Applying the Quality Connector Pattern to Optimize Distributed Real-time and Embedded Middleware and Applications

Joseph K. Cross
joseph.k.cross@lmco.com
Lockheed Martin Tactical Systems
St. Paul, Minnesota

Douglas C. Schmidt
schmidt@uci.edu
Electrical & Computer Engineering
University of California, Irvine

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 Real-time CORBA object request broker (ORB) permits application developers to configure server thread pooling policies. This flexibility makes it possible to use standard functional interfaces in applications where they were not applicable previously. However, the non-standard nature of the optimization mechanisms—*i.e.*, the "knobs and dials"—against the very product-independence that standardized COTS interfaces are intended to provide.*

This chapter provides three contributions to the study of patterns and mechanisms for reducing the life- cycle costs and improving the quality of service (QoS) of distributed real-time and embedded (DRE) systems. First, we describe key sources of dependencies that reduce the flexibility and increase total ownership costs of DRE software. Second, we present an architectural pattern called Quality Connector, which is a metaprogramming technique that enables applications to specify the QoS they require from their infrastructure, and then manages the operations that optimize the middleware to implement those QoS requirements. Third, we describe how Quality Connectors are being implemented in practice to allocate communication resources automatically for Real-time CORBA event propagation. Although middleware that configures itself in response to QoS requests has been investigated and applied in general-purpose computing contexts, 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 for DRE Systems

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 humans in the loop is too expensive or their responses are too slow, these systems must respond autonomously and flexibly to unanticipated combinations of events at runtime. Moreover, these DRE systems are increasingly being 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 chapter, middleware is software that functionally bridges the gap between application programs and the lower-level underlying operating systems and network protocol stacks [1].

The qualities of the services that middleware provides are critical to DRE systems. Moreover, the required qualities of a given service can vary over time. For example, consider a crew entertainment video that is distributed over a shipboard backbone network. This video distribution requires a low jitter, and therefore constitutes a high priority flow of information. But when the platform detects an incoming anti-ship cruise missile and enters battle mode, however, the priority of the crew entertainment video must drop to zero and yield the backbone network to mission critical data flows. In general, DRE systems require middleware that exposes mechanisms for the programmatic control of qualities of service.

Recent advances in fundamental software technologies, such as aspect-weaving software [2] and adaptive and reflec-

tive middleware, are beginning to provide the mechanisms described above. Adaptive middleware [3, 4, 5] 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 in advance 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 responsible for making these modifications while still meeting stringent end-to-end QoS requirements.

Reflective middleware [6, 7, 8, 9] 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.

1.2 Problem: Dependencies of Applications on Middleware

In many commercial application domains, such as e-commerce or consumer electronics, application software evolves faster than middleware software. 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 could not be upgraded to use these newer products easily 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).

When the consequences of these and similar dependencies are taken into account, what might seem to be a simple version replacement may in fact require a large-scale, prohibitively expensive effort. Not surprisingly, these types of problems are also found in long-lived commercial systems, such as complex telecom switches.

1.2.1 Primary Dependency of DRE Applications on Middleware

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 the use of proprietary COTS may decrease initial system acquisition 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 [10]. Using COTS products that offer only vendor-specific interfaces is therefore not generally in the long-term best interest of DRE system owners. 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.

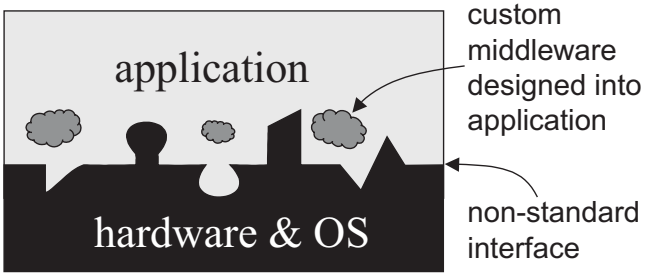

Figure 1: Historical Primary Dependencies

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.

The following example of primary dependencies is taken from a production DRE system development effort:

- 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 systems such as this were high, the maintenance costs could be low, simply because little maintenance was required. If no enhancements to such a system were needed, 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 if they are mission-critical DRE systems— are often unacceptable due to budgetary constraints. Moreover, brittle end products are also 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 software evolution.

Fortunately, the functional interface to DRE middleware products can be—and increasingly is—standardized. As a result, 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 [12], and Real-time POSIX [13]. These standards enable system integrators to choose among various COTS implementations, which can reduce the on-going, recurring cost of these systems. Moreover, some implementations of these open, standard interfaces can be configured to provide qualities of service that are suitable for many DRE applications. For example, standards-compliant Real-time CORBA implementations [14] can now be selected and configured such that their resource consumption overhead is low enough and their qualities of service are high enough for all but the most demanding DRE applications.

1.2.2 Secondary Dependency of DRE Applications on Middleware

Fortunately, many middleware products that implement standard functional interfaces are also adaptive and reflective in the sense that they permit their qualities of service to be manipulated programmatically. The interface through which such reflection and adaptation is accomplished, namely, the quality interface, is not yet standardized, however. Instead, these capabilities are provided via ad hoc proprietary configuration and control parameters.

Thus, the capabilities of COTS components to optimize their performance and resource consumption are not generally available through open standard interfaces. Consequently, any system that uses the quality interface–as DRE systems in general must–loses its infrastructure independence. 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 [15].

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. In this

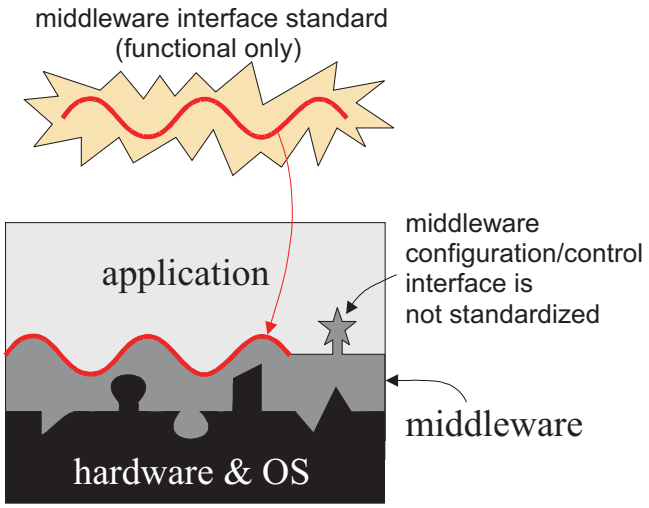

Figure 2: Secondary Dependencies

chapter, 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 [16] for data distribution. This program has avoided the primary dependency problem, since there are many products available on the open market that implement the standard 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 marshaling and demarshaling event parameters
- Efficiency of demultiplexing incoming method calls
- Threading models and thread priority settings and
- Buffer sizes, flow control, and buffer overflow handling

Most of these properties are 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, runtime environment variables, parameters passed to the ORB at initialization, and runtime interfaces for property value alteration. For example, consider the large-scale, HLA/RTI distributed interactive simulation environment described in [16]. 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 to 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. Moreover, 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 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 Sidebar 1, *Mission Critical System Modes*, on page 10.

1.3 Solution: Meta-Programming Techniques for DRE Middleware

Meta-programming [17] is a term given to a collection of technologies designed to improve software adaptability by decoupling application behavior from the various cross-cutting aspects [18] and resources used by applications. Applying meta-programming involves identifying and dissecting programming constructs into the following entities:

- Base-objects, which implement certain application-centric functionality and
- Meta-objects, which provide access to certain properties of base-objects, such as persistence, concurrency, scheduling, atomicity, ordering, state, replication, and change notifications, including the ability to modify these properties at runtime.

Meta-programming techniques can be applied to DRE middleware, where various aspects of application behavior can be affected by meta-objects, such as the smart proxies, interceptors, and interface repositories [8]. These meta-objects present a higher level of control than the base-objects that perform application-specific processing. For example, meta-objects can help connect clients to their remote server (base-)objects. They can also coordinate the QoS of resources used by DRE middleware in support of client and server applications end-to-end. A primary factor that distinguishes the QoS requirements of DRE systems from those of "best-effort" commercial systems is that 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.

Meta-programming techniques have become prevalent in DRE middleware R&D. For example, the Quality Object (QuO) middleware [3] developed at BBN Technologies applies meta-programming techniques, such as smart proxies, interceptors, and bridges, to imbue regular CORBA base-objects with QoS characteristics controlled by meta-objects. Likewise, the dynamicTAO [7] and TAO [8] reflective ORBs apply meta-programming techniques to dynamically configure ORB properties for concurrency, scheduling, security, and monitoring. As these R&D activities mature and transition into COTS middleware, they can help address the secondary dependency

problems described in Section 1.2.2, *Secondary Dependency of DRE Applications on Middleware*.

To illustrate the concept of meta-programming concretely, Section 2 describes the intent, structure, interactions, and implementation of the Quality Connector pattern and shows how this pattern can be used to address secondary dependency problems. The remainder of this chapter then discusses related work, and presents concluding remarks and a synopsis of current and future directions of work in this area.

2 The Quality Connector Pattern

The Quality Connector architectural pattern decouples application components from the QoS configuration mechanisms provided by infrastructure components to permit the infrastructure to evolve without requiring that the application be revised. As illustrated in Figure 3, the Quality Connector mediates between the application and the non-standard configuration and control interface of the middleware.

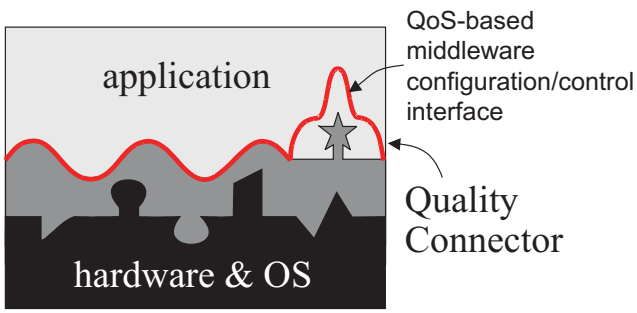

Figure 3: Role of the Quality Connector in DRE Systems

2.1 Example

CORBA event channels decouple communication between suppliers and consumers of data, as shown in Figure 4. An event channel logically mediates the communication from each supplier to all consumers, where by "logical" mediation we mean that the actual communication may use unicast, broadcast, or multicast. In many implementations, however, the event channel base-object physically mediates these communications, *i.e.*, all events are routed through a process where an event channel base-object resides. In either case, the communication between suppliers and consumers is decoupled in the sense that

- It is asynchronous, *i.e.*, consumers will receive data some time after a supplier has completed its push() operation, and

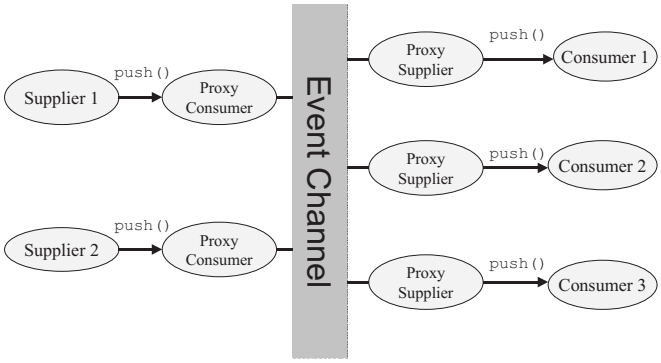

Figure 4: A Simple CORBA Event Channel

- The suppliers and consumers must be aware of the event channel's identity, but 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.

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

- Latency of event delivery
- Where and how often event data are copied
- Threading and synchronization policies for event dispatching
- 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
- How and where event data are buffered, and how large the event data buffers are
- What happens when an event data buffer overflows
- Reliability of event delivery
- Whether events from one supplier will be delivered to each consumer in the order in which they were supplied
- 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?
- 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?

Consider a DRE application that uses the CORBA Event Service and that will meet or not meet its requirements depending on the value of one or more or the event delivery properties outlined above. Such an application cannot – or certainly should not – pass any design review since its correct functioning cannot be predicted with any certainty. The application may or may not pass its acceptance test, depending on the properties of the chosen Event Service implementation and the cleverness of the test plan. The application may fail if it is ported to a platform that includes a different Event Service implementation. Finally, the application may fail if a new Event Service implementation is installed in the field, including the case when the new implementation is a "minor" revision to the same vendor's previous implementation. This situation is clearly unacceptable.

2.2 Context

The Quality Connector pattern can be applied in a DRE system that has the following characteristics:

- It uses components via standardized functional interfaces,
- The qualities of the services provided by those components are critical to the system's conformance to its requirements, and
- Long-term maintainability and portability are necessary for the success of the system.

2.3 Problem

Implementations of services that are available through standardized functional interfaces expose only non-standard mechanisms for controlling the qualities of the services provided. When an application uses such a service implementation, four forces arise:

- A quality-sensitive application should be able to monitor and control the qualities of its supporting services. The required qualities should be permitted to depend on system mode (see Sidebar 1).
- A long-lived application should be capable of executing without manual modifications on multiple implementations of infrastructure services with standard functional interfaces.
- Applications that are programmed to use non-standard QoS control mechanisms directly are vulnerable to the secondary dependencies described in Section 1.2.2, *Secondary Dependency of DRE Applications on Middleware*.
- For time-critical mode transitions, infrastructure resources must be reallocated quickly to provide the services required in the new mode.

Sidebar 1: Mission-Critical System Modes

Mission-critical systems are often characterized as a hierarchy of parts that we call configuration items. A configuration item may be small (such as a motherboard 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.

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.

We can now 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 the modes of a system.

The qualities of distributed communication services that applications require will differ in different modes. The example was given above of a crew entertainment video whose priority drops when the ship enters battle mode. 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 can be 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. 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."

2.4 Solution

For each infrastructure component that provides only a non-standard QoS-control interface, implement a Quality Connector object. The Quality Connector object configures the infrastructure component to provide, if possible, the requested QoS in the specified system modes. The interface between the application and the Quality Connector object should be independent of the choice of infrastructure component implementation, and be concerned only with

- The qualities of the service provided,
- The load that will be imposed on the service, and
- The modes of the system.

If a new infrastructure implementation is employed, then a corresponding Quality Connector object will be required that provides the same interface to the application as before. The application will therefore not require manual modification. The runtime interface between the Quality Connector object and the component implementation depends on the infrastructure component's QoS-control interface.

2.5 Structure

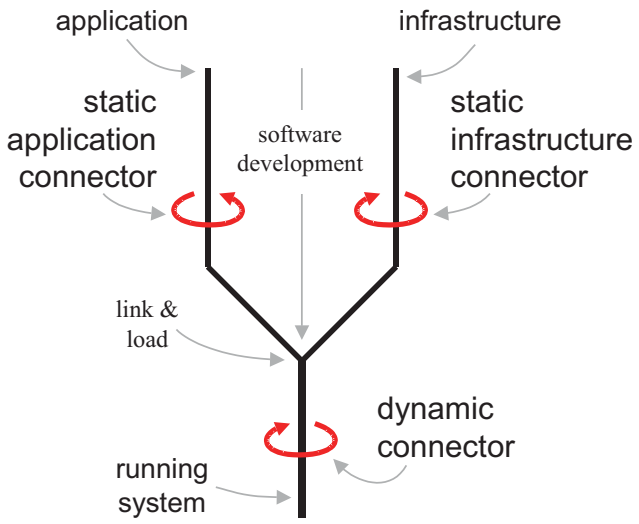

Figure 5: Quality Connector Participants

The Quality Connector pattern includes the participants shown in Figure 5 and described below.

- The Static Application Connector component acts on the application source code before it is compiled and is akin to aspect weaving tools, such as AspectJ [2]. That is, the Static Application Connector scans that application source code to detect statements and declarations that are related to the service being provided; this detection process may be as sophisticated as that used in globally optimizing compilers, or as simple as the detection of flags embedded in comments. Then the Static Application Connector modifies the source code at certain of these locations, generating new source code.

  → For example, consider an application that intends to supply events to an Event Service, as described in section 2.1, and whose QoS requirements are known statically. Such an application must first create an Event Service access point called a ProxyPushConsumer by invoking the `obtain_push_consumer()` method. The Static Application Connector component of the Quality Connector locates these method invocations in the application source code, and inserts new code after each that proposes the appropriate contract.
- The Static Infrastructure Connector component acts on the underlying middleware components before they are linked into the deployed system. This action may be as simple as selecting one of several implementations of an interface, and may be as complex as re-compiling and re-linking the middleware component using appropriately chosen values for configuration parameters such as include file search paths, macro symbol definitions, and compiler options.

  → For example, the TAO ORB, which we use for its Event Service, is highly configurable by both runtime and compile-time mechanisms. Specifically, we exploit the efficiencies available when the target system is known to be homogeneous by enabling a macro in an include file that streamlines the marshalling and de-marshalling process.
- The Dynamic Connector component is linked in with the application and acts during its operation. This component allocates resources, such as ATM circuits, processors, and radios, to data flows. When the quality connector object receives a request for a QoS contract, it uses the Configuration object (see below) or similar mechanism to discover the infrastructure base-objects and corresponding meta-objects that might be used to provide the requested service in the specified mode. It then negotiates with the meta-objects in an attempt to obtain support for the requested service. If these negotiations are successful, the quality connector object records the successful strategy, and directs the meta-objects involved to record their commitment to this contract.

  → For example, since an Event Service is permitted by the CORBA specification to use any mechanism to propagate events from suppliers to consumers, the Dynamic Connector component can (and should) examine the available communication resources to determine the

best means of propagating events.

In addition to the participants of the Quality Connector pattern described above, there are several optional participants, including

- Configuration tools that assist system engineers in selecting compatible sets of infrastructure components that implement required services,
- Simulation tools to determine whether locally specified qualities of service will combine to meet system-level requirements, and
- A Configuration object that provides visibility at runtime of the set of configuration items that currently comprise the executing system.

These optional participants are not addressed in this chapter.

2.6 Dynamics

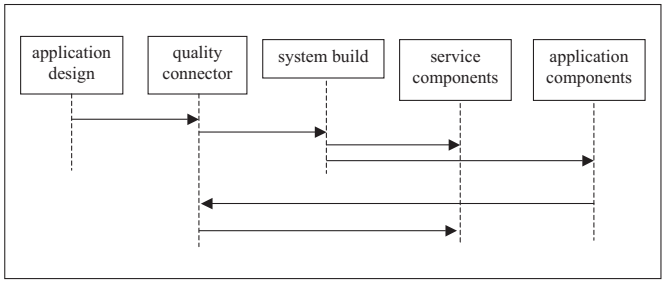

Figure 6: Quality Connector Dynamics

The behavior of the Quality Connector pattern can be divided into the three phases shown in Figure 6 and described below:

1. **Pre-runtime**. When the identities of the services to which QoS requests will be made are known, the application source code is, if necessary, modified automatically to insert the code that makes the runtime requests. Infrastructure components are selected and constructed using whatever information is known about the QoS required and load imposed on the service.
2. **Runtime preparation**. At runtime, the application requests a QoS in a specified mode, including the specification of a load. The Quality Connector determines whether that request could be satisfied using the presently available infrastructure, considering the QoS contracts accepted previously. If the request would be feasible, the application is granted a contract, and the strategy by which the service would be provided is recorded. Moreover, listeners are attached to the configuration items whose mode changes might signal transition to or from the relevant mode.
3. **Runtime employment**. After a QoS contract has been established, when the system enters the mode to which that contract applies, the Quality Connector receives notification of the mode change, and reallocates infrastructure resources immediately according to its pre-computed strategy.

2.7 Implementation

After a configurable infrastructure service has been selected, a quality connector for that service can be implemented as follows:

1. Define a small language in which acceptable values (or sets of acceptable values) of the service's qualities can be specified, depending on the system mode. Consider defining this language using XML so that it can be understood readily by humans and parsed easily by COTS tools. This activity can take place even in advance of the system design; ideally the language will be defined by an open standard.
2. Provide configuration-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.
3. Implement the Dynamic Connector. This is the Dynamic Connector component of the Quality Connector, described in section 2.5; it carries out the runtime allocation of resources.

We describe each of these implementation activities below.

2.7.1 Specify the Quality Connector QoS Language

Define a Quality Connector QoS language that is capable of specifying

- Values for all qualities of the this service that are of interest in the system,
- Values for all relevant parameters of the load that the clients will impose on the service,
- Relative priorities of clients, for use when not all requests can be supported, and
- System modes in which quality requests apply.

Consider specifying this language in a form that is easy for humans to read, such as XML.
→ A QoS language that applies to a CORBA Event Service is illustrated in Figure 7. We have not used worst-case bounds for qualities such as latency, on the ground that if "worst case" is interpreted literally, then resource utilization will necessarily be low. Rather, 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

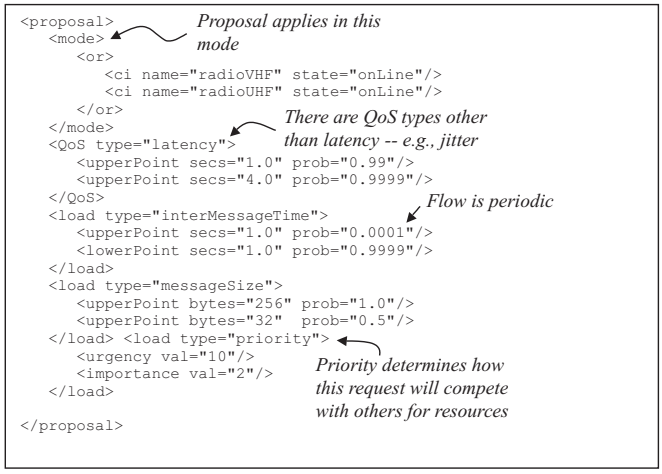

Figure 7: A Proposal in XML

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."

The load that will be imposed by the event service is specified in terms of a distribution of event sizes, in bytes, and a distribution of the times between event-push invocations. Relative priorities of clients are specified by two integers: 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.

The proposal in Figure 7 applies only when either of a pair of tactical military or emergency response team radios is online. 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 50of 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 the two integral values outlined above.

2.7.2 Providing Build-time Tools

Procure or build tools to help the programmers conduct the configuration-time and runtime Quality Connector activities. The decisions concerning which tools to use, if any, are subject to cost/benefit tradeoffs, such as the cost to build or buy the tool plus the tool maintenance cost vs. the anticipated productivity improvement and risk reduction from its use. Consider using design tools, such as design- tool interfaces, QoS language checkers, simulators, and source code generators, such as AspectJ [2] or scripts.

→ For our Event Channel service, our QoS language is in XML, so schemas are a natural mechanism for language checking. Our application is written in C++, so we explicitly mark locations in the source code where modifications are to be applied, and we use a Perl script to insert the QoS requests automatically.

2.7.3 Implement the Dynamic Connector

The Dynamic Connector component implements the runtime functionality of the Quality Connector. This component is therefore responsible for

- Receiving QoS contract requests from the application,
- Negotiating with the available resource meta-objects for support,
- Replying to the application's request with either a QoS contract or an explicit denial, and
- Distributing strategies to the components that will employ them if the contract is granted.
- In our CORBA Event Service, QoS contracts are requested by the application from the Proxy PushSupplier and ProxyPushConsumer objects. These forward the request to the event channel object, which negotiates with the resource meta-objects.

The event channel object first requests service from the resource meta-objects with a parameter called pullRank set to false, which has the effect of attempting to provide the requested service without disrupting any existing service contracts. If this negotiation fails, then the event channel object tries the negotiation again but with pullRank set to true; which has the effect that if this second round of negotiation succeeds, then at the time when the negotiated contract comes into force, contracts of less importance than the present request may be abrogated.

If the negotiation process succeeds, then a collection of resources will be allocated for the event flow in the specified mode. The event channel object distributes strategy objects, represented as XML strings, to the affected service ob-

jects. For example, the strategy given to a ProxyPushConsumer might direct the immediate creation of a socket with specified parameters to which supplied events should be written.

2.8 Example Resolved

The use of the CORBA Event Service can be enhanced via the Quality Connector pattern to allow applications to control qualities of the event service without affecting its implementation. To accomplish this, we permit the application to associate a QoS contract of the form shown in Figure 7 with each consumer and supplier proxy. These contracts support the requirement that QoS be permitted to depend on system mode.

To avoid manual modifications to the application source code, we provide automatic tools possessing aspect-oriented programming (AOP) [18] capabilities to insert the required calls to request QoS contracts, following the creation of each Event Service proxy. The rejection of a QoS request raises an exception. As a result, no manual modification of application code is required when a different event service implementation is used, thereby avoiding secondary dependency problems.

The runtime result of a QoS-request is that the proposed contract is forwarded from the proxy to the Event Service object, where the negotiation for infrastructure support takes place with the infrastructure meta-objects. If the mode specified in the contract is not the current mode, then the strategy for the specified mode is retained in the proxy object, for use when the specified mode is entered. As a result, the infrastructure resources are reallocated quickly when the mode is entered, as required for time- critical mode transitions.

2.9 Known Uses

Meta-INterface for Real-time Embedded Systems (MINERS). There is an ongoing independent research and development project at Lockheed Martin Tactical Systems in Eagan, Minnesota, USA, called MINERS. 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, such as CORBA and Real-time Java. In addition to the functional software that uses these interfaces, applications specify their required qualities of service (QoS), such as the latency of event delivery or the 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 runtime, uses the configuration/control interfaces of the (necessarily adaptive and reflective) middleware to monitor and enforce the qualities of service specified by DRE applications.

QuO. The BBN Quality Objects (QuO) framework [3] was a Quorum project that uses QoS definition languages [4] that are based on the separation of concerns promoted by AOP [18]. In particular, QuO includes the notion of a connection between a client and an object, which encapsulates QoS requirements and intended usage patterns; this is analogous to MINERS contracts. QuO provides system condition objects, which are similar to MINERS modes. QuO provides a Quality Description Language (QDL) that includes three aspect languages:

- A contract description language (CDL) that describes contracts as outlined above,
- A structure description language (SDL) that describes the internal structure of object implementations and the amount of resources they require, and
- A resource description language (RDL) that describes the available resources and their status.

These languages perform functions similar to the MINERS QoS language described in section 2.7.1.

QuO has in the past emphasized reactive resource allocation [19], which monitors the QoS being provided and acting to correct contract violations or anticipated violations. There is nothing inherent in the structure of QuO, however, that prohibits implementing the proactive resource allocation style described in this paper.

Human uses. Applications behave analogously to an executive who gives a package to his staff with direction that it must be delivered by a specified time. The Quality Connector acts, analogously to the staff, by selecting mechanisms for transport and setting the controllable parameters of those mechanisms.

2.10 Consequences

The Quality Connector pattern has the following benefits:

- **Infrastructure independence**. The Quality Connector pattern decouples an application from dependencies on the infrastructure it executes upon, even when that infrastructure requires explicit configuration to provide the QoS that the application requires.
- **Fast response to mode changes**. The Quality Connector negotiates contracts for service in modes other than the current mode, and in so doing performs the possibly long and difficult determination of the necessary resource allocations. When the new mode in fact arises, the resources can then be reallocated quickly.

The Quality Connector pattern also has the following liabilities:

- **Reimplementation**. When a new infrastructure is deployed, a new Quality Connector object may be required.
- **Potential for low utilization**. Resource utilization may be low if the Quality Connector implementations in a system are unduly conservative in accepting contract proposals.

2.11 See Also

The Quality Connector pattern is related to the Component Configurator pattern [20], which permits various component implementations to be linked/unlinked into/from a running application without shutting down the application. Both the Quality Connector pattern and the Component Configurator pattern provide means to change the behavior of a service during application execution. The Component Configurator is concerned with one mechanism – dynamic linking – for doing so, while the Quality Connector focuses on policies and mechanisms that ensure rapid response to changing system state by switching present components among pre-computed and pre-supplied strategies.

The goal of the Quality Connector pattern is similar to that of the Interceptor pattern [20], in that both adjust infrastructure behavior without modifying the application manually. The Interceptor pattern accomplishes this by a highly flexible method of adding services that are triggered automatically when specified events occur. The Interceptor pattern applies to the design of a framework, specifying that it expose application-callback interfaces, and that it open aspects of its internal state and behavior to control by the application. The Quality Connector pattern imposes no design requirements on its constituent components (although if the service components expose only weak configuration controls, then the Quality Connector will be unable to provide good resource utilization.) The Quality Connector is concerned with service quality, while the Interceptor pattern is more general, and can provide functions such as event logging.

The Reflection pattern [21] provides for the inclusion in a running application of meta-objects that provide information about, and control over, the base-objects that implement the application logic. This pattern is clearly a basis on which the Quality Connector pattern depends since the latter requires that the infrastructure services support reflective capabilities. The Proxy pattern [22] shields an application from details of the implementation of a service, *e.g.*, it hides the physical location of a service implementation from its clients. The Quality Connector pattern serves to modify the behavior of existing, fully visible, service-providing components.

3 Related Work

Our work on Quality Connectors complements the work being done on the DARPA Quorum program [23], particularly the QuO framework [3]. Quorum's goal was to develop technologies that allowed 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 Distributed Multimedia Research Group at Lancaster University has proposed and implemented a prototype of next-generation reflective middleware [6] 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 [5].

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 [7] and Reflective CCM [8] projects have demonstrated that CORBA can be reconfigured at runtime by dynamically linking and unlinking certain components. Similarly, AspectIX [9] 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). Work in this area is underway in various places, including Xerox PARC [18], IBM [24], and MCC [25]. We have chosen to use AspectJ [2], which is an aspect-oriented extension to the Java programming language. AspectJ addresses the problem of cross-cutting 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 [26], 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].

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. The IETF has specified mechanisms for scalable differentiated [28] and integrated [29] classes of service on the Internet.

- **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.

4 Concluding Remarks and Future Directions

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

COTS middleware, such as Real-time CORBA [11], 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) [30] 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.

The potential affordability gains offered by COTS middleware have therefore become strategically important. Without a product- and component-independent mechanism to configure COTS middleware optimally, however, this affordability gain is threatened.

Our prior experience [16, 8, 14, 17, 5] illustrates that effective operation, interoperability, and integration of complex DRE systems 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 and practitioners therefore have

a pressing need to coordinate individual advances in the COTS solution space that are being addressed by different sectors of the DRE R&D community.

The problems faced by researchers and developers of DRE systems are highly challenging, with many interlocking aspects [1]. Unless pieces of the emerging, independently developed, COTS solutions can be delivered to application designers as coordinated, integrated packages, their value will be diminished. In fact, COTS can even make matters worse instead of better, *e.g.*, due to excessive costs for COTS refresh and integration [15]. This chapter proposes an architectural pattern called Quality Connector that allows a variety of separately developed, and continuously evolving, tools and components to appear to application designers as an integrated, coordinated, and stable infrastructure. Instantiations of the Quality Connector pattern encapsulate the various configuration and control mechanisms provided by COTS middleware, thereby exposing a stable QoS-based interface to applications.

Implementation of the capabilities described in this chapter 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 ACE framework [31] and the TAO Real-time CORBA ORB [14]. ACE and TAO are highly configurable middleware based on patterns [20] that support DRE applications with demanding QoS requirements.

In the longer term, if the mission-critical DRE system community can achieve a shared understanding of what qualities of services need to be specified and how to specify them, we envision the availability of middleware that can be configured to meet such requirements, and the development of applications that include their QoS requirements as part of their design. These applications should be far more stable over evolving infrastructure than current applications. Moreover, they may also be verifiable independently of any infrastructure, based on their QoS requirements, which will substantially reduce costs in mission- critical DRE applications.

References

[1] R. E. Schantz and D. C. Schmidt, "Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications," in *Encyclopedia of Software Engineering* (J. Marciniak and G. Telecki, eds.), New York: Wiley & Sons, 2001.

[2] The AspectJ Organization, "Aspect-Oriented Programming for Java." `www.aspectj.org`, 2001.

[3] J. A. Zinky, D. E. Bakken, and R. Schantz, "Architectural Support for Quality of Service for CORBA Objects," *Theory and Practice of Object Systems*, vol. 3, no. 1, pp. 1–20, 1997.

[4] P. Pal, J. Loyall, R. Schantz, J. Zinky, R. Shapiro, and J. Megquier, "Using QDL to Specify QoS Aware Distributed (QuO) Application Configuration," in *Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*, (Newport Beach, CA), IEEE/IFIP, Mar. 2000.

[5] C. D. Gill, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-Time CORBA Scheduling Service," *Real-Time Systems, The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*, vol. 20, March 2001.

[6] Gordon S. Blair and G. Coulson and 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*, (London), pp. 191–206, Springer-Verlag, 1998.

[7] F. Kon, M. Roman, P. Liu, J. Mao, T. Yamane, L. Magalhaes, and R. Campbell, "Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB," in *Proceedings of the Middleware 2000 Conference*, ACM/IFIP, Apr. 2000.

[8] N. Wang, D. C. Schmidt, M. Kircher, and K. Parameswaran, "Towards a Reflective Middleware Framework for QoS-enabled CORBA Component Model Applications," *IEEE Distributed Systems Online*, vol. 2, July 2001.

[9] F. J. Hauck, U. Becker, M. Geier, E. Meier, U. Rastofer, and M. Steckermeier, "The AspectIX Approach to Quality-of-Service Integration into CORBA," Tech. Rep. TR-I4-99-09, Friedrich-Alexander University, Erlangen-Nurnberg, Germany, 1999.

[10] S. T. S. C. Department of the Air Force, "Guidelines for Successful Acquisition and Management of Software Intensive Systems: Volume 1 – Version 3.0." `http://web2.deskbook.osd.mil/reflib/DAF/035GZ/013/035GZ013DOC.HTM#T2`, May 2000.

[11] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.6 ed., Dec. 2001.

[12] Bollella, Gosling, Brosgol, Dibble, Furr, Hardin, and Turnbull, *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[13] B. Gallmeister, *POSIX.4 Programming for the Real World*. Sebastopol, California: O'Reilly, 1995.

[14] D. C. Schmidt, D. L. Levine, and S. Mungee, "The Design and Performance of Real-Time Object Request Brokers," *Computer Communications*, vol. 21, pp. 294–324, Apr. 1998.

[15] J. Clapp and A. Taub, "A Management Guide to Software Maintenance in COTS-Based Systems," Tech. Rep. MP 98B0000069, The MITRE Corporation, Bedford, MA, November 1998 1998.

[16] C. O'Ryan, D. C. Schmidt, and J. R. Noseworthy, "Patterns and Performance of a CORBA Event Service for Large-scale Distributed Interactive Simulations," *International Journal of Computer Systems Science and Engineering*, vol. 17, Mar. 2002.

[17] N. Wang, D. C. Schmidt, O. Othman, and K. Parameswaran, "Evaluating Meta-Programming Mechanisms for ORB Middleware," *IEEE Communication Magazine, special issue on Evolving Communications Software: Techniques and Technologies*, vol. 39, Oct. 2001.

[18] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-Oriented Programming," in *Proceedings of the 11th European Conference on Object-Oriented Programming*, June 1997.

[19] Joseph K. Cross and Patrick Lardieri, "Proactive and Reactive Resource Reallocation in DoD DRE Systems," in *Proceedings of the OOPSLA 2001 Workshop "Towards Patterns and Pattern Languages for OO Distributed Real-time and Embedded Systems"*, Oct. 2001.

[20] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. New York: Wiley & Sons, 2000.

[21] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture – A System of Patterns*. New York: Wiley and Sons, 1996.

[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Massachusetts: Addison-Wesley, 1995.

[23] DARPA, "The Quorum Program." www.darpa.mil/ito/research/quorum/index.html, 1999.

[24] P. Tarr and H. Ossher and W. Harrison and S.M. Sutton, "N Degrees of Separation: Multi-Dimensional Separation of Concerns," in *Proceedings of the International Conference on Software Engineering*, May 1999.

[25] R. E. Filman, S. Barrett, D. D. Lee, and T. Linden, "Inserting ilities by controlling communications," *Communications of the ACM*, 2002.

[26] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. Boston: Addison-Wesley, 2000.

[27] D. I. T. Office, "The Programmable Composition of Embedded Software (PCES) Program." `http://www.darpa.mil/ito/research/pces/index.html`.

[28] Internet Engineering Task Force, "Differentiated Services Working Group (diffserv) Charter." www.ietf.org/html.charters/diffserv-charter.html, 2000.

[29] Internet Engineering Task Force, "Integrated Services Working Group (intserv) Charter." www.ietf.org/html.charters/intserv-charter.html, 2000.

[30] "About Global Air Traffic Management." `http://www.hanscom.af.mil/esc-gat/aboutgatm.htm`.

[31] D. C. Schmidt and S. D. Huston, *C++ Network Programming, Volume 1: Mastering Complexity With ACE and Patterns*. Boston: Addison-Wesley, 2002.