the adjustment of middleware "knobs and dials" by the Quality Connector. Note that the applications do not directly manipulate—and thus do not depend on—the quality interface to the DRE middleware components.

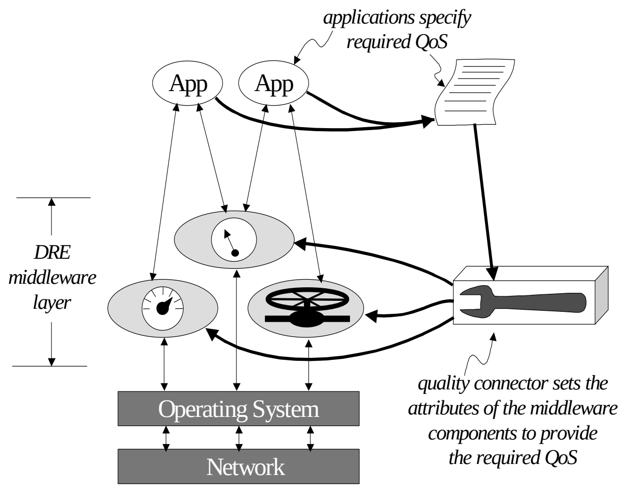

applications specify required QoS
App
App
DRE middleware layer
Operating System
Network
quality connector sets the attributes of the middleware components to provide the required QoS

Figure 4. The Operation of a Quality Connector in DRE Systems

Quality connectors are implemented in the MINERS project as follows.

1. First, we select a standard middleware service, such as the CORBA Event Service, where significant variability in QoS is possible across implementations of the standard interface. For this middleware service we define a set of qualities, such as the end-to-end latency of event delivery, that are important to the functioning of DRE applications.
2. We then define a small language in which acceptable values (or sets of acceptable values) of these qualities can be specified, and we permit the values specified to depend on the system mode. We define this language using XML so that it can be understood readily by humans and parsed easily by COTS tools.
3. Finally, we provide code-authoring-time, build-time, and run-time tools to check for feasibility and consistency of the requested quality values, and to set the properties of the middleware components to provide the required qualities.

The following section describes Quality Connectors in more depth.

3 <u>Detailed Description of the Quality Connector</u>

The implementation of Quality Connectors differs significantly depending on which middleware component and which of its interface functions is being addressed. To focus the discussion, we will pick one example and describe its specification and implementation, indicating how the specification and implementation might differ in other cases. The example middleware component we use is CORBA and the CORBA Event Service, the interface function we address is the event channel `push()`, and the property of that interface that we set through the Quality Connector is event delivery latency. It will be readily apparent that the techniques described here can be applied, *mutatis mutandis*, to many middleware services in addition to the CORBA Event Service.

3.1 Components of a Quality Connector

The CORBA Event Service Quality Connector, like most Quality Connectors, is implemented by several related but separate software components shown in Figure 5.

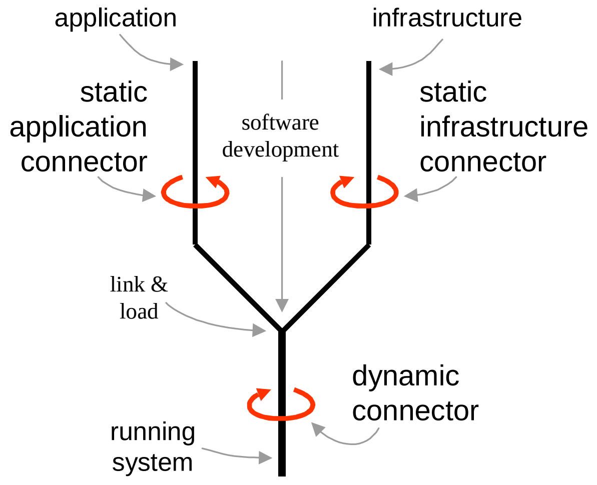


Figure 5. Components of a Quality Connector

These components behave as follows:

1. The *Static Application Connector* component acts on the application source code before it is compiled.

This component is akin to “aspect weaving”tools, such as AspectJ [12]. For example, this component can insert calls to run-time quality management functions following the creation of a consumer proxy.

2. The *Static Infrastructure Connector* component acts on the underlying middleware components before they are linked into the deployed system. For example, this component may set values for options before compiling the ORB itself, and it may select appropriate infrastructure components from ORB run-time libraries.
3. The *Dynamic Connector* component is linked in with the application and acts during its operation. For example, this component allocates resources, such as ATM circuits, processors, and radios, to each event flow.

In addition to the preceding essential components of the CORBA Event Service Quality Connector, there are several optional support components, such as

- Configuration tools that assist system engineers in selecting compatible sets of infrastructure components that implement required services and
- Simulation tools to determine whether locally specified qualities of service will combine to meet system-level requirements.

These support components will not be further addressed in this paper.

3.2 Background

This section provides a brief overview of event channels, which are a key component in the CORBA Event Service. We emphasize the choices and properties open to a DRE application implementation. In addition, the concept of *system mode* is described in more depth.

3.2.1 Synopsis of the CORBA Event Channel

CORBA event channels provide decoupled communications between suppliers and consumers of data, as shown in Figure 6. An event channel logically mediates the communication from each supplier to all consumers, *e.g.,* the actual communication can use multicast. In many implementations, however, event channels physically mediate these communications, *e.g.,* all events are routed through a separate process where an event channel resides. In either case, the communication between suppliers and consumers is “decoupled” in the sense that

1. It is asynchronous, *i.e.,* consumers will receive data after a supplier has completed its `push()` operation, and
2. The suppliers and consumers need not be aware of each other's identities.

There is no pre-defined limit on the number of suppliers and consumers that can be connected to an event channel at any time. Moreover, they can connect and disconnect at any time. There may be many event channels active at one time in a DRE system.

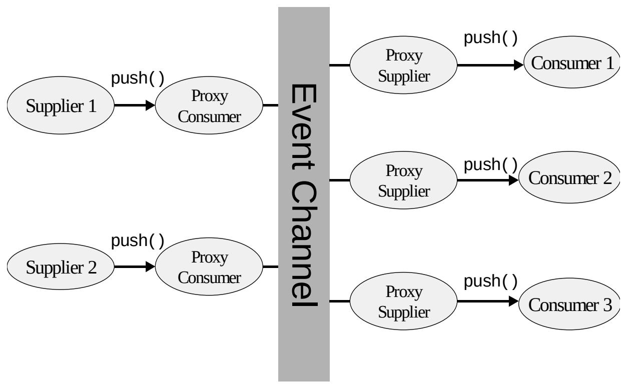


Figure 6. A Simple CORBA Event Channel

In this paper, we emphasize the “push” model of event delivery, where a supplier invokes a `push(data)` method to supply any type of data, and the event channel causes `push(data)` methods to be invoked on all consumers registered with that event channel. In addition to the “push” model, there is also a “pull” mode of event delivery, which we do not address further in this paper.

The CORBA specification leaves many aspects of event channel behavior unspecified intentionally. For example, the following properties of event delivery are not specified:

1. Latency of event delivery
2. Where and how often event data are copied
3. Threading and synchronization policies for event dispatching
4. What communication mechanism is used to convey the event data from the supplier to the consumers; *e.g.,* which of several radio channels will be used
5. How and where event data are buffered, and how big event data buffers are
6. What happens when an event data buffer overflows
7. Reliability of event delivery
8. Whether events from one supplier will be delivered to each consumer in the order in which they were supplied
9. If supplier `Alpha` supplies an event `E1` to an event channel, and only after consuming `E1` does `Beta`, who is both a supplier and consumer, supply an event `E2` to the same event channel, and if consumer `Omega` consumes both events, must `Omega` receive `E1` before `E2`?
10. If a consumer connects to an event channel, and if an event is supplied to that channel one minute later, will that consumer receive that event? Does the answer depend on whether the supplier and consumer are on different continents?

Different implementations of the CORBA Event Service [2] provide different APIs for controlling the various properties outlined above. Applications that do not address these variations are therefore prone to the secondary dependencies described in Section 2.1.2.

3.2.2 Mission-Critical System Modes

Mission-critical systems are often characterized as a hierarchy of parts, which we call *configuration items*. A configuration item may be *small* (such as a board in a computer) or *large* (such as a ship). A configuration item may exist *statically* (as does a router) or may be created and destroyed *dynamically* (as is a thread within a process). Configuration items may contain other configuration items; this containment relation forms a directed acyclic graph.[3]

We assume that every configuration item is always in one of a fixed, finite set of states. For example, a workstation may be in a training state or an operational state, and a radar may be in a search state, tracking state, self-test state, or off-line state. The state of a configuration item may (but need not) be a function of the states of its contained configuration items.

Now we can define a system *mode* as a Boolean function on the states of its constituent configuration items. For example, “the ship is in battle state” is a mode, and “all ATM backbone configuration items are in their operational states” is a mode. The value of a mode can change abruptly. For example, the failure of a component can affect modes.

The qualities of distributed communication services that applications require will differ in different modes. For example, a crew entertainment video that is distributed over a shipboard backbone network requires a low jitter, and therefore constitutes a high priority[4] flow of information. However, when the platform enters battle mode due to the detection of an incoming anti-ship cruise missile, then the priority of the crew entertainment video must drop rapidly. Similarly, the importance of processes within a nuclear reactor control system might be expected to change when the reactor enters the “over-temperature” mode. The mode-change problem is addressed by permitting applications to specify QoS as a function of mode. The result is that resource allocations can be made in advance of their need.

[3] The containment relation on configuration items need not form a tree or set of trees since some configuration items may be part of several others; e.g., a LAN may be part of the combat system configuration item and part of the command and control system configuration item.

[4] This notion of priority will be refined in Section 3.3 below, where we call this property of the video flow its *urgency*.

A related problem arises when a mode changes but QoS requirements do not change. When the failure of a resource, such as a LAN, occurs and requirements which that LAN had been supporting remain in effect, then new resources must be identified and configured into operation as quickly as possible. This operation is often called “fault reconfiguration.”

3.3 Quality Connector QoS Specification

The problem of how to specify the required QoS is surprisingly subtle, even when the quality in question is simple, such as latency. “Worst case” bounds are the obvious choice. But if “worst case” is interpreted literally–for example, that the latency of message delivery will never, under any circumstances, exceed a specified time limit–then such bounds are clearly not feasible in practice because no infrastructure is infinitely reliable. Moreover, many infrastructure components have no fixed response time, but rather a distribution of response times, which often has a long tail, as shown in Figure 7.

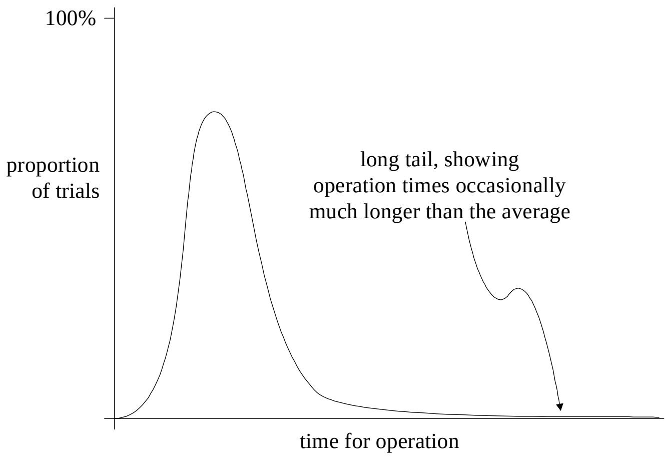


Figure 7. Problem with "Worst Case" Analysis

This distribution applies to operations such as task dispatching (due to such influences as priority inversions and interrupt lockout by device drivers), and even to straight-line code execution (due to cache state). If resources were allocated to support the actual worst case, then resource utilization would inevitably be very low, which can render a DRE system ineffective in practice.

If there is a primary factor that distinguishes the QoS requirements of DRE systems from those of “best-effort” commercial systems, it is this: *best-effort commercial systems are concerned primarily with average values of qualities of service, while DRE systems are concerned with extreme values of their distributions.*

Instead of worst case bounds, therefore, we assume that latencies will be constrained by a conjunction of one or more conditions of the form “<proportion> of latencies shall be less than or equal to <time-interval>.” For example, a QoS specification for latency might be “99%

of latencies less than or equal to 1.0 seconds and 99.99% of latencies less than or equal to 4.0 seconds.”

It should be noted that the preceding is a special case of a much more general and powerful technique, which we call *probability assertions*. A probability assertion about a distribution of values is the assertion that the cumulative density function of the distribution in question lies entirely between an upper bounding function and a lower bounding function. In the preceding example, only an upper bounding function is specified, and that is a step function. Since the generality of probability assertions is not essential to the present discussion, this subject will not be treated in detail here.

In addition to specifying the QoS required from the middleware service, the application must also bound the load that it will impose on the service. Such a bound is necessary not only to assess whether the service can handle the load at the specified QoS, but also to determine whether future service requests can share resources with the current service request. For a CORBA Event Service, the load consists of the distribution of event sizes in bytes and the distribution of inter-service-request times.

A convenient choice for specifying the distribution of inter-event times is to use token buckets [3], which are a means to express constraints on sequences of events in time. A token bucket constraint is specified in terms of a container of fixed size – the bucket – into which tokens periodically fall. No event can take place until an appropriate number of tokens are present in the bucket.

Although token bucket specifications are simple, portable, and define realistic traffic distributions, they are not sufficiently expressive for our purposes. For example, one cannot use token buckets to specify either a strictly periodic load nor a purely random, Poisson-distributed, load, both of which are important in the DRE domain. Fortunately, probability assertions are adequate, *e.g.*, a probability assertion on the inter-event time distribution can specify that events will be no more frequent than Poisson-distributed with a mean of 4 events per second, and no two consecutive events will be closer together than 0.01 seconds. Again, this generality is not essential to the present subject, and only simple special cases will be used here. For a reason that will be clear shortly, the combination of a mode, a QoS specification, and a load specification is called a “proposal.” Figure 8 presents the proposal alluded to above, expressed in XML.

The proposal in Figure 8 applies only when either of a pair of tactical military or emergency response team radios is on-line. In that case, the time between a supplier's `push()` call and all consumers' corresponding `push()` calls for every event are to be less than 1.0 second 99% of the time and less than 4 seconds 99.99% of the time. The sizes of the event data are always at most 256 bytes, and 50% of the time are less than or equal to 32 bytes. The supplier's `push()` calls occur periodically, once per second. Note that the priority of the request consists of two integral values: urgency and importance:

- *The urgency* of a request determines which of several eligible requests will get access to a shared resource. For example, if either of two packets of data could be sent over a communication link, the packet with the higher urgency will be sent.

- The *importance* of a request determines which of two requests (both of which cannot be supported) will be accepted. For example, if both of two requests for event data propagation cannot be supported on the present infrastructure, then the request with the higher importance will be accepted and the other will be rejected. Moreover, if a new request for service is received, and that request can be accommodated only if some currently operating, lower importance service is shut down, then that will be done; in this case, we say that the lower importance request are *abrogated*.

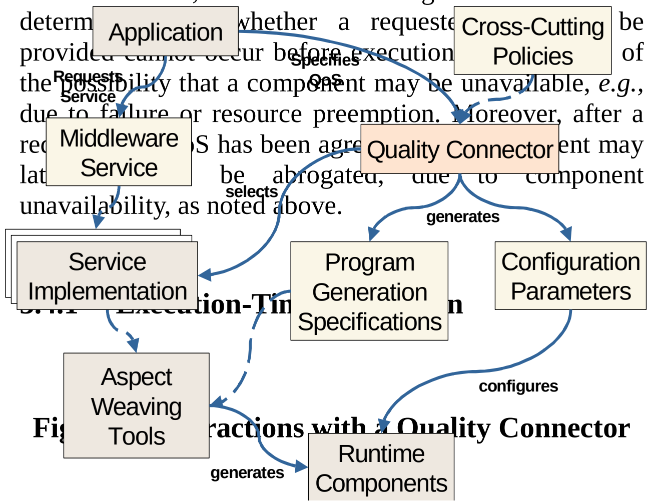
```
<proposal>
   <mode>                       Proposal applies in this
                                mode
      <or>
         <ci name="radioVHF" state="onLine"/>
         <ci name="radioUHF" state="onLine"/>
      </or>
   </mode>                      There are QoS types other
   <QoS type="latency">         than latency -- e.g., jitter
      <upperPoint secs="1.0" prob="0.99"/>
      <upperPoint secs="4.0" prob="0.9999"/>
   </QoS>                               Flow is periodic
   <load type="interMessageTime">
      <upperPoint secs="1.0" prob="0.0001"/>
      <lowerPoint secs="1.0" prob="0.9999"/>
   </load>
   <load type="messageSize">
      <upperPoint bytes="256" prob="1.0"/>
      <upperPoint bytes="32"  prob="0.5"/>
   </load> <load type="priority">
      <urgency val="10"/>       Priority determines how
      <importance val="2"/>     this request will compete
   </load>                      with others for resources

</proposal>
```

Figure 9. A Proposal in XML

3.4 Interaction

Interactions with a Quality Connector can occur at several points in time. The specification of the required QoS may occur as early as system design time or as late as system execution time, as shown in Figure 9. The final determ[illegible] whether a requeste[illegible] be provid[illegible]cur before execution [illegible] of the possibility that a component may be unavailable, *e.g.*, due to failure or resource preemption. Moreover, after a re[illegible]S has been agre[illegible]ent may lat[illegible] be abrogated, due to component unavailability, as noted above.



[illegible]Execution-Ti[illegible]n

Fi[illegible]ractions with a Quality Connector

At execution time, DRE application software generally submits its proposal to a Quality Connector before making any use of the associated service, such as the CORBA Event Service. If the Quality Connector determines that the requested QoS can be provided, then the Quality Connector sets whatever control parameters are necessary to configure the middleware service, and returns an indication of acceptance to the application. Thereafter, the application can use the service and receive the QoS it requires. If the Quality Connector determines that the requested QoS cannot be provided, it returns an indication of rejection to the application.

Although it would be possible to apply a QoS property, such as end-to-end latency, to an event channel as a whole, this would not be acceptable in applications with a wide variation in propagation distances. For example, consider a coordinated group of autonomous vehicles that supply "Here I am" messages to an event channel. It might generally be desirable that these messages be received quickly by other vehicles in the group. A much larger latency might be permitted, however, to messages received at a central monitoring function, located several satellite hops away. As a result, we permit different QoS values to be specified at each supplier and consumer proxy. In the autonomous vehicles example, for instance, suppliers would specify no latency, consumers on the vehicles would specify a short latency, and the consumer at the central monitoring function would specify a longer latency.

Note that there is no conflict possible between different QoS values set at suppliers and consumers. If a supplier requests a latency of 1 second, and a consumer on the same event channel requests a latency of 2 seconds, and if the Quality Connector accepts both proposals, then both proposals must be honored. In this case, the latency from the supplier to consumer must not exceed 1 second.

3.4.2 Program-Generation-Time Interaction

For the preceding run-time interaction between the application and a Quality Connector to occur, application software must be modified somehow. In particular, the QoS proposal must be delivered to the Quality Connector run-time component at some point between the creation of an event channel proxy and the supplying or consuming of an event through that proxy. In our implementation, aspect-oriented programming (AOP) techniques [4, 26] are the means by which system designers specify the following information to a Quality Connector:

1. The identity of the `push()` call to which a proposal applies and
2. The modifications that the Quality Connector must apply to the generated application.

In our work thus far, the following AOP features of AspectJ [12] appear to be adequate for the applications envisioned in MINERS:

- *Join points* are the mechanism used in AspectJ to select locations in the program source code at which to apply designated modifications to the code.
- *After advice* is one style of such modification, which takes the form of adding code following the selected locations.

3.5 *Implementation*

Implementing end-to-end QoS guarantees for a service requires that all components that participate in providing the service be represented by software *avatars*. An avatar is a software object that represents a component to the Quality Connector function of the middleware, as shown in Figure 10.

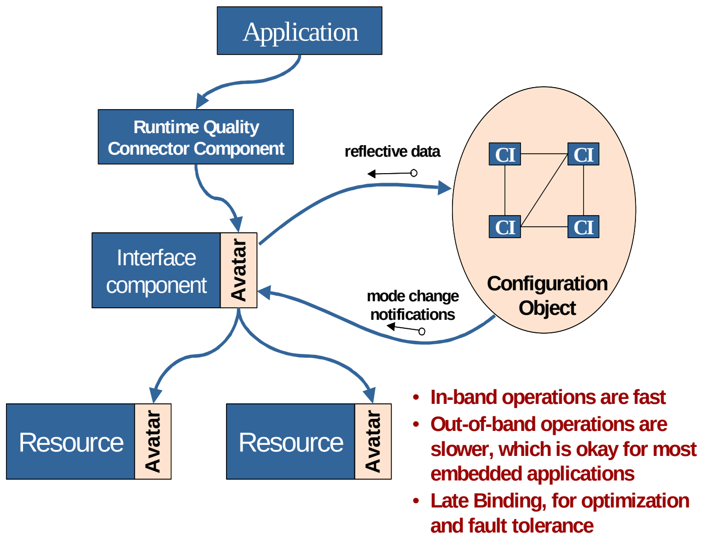

Application
Runtime Quality Connector Component
reflective data
CI
CI
CI
CI
Interface component
Avatar
Configuration Object
mode change notifications
Resource
Avatar
Resource
Avatar
• In-band operations are fast
• Out-of-band operations are slower, which is okay for most embedded applications
• Late Binding, for optimization and fault tolerance

Figure 10. Avatars Represent Components

Hence an avatar must be able to report what QoS the component is presently providing, and what QoS the component *would* provide if a specified additional load *were* added. In addition, an avatar provides the interface through which the Quality Connector function configures the properties of the component.

The use of avatars to support run-time negotiations for QoS-constrained DRE application services was demonstrated by one of us (Cross) using the Real-time CORBA ORB [19] implemented by the other (Schmidt). We provided an interface through which suppliers and consumers of events to a real-time event channel could request specific qualities, such as latency, of the push event delivery service. That request was forwarded to the event channel object itself, which negotiated with the available avatars to obtain the requested QoS. The result – success or failure – of those negotiations was then returned to the application-level requestors.

Note that the greater the configurability that the components provide, the greater the opportunity for middleware to support changing conditions, where such changes may take place over periods of milliseconds, days, or even years. Conversely, the absence of configuration control capabilities and information about service components, such as an ORB that cannot be told how big to make its buffers or how big its default buffer sizes will be, has the result that no QoS can be guaranteed for services that use the component.

In the present example, we are concerned with the configurability of a CORBA Event Service. Such configuration may involve ORB parameters that are

1. Set on ORB initiation
2. Changed during execution and
3. Included in interface calls, such as on event channel creation and on a `push()` invocation.

Clearly, more detailed discussion of these configuration issues depends entirely on the ORB implementation that is used.

4 Related Work

Our work on Quality Connectors complements the work being done by the DARPA Quorum program [5]. Quorum's goal is to develop technologies to allow tactical applications with mission-critical performance requirements to dynamically access distributed COTS resources with guaranteed quality of service. Applications negotiate service contracts with the system, which are then enforced through layered resource management mechanisms and maintained through continual monitoring, adaptation, and feedback control.

The BBN Quality Objects (QuO) framework [6] is a Quorum project that uses QoS definition languages [7] that are based on the separation of concerns promoted by AOP [4] (see below). A significant difference between the MINERS approach and that of QuO is that MINERS relies on an explicit specification of resource requirements against system state to provide immediate access to those resources when a change occurs, rather than relying on the gradual adaptation of resource allocations to changing demand.

The Distributed Multimedia Research Group at Lancaster University has proposed and implemented a prototype of next-generation reflective middleware [21, 22] called *Adapt*. Their middleware model concentrates on dynamic composition of objects through *open-binding*, which (1) allows object implementations to be configured dynamically and (2) determines various aspects of object implementations, such as adding or removing methods from an object. The Adapt project model also facilitates QoS properties management and monitoring. Compared to the Adapt project, MINERS concentrates on identifying and using meta-programming techniques to implement and improve the implementation of an existing middleware standard (CORBA), whereas the Adapt project defines and implements the meta-space of a new middleware framework at a higher level.

The Real-time (RT) CORBA 1.0 specification [11] extends the Object Management Group (OMG) CORBA standard to support real-time distributed, object-oriented applications. The initial 1.0 version of the RT CORBA specification focuses on fixed-priority applications to ensure end-to-end predictable behavior for information that flows between distributed objects. It does this by giving developers explicit control over allocation and use of the following resources:

- Processor resources are configured and controlled using thread pools, priority control and synchronization mechanisms.
- Communication resources are managed through the ability to specify protocol properties and by making explicit bindings to communication resources.
- Memory resources are managed through buffering requests and limiting thread pool sizes.
- A global scheduling service is also available [20].

In addition to RT CORBA, the CORBA Notification Service incorporates important QoS and filtering features into the previously defined CORBA Event Service. These middleware capabilities, appearing in an open specification that is independent of platform, OS, and vendor-specific communication mechanisms, offer a solid foundation for an open implementation of meta-programming interfaces.

The dynamic TAO [23] and Reflective CCM [24] projects have demonstrated that CORBA can be reconfigured at run-time by dynamically linking and unlinking certain components. Similarly, AspectIX [25] is a novel CORBA-compliant middleware architecture that defines and describes QoS requirements on a per-object basis independently from functional interfaces. Clients in AspectIX systems are allowed to *set* the QoS *aspects* of objects. Systems may adapt, report aspect changes back to clients, or *reflect* to clients on how to adapt. The MINERS work, however, also focuses on QoS adaptation as a deployable entity in the system to standardize and automate the *server*-side QoS control/adaptation issues.

Our approach to specifying QoS at the application level in a form that is relatively independent of the functional behavior of the application is facilitated by the emerging research in Aspect Oriented Programming (AOP) [26]. Work in this area is underway in various places, including Xerox PARC [4], IBM [13], MCC [14, 15], Northeastern University [16], and the University of Twente [17]. We have chosen to use AspectJ [12], which is an aspect-oriented extension to the Java programming language. AspectJ addresses the problem of crosscutting concerns by extending Java with constructs that can be used to implement such concerns in a modular way. AspectJ is in the late beta stages of development, yet promises to provide more generalized aspects than much of the related work being done in this area.

A related area of research is Generative Programming [18], which is an approach to constructing systems that involves modeling an entire family of systems. Given requirements for a particular member of that family, this approach generates that member as a composition of elementary components. Both AOP and Generative Programming are being explored in the context of the DARPA PCES program [27]. The IETF has specified mechanisms for scalable differentiated [8, 9] and integrated [10, 3] classes of service on the Internet.

A number of enabling technologies are emerging that will make it possible to implement meta-interface mechanisms more easily in the future. Available at different levels, including the middleware itself, these technologies provide various forms of support for QoS.

- Differentiated services (DiffServ) provide QoS using a small, well-defined set of building blocks from which a variety of aggregate behaviors may be built. Service characteristics may be specified in quantitative or statistical terms of throughput, delay, jitter, and/or loss, or they may be specified in terms of priority of access to network resources. A small bit-pattern in each packet is used to mark the packet to receive a particular forwarding treatment, or per-hop behavior, at each network node along its path. The DiffServ specifications provide a common understanding of the use and interpretation of this bit-pattern. Sophisticated classification, marking, policing, and shaping operations can now be implemented at network boundaries or hosts. Network resources are allocated to traffic streams by service provisioning policies which govern how traffic is marked and conditioned upon entry to a differentiated services-capable network, and how that traffic is forwarded within that network.
- Integrated services (IntServ) provides the ability to transport audio, video, real-time, and data traffic within a single packet switched network infrastructure. IntServ defines a minimal set of global requirements and services which transition the Internet into an integrated-service communications infrastructure. It includes interfaces to specify an application's end-to-end QoS requirements.

5 Concluding Remarks and Future Directions

COTS middleware has become more capable and the proportion of mission-critical system requirements that *cannot* be met using COTS middleware is shrinking dramatically. This trend applies even to mission-critical DRE systems, such as ship-board combat systems and commercial avionics mission computing systems, that are subject to stringent reliability and quality of service (QoS) requirements. The result is a reduction in the initial, non-recurring cost of these systems.

COTS middleware is playing an increasingly important role in developing mission-critical DRE systems due to

- Economic and organizational constraints, such as severely constrained procurement budgets, and the movement toward prime-vendor support contracts that allocate the uncertainty in system maintenance costs to the developing contractor;
- Increasingly complex system requirements, such as Global Air Traffic Management (GATM) requirements for military aircraft that fly in commercial airspace; and
- Competitive pressures, such as enticements for scientists and engineers from many sectors of the global economy.

Thus, the potential affordability gains offered by COTS middleware have become strategically important. Without a product- and component-independent mechanism for optimally configuring COTS middleware, however, this affordability gain is threatened.

Our experience developing previous generations of complex DRE systems [2, 5, 19, 20] illustrates that effective operation, interoperability, and integration requires more than individual COTS standards and tools. Instead, it requires that adaptability, assurability, and affordability be designed into DRE system/network architectures *a priori*. Researchers have a pressing need, therefore, to coordinate individual advances in the COTS solution space that are being addressed by different sectors of the R&D community.

The problems faced by researchers and developers of DRE systems are highly challenging, with many interlocking aspects. Unless pieces of the emerging, independently developed, COTS solutions can be delivered to application designers as coordinated, integrated packages, their value will be diminished and may in fact make matters worse instead of better, *e.g.*, due to excessive costs for COTS refresh and integration. This paper proposes a meta-programming mechanism called *Quality Connectors* that allow a variety of separately developed, and continuously evolving, tools and components to appear to application designers as an integrated, coordinated, and stable infrastructure. A Quality Connector provides this appearance by encapsulating the various configuration and control mechanisms provided by COTS middleware, and exposing a stable, QoS-based interface to applications.

Implementation of the capabilities described in this paper is underway in the MINERS project at Lockheed Martin Tactical Systems, in Eagan, Minnesota, as part of the DARPA PCES Program [27]. We are using the TAO RT CORBA ORB [19], which is a highly configurable middleware component designed to support DRE applications with demanding QoS requirements. In the longer term, if the mission-critical, real-time embedded system community can achieve a shared understanding of what qualities of services need to be specified and how to specify them, then we envision the availability of DRE middleware that is designed to be configured to meet such requirements, and the development of applications that include their QoS requirements as part of their design. Such applications should be be far more stable over evolving infrastructure than current applications. Moreover, such applications might be verifiable independently of any infrastructure, based on their QoS requirements, which will substantially reduce costs in mission-critical DRE applications.

Bibliography

[1] Guidelines for Successful Acquisition and Management of Software Intensive Systems: Volume 1 -- Version 3.0, May 2000, Department of the Air Force, Software Technology Support Center. http://web2.deskbook.osd.mil/reflib/DAF/035GZ/013/035GZ013DOC.HTM#T2

[2] Carlos O'Ryan, Douglas C. Schmidt, and David Levine, “Applying a Scalable CORBA Events Service to Large-scale Distributed Interactive Simulations,” Proceedings of the IEEE 5th Workshop on Object-oriented Real-time Dependable Systems, Montery, CA, November, 1999.

[3] General Characterization Parameters for Integrated Service Network Elements, TOKEN_BUCKET_TSPEC. IETF Integrated Services, RFC 2215 (section 3.6.) http://www.ietf.org/rfc/rfc2215.txt?number=2215.

[4] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin, “Aspect Oriented Programming.” Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland, June 1997. http://www.parc.xerox.com/spl/groups/eca/pubs/papers/Kiczales-ECOOP97/for-web.pdf, and also http://www.parc.xerox.com/csl/projects/aop/.

[5] The Quorum Project, DARPA Information Technology Office. http://www.darpa.mil/ito/research/quorum/index.html

[6] Quality Objects website, BBN Technologies. http://www.dist-systems.bbn.com/tech/QuO/

[7] Pal PP, Loyall JP, Schantz RE, Zinky JA, Shapiro R, Megquier J. “Using QDL to Specify QoS Aware Distributed (QuO) Application Configuration. Proceedings of ISORC 2000,” The Third IEEE International Symposium on Object-Oriented Real-time Distributed Computing, March 15-17, 2000, Newport Beach, CA.

[8] Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers. IETF Differentiated Services. http://www.ietf.org/rfc/rfc2474.txt?number=2474

[9] An Architecture for Differentiated Services. IETF Differentiated Services. http://www.ietf.org/rfc/rfc2475.txt?number=2475

[10] Specification of Guaranteed Quality of Service, IETF Integrated Services, RFC 2212. http://www2.ietf.org/rfc/rfc2212.txt

[11] Real-time CORBA (Chapter 24). Common Object Request Broker Architecture v2.4.2. http://www.omg.org/cgi-bin/doc?formal/01-02-60

[12] The AspectJ website at http://aspectj.org.

[13] P. Tarr, H. Ossher, W. Harrison and S.M. Sutton, Jr. "N Degrees of Separation: Multi-Dimensional Separation of Concerns." Proceedings of the International Conference on Software Engineering (ICSE'99), May, 1999. http://www.acm.org/pubs/articles/proceedings/soft/302405/p107-tarr/p107-tarr.pdf

[14] Robert E. Filman, Stuart Barrett, Diana D. Lee, Ted Linden, “Inserting Ilities by Conrolling Communications,” Communications of the ACM, in press. http://ic-www.arc.nasa.gov/ic/darwin/oif/leo/filman/text/oif/oif-cacm-final.pdf.

[15] Robert E. Filman, “Applying Aspect-Oriented Programming to Intelligent Synthesis,” Research Institute for Advanced Computer Science, NASA Ames Research Center. June 2000.

[16] The Demeter Project homepage at http://www.ccs.neu.edu/research/demeter/

[17] TRESE Aspects and advanced separation of concerns homepage http://trese.cs.utwente.nl/aspects_asoc/index.htm

[18] K. Czarnecki and U. Eisenecker, "Generative Programming : Methods, Tools, and Applications." Addison-Wesley, June 2000.

[19] Douglas C. Schmidt, David Levine, and Sumedh Mungee "The Design and Performance of Real-Time Object Request Brokers," Computer Communications, Elsivier, Vol. 21, No. 4, April 1998.

[20] Chris Gill, David Levine, and Douglas C. Schmidt, “The Design and Performance of a Real-time CORBA Scheduling Service,” Real-time Systems, Kluwer, Vol. 20, No. 2, March, 2001.

[21] Gordon S. Blair, G. Coulson, P. Robin, and M. Papathomas, "An architecture for next generation middleware," in Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing, Springer-Verlag, London, 1998.

[22] Fabio M. Costa and Gordon S. Blair, "A Reflective Architecture for Middleware: Design and Implementation," in ECOOP'99 PhDOOS Workshop, Lisbon, Portugal, 1999.

[23] F. Kon and R. H. Campbell, "Supporting Automatic Configuration of Component-Based Distributed Systems," in Proceedings of the 5th Conference on Object-Oriented Technologies and Systems, (San Diego, CA), USENIX, May 1999.

[24] Nanbor Wang, Douglas C. Schmidt, Kirthika Parameswaran, and Michael Kircher, “Towards a Reflective Middleware Framework for QoS-enabled CORBA Component Model Applications,” IEEE Distributed Systems Online special issue on Reflective Middleware, 2001.

[25] F. J. Hauck, U. Becker, M. Geier, E. Meier, U. Rastofer, and M. Steckermeier, “The AspectIX Approach to Quality-of-Service Integration into CORBA,” Technical Report TR-I4-99-09, Operating Systems Dept., Friedrich-Alexander University, Erlangen-Nürnberg, Germany, 1999.

[26] G. Kiczales, "Aspect-Oriented Programming," in Proceedings of the 11th European Conference on Object-oriented Programming, June, 1997.

[27] The Programmable Composition of Embedded Software (PCES) Project, DARPA Information Technology Office. http://www.dsic-web.net/ito/programs/pces/.