# Quality Connector

## An Architectural Pattern to Enhance QoS and Alleviate Dependencies in Distributed Real-time and Embedded Middleware

Joseph K. Cross
Lockheed Martin Tactical Systems
P.O. Box 64525, M.S. U2N29
St. Paul, MN 55164-0525, USA
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 distributed real-time and embedded (DRE) applications functional support for standard interfaces, along with the ability to optimize application resource utilization. For example, a Real-time CORBA object request broker (ORB) permits DRE 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" – acts against the very product-independence that standardized COTS interfaces are intended to provide. This paper presents an architectural pattern called Quality Connector, which is a meta-programming 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.*

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 manual changes to application component functionality. The Quality Connector mediates between the application and non-standard middleware configuration and control interfaces.

## Example

CORBA *event channels* [2] decouple communication between suppliers and consumers of data, as shown in Figure 1. 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 any type of unicast, broadcast, or multicast protocol. In many implementations, however, the event channel object *physically* mediates these communications, *i.e.,* all events are routed through a process where the event channel 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
- The suppliers and consumers must be aware of the event channel's identity, but need not be aware of each other' s identities.

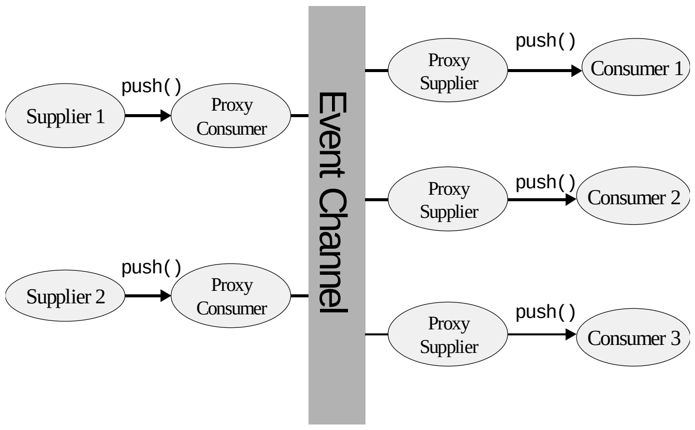


**Figure 1. A Simple CORBA Event Channel**

There is no pre-defined limit on the number of suppliers and consumers that can be connected to a CORBA 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 distributed system.[3]

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

[3] Our discussion focuses on the "push" model of event delivery, where a supplier invokes a `push(data)` operation to supply any type of data, and the event channel causes `push(data)` operations to be invoked on the consumers registered with that event channel. In addition to the "push" model, there is also a "pull" model of event delivery, which we do not address in this paper.

- 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 distributed real-time and embedded (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. It should be possible to determine whether a given Event Service implementation will successfully support the application. It should also be possible to port the application easily from one implementation of the service implementation to another. Finally, it should be possible to modify the service implementation – including making changes to hardware and revising support software – and retain confidence that the application will continue to function correctly and with the appropriate quality of service.

## Context

The Quality Connector pattern can be applied in a DRE application 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 application’s conformance to its requirements, and
- Long-term maintainability and portability are necessary for the success of the application.

## Problem

Implementations of services that are available through standardized functional interfaces expose only non-standard mechanisms for controlling the qualities of the services provided, such as throughput, latency, jitter, scalability, dependability, and security. When an application uses such a service implementation, three 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 the current 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.
- For time-critical mode transitions, infrastructure resources, by which we mean resources such as ATM virtual circuits, processors, or radios, 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* (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).

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. Consider the example 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 their 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."

## Solution

Implement a *Quality Connector* object for each infrastructure component[4] that provides only a non-standard QoS-control interface. 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 should 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.

*In detail*: Before the application source code is compiled, a *static application connector* acts on that code (or on some higher-level representation of it, such as a model), inserting

[4] An "infrastructure component" is any hardware or s oftware component of the deployable system whose function is to provide infrastructure services to the application; an ORB together with its supporting computing and communication hardware is an example.

hooks through which the *dynamic connector* will act at application run-time to configure the infrastructure components. In addition, a *static infrastructure connector* selects and configures the implementations of infrastructure services before the system is linked.

If a new infrastructure component 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 changes to its functionality. The runtime interface between the Quality Connector object and the component implementation depends on the infrastructure component's QoS-control interface.

## Structure

A Quality Connector consists of three components:

- The *Static Application Connector* component acts on the application source code before it is compiled and may operate similarly to "aspect weaving" tools, such as AspectJ [12]. For example, the Static Application Connector scans the 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. The Static Application Connector then 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 above, and whose QoS requirements are known statically. Such an application must first create an Event Service access point called a `ProxyPushConsumer` by invoking the standard `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 will request the appropriate QoS.

| ***Class*** | ***Collaborator*** |
|---|---|
| Static Application Connector | ! Application<br>! Dynamic Connector |
| ***Responsibility***<br>! Accepts QoS specifications for infrastructure services<br>! Modifies the application source code as required by the Dynamic Connector | |

- 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 or it 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 [19], which we use for its Real-time Event Service [31], is highly configurable by both runtime and compile-time mechanisms [2]. 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 marshaling and demarshaling activities.

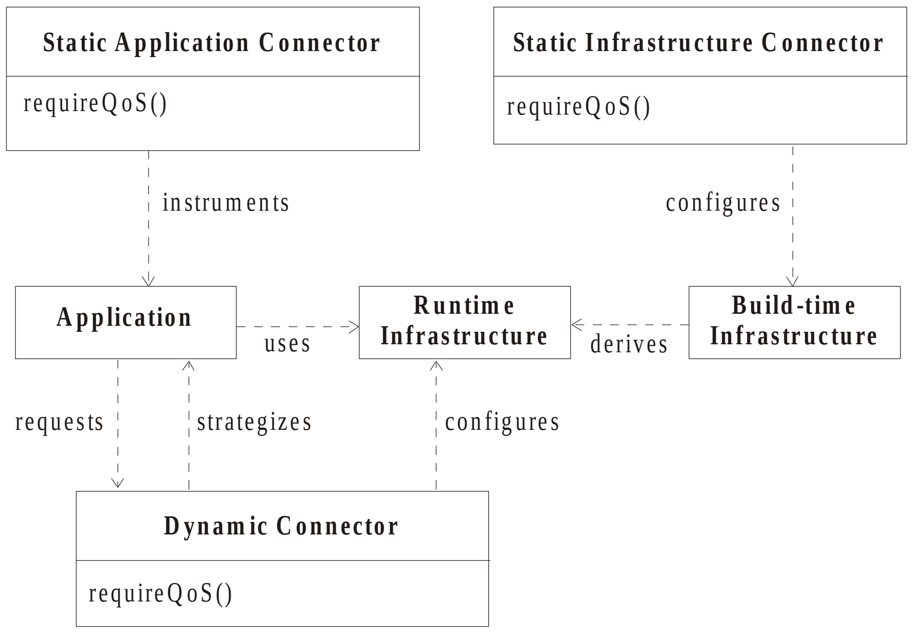
| ***Class*** | ***Collaborator*** |
|---|---|
| Static Infrastructure Connector | ! Middleware Components<br>! Dynamic Connector |
| ***Responsibility*** | |
| ! Accepts QoS specifications for infrastructure services<br>! Selects or modifies the middleware components that will be available to the Dynamic Connector | |

- The *Dynamic Connector* component is linked in with the application and acts during its operation. This component allocates infrastructure resources to data flows. When the quality connector object receives a request for a specific QoS, it uses the Configuration object (see below) or similar mechanism to discover the infrastructure components that might be used to provide the requested service in the specified mode. It then negotiates with the infrastructure resources 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 resources involved to record their commitment to this QoS in this mode.

➥ 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 to propagate events.

| ***Class*** | ***Collaborator*** |
|---|---|
| Dynamic Connector | ! Application<br>! Infrastructure resources (or their proxies)<br>! Middleware components |
| ***Responsibility*** | |
| ! Negotiates with infrastructure resources (or their proxies)<br>! Allocates infrastructure resources to data flows<br>! Configures middleware components during application execution | |

The class diagram for the Quality Connector pattern is shown in the following figure:



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

- Configuration tools that assist system builders 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 run-time of the set of configuration items that currently comprise the executing system.

These optional participants are not addressed further in this paper.

## Dynamics

The dynamic sof the Quality Connector pattern is illustrated in Figure 2. These interactions can be divided into the three phases as described below:

1. *Pre-runtime*. When the identities of the services to which QoS requests will be made are known, the application source code can be 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 requirements and load imposed on the service.
2. *Runtime preparation*. The runtime dynamics of the Quality Connector are illustrated in Figure 2. At runtime, the application requests a QoS in a specified mode, including the specification of a load. The code included by the Quality Connector determines whether that request could be satisfied using the presently available infrastructure, considering any extant QoS agreements. If the request would be feasible, the QoS request is granted, 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 agreement has been established and the system enters the mode in which that agreement applies, the code included by the Quality Connector receives notification of the mode change and reallocates infrastructure resources immediately according to its pre-computed strategy.

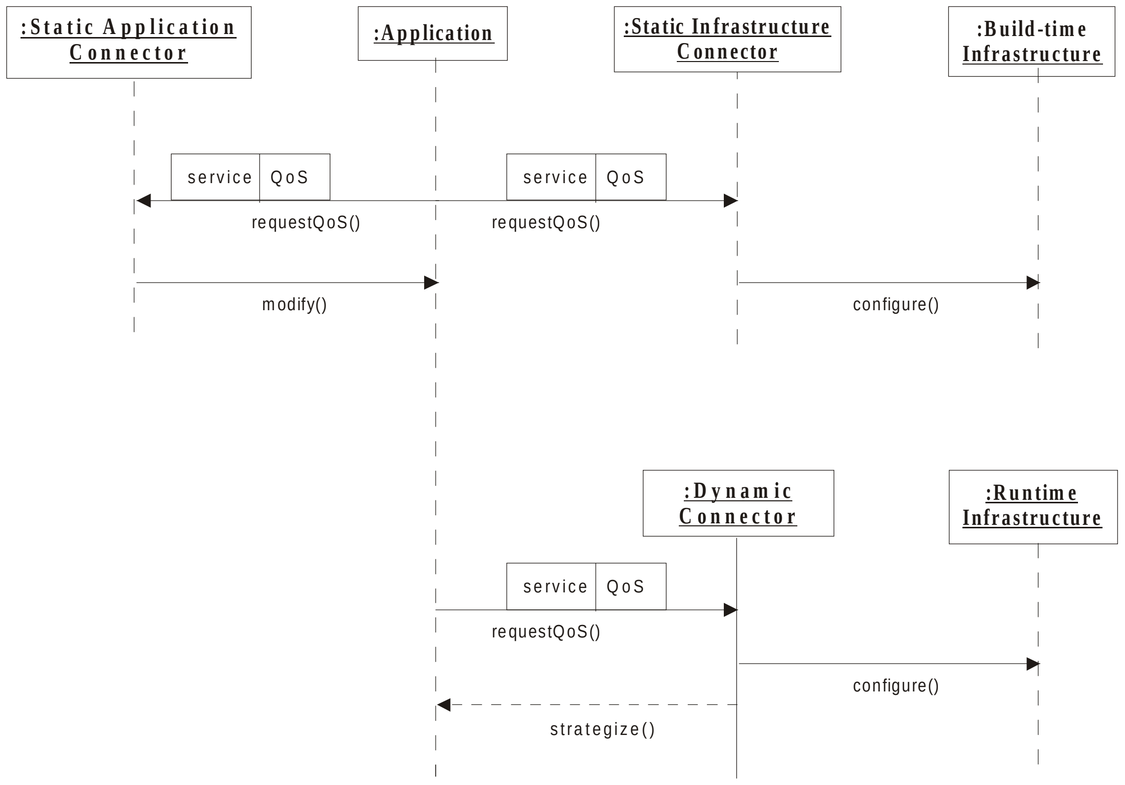


**Figure 2. Dynamics of the Quality Connector**

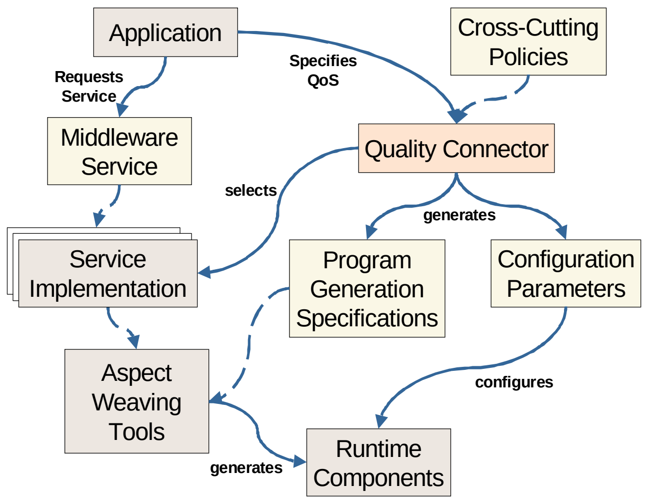
## Implementation

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. This language is the form in which data flows over the "Specifies QoS" arrow in the figure below. 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, as are, for example, UML and XML.
2. **Provide configuration-time tools** to check for feasibility and consistency of the requested quality values, and to set the properties of the Runtime Components to provide the required qualities, as illustrated below.

3. **Implement the Dynamic Connector**. This is the Dynamic Connector component of the Quality Connector, described above; it carries out the runtime allocation of resources. This function is performed in the Middleware Service box below.

The following figure outlines how these activities interact when implementing the Quality Connector pattern:



We describe each of these implementation activities below.

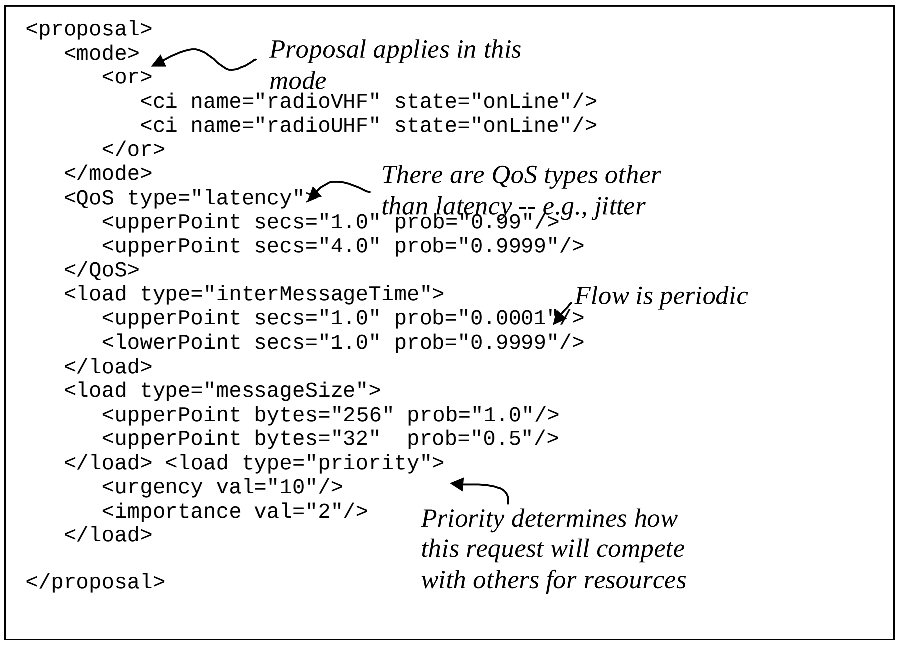
**Step 1: Specify the Quality Connector QoS Language**

Define a Quality Connector QoS language that is capable of specifying

- Values for all qualities of the 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.

➪ A QoS language that applies to a CORBA Event Service is illustrated in Figure 3. 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 may be too low to be effective for production DRE applications. 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 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."[6]

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



```
<proposal>
   <mode>
      <or>
         <ci name="radioVHF" state="onLine"/>
         <ci name="radioUHF" state="onLine"/>
      </or>
   </mode>
   <QoS type="latency">
      <upperPoint secs="1.0" prob="0.99"/>
      <upperPoint secs="4.0" prob="0.9999"/>
   </QoS>
   <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"/>
      <importance val="2"/>
   </load>

</proposal>
```

Proposal applies in this mode
There are QoS types other than latency -- e.g., jitter
Flow is periodic
Priority determines how this request will compete with others for resources

**Figure 3. A QoS Request in XML**

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 the following two integral values:

- 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 not both of which can 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 is abrogated.

The proposal in Figure 3 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 the two integral values defined above.

**Step 2: Provide Configuration-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 [12] 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.

**Step 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 requests from the application,
- Negotiating with the available infrastructure resources for support,
- Replying to the application's request for QoS with either acquiescence or an explicit denial, and, if acquiesced, then
- Distributing strategies to the components that will employ them when the applicable mode is entered.

➩ In our CORBA Event Service, QoS requests are made by the application through the `ProxyPushSupplier` and `ProxyPushConsumer` objects. These forward the request to the event channel object, which negotiates with the infrastructure resources.

The event channel object first requests service from the infrastructure components with a parameter called `pullRank` set to `false`, which has the effect of attempting to provide the requested service without disrupting any existing QoS agreements. 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 presently negotiated QoS is required, then agreements of less importance than the present request may be abrogated.

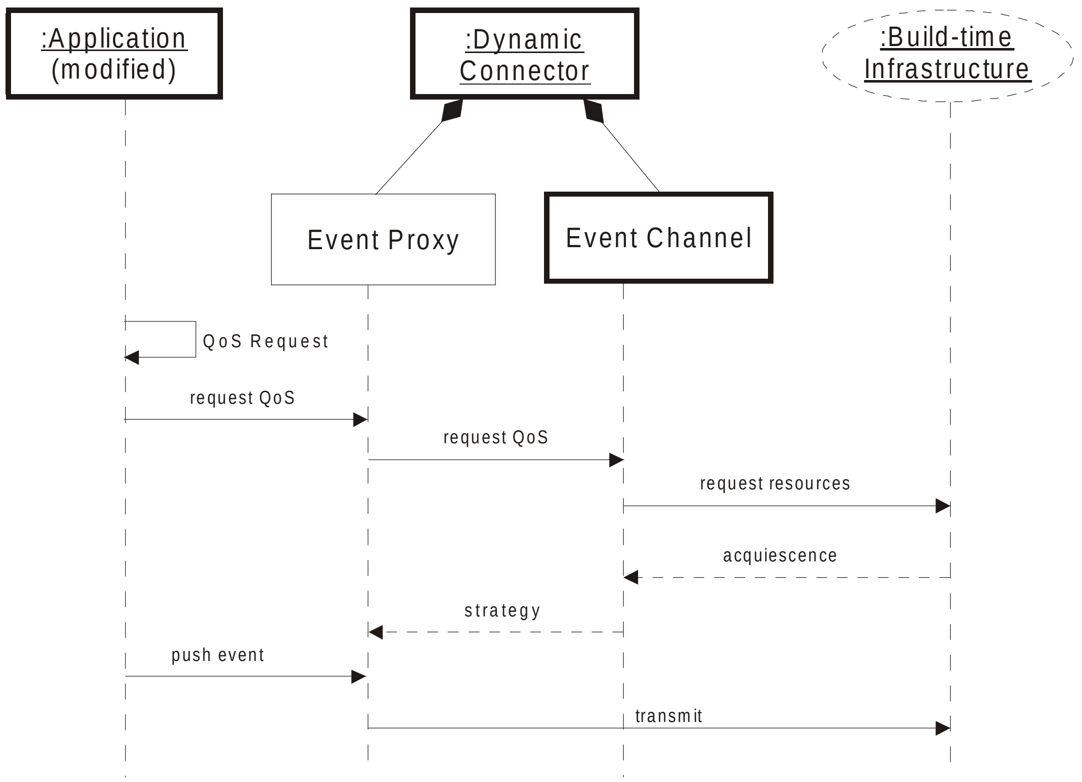
If the negotiation process succeeds, then a collection of resources will be allocated for the event flow in the specified mode. The event channel object distributes strategy objects, represented as XML strings, to the affected service objects. For example, the strategy given to a `ProxyPushConsumer` might direct the immediate creation of a socket with specified parameters to which supplied events should be written.

**Example Resolved**

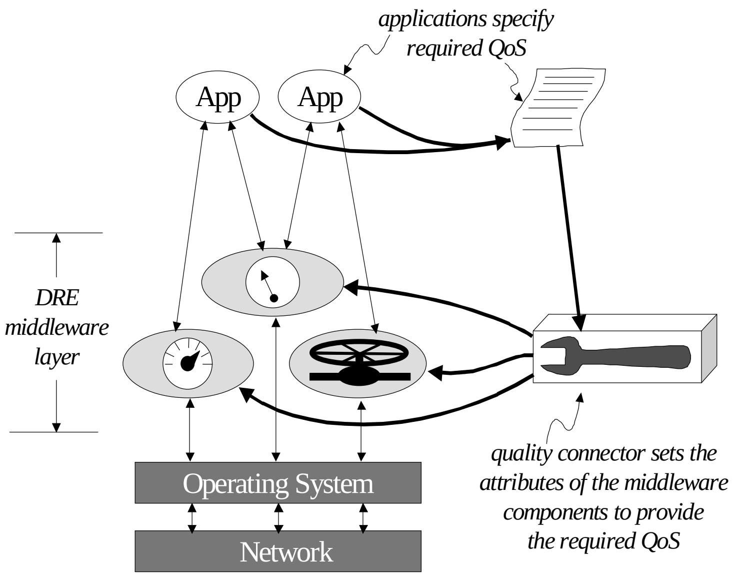
We can apply the Quality Connector pattern to enhance the CORBA Event Service so applications can control qualities of the event service without being unduly affected by its implementation. To accomplish this, we permit the application to submit a QoS request of the form shown in Figure 3 with each consumer and supplier proxy. These requests support the requirement that QoS be permitted to depend on system mode.

To avoid manual modifications to the application source code, we can apply automatic tools possessing aspect-oriented programming (AOP) [4] capabilities to insert the required calls to request QoSs, 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.

The runtime result of a QoS request is that the request is forwarded from the Event Service proxy to the Event Channel object, where the negotiation for infrastructure support takes place with the infrastructure resources or their proxies. If the mode specified in the request 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. These interactions are shown in the following figure:



In summary, the following figure illustrates the tool interactions (e.g., the instrumentation and weaving of aspects).


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

## 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.

**QuO**. The BBN Quality Objects (QuO) framework [6] uses QoS definition languages [7] that are based on the separation of concerns promoted by AOP [4]. 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 QoS requests. QuO provides *system condition* objects, which are similar to MINERS modes. QuO provides a Quality Description Language (QDL) that includes three aspect languages:

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

These languages perform functions similar to the MINERS QoS language described above.

QuO has in the past emphasized *reactive* resource allocation [32], 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.

**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 requests 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 are unduly conservative in rejecting QoS requests.
- *Requires source code*. This pattern require access to the source code of the application and/or infrastructure implementation in order to instrument it.

**See Also**

The Quality Connector pattern is related to the Component Configurator [26] and the Virtual Component [37] patterns, which permit component implementations to be linked into and unlinked from a running application without shutting down the application. Both the Quality Connector pattern and these other two patterns provide a means to change the behavior of a service during application execution. The Component Configurator and Virtual Component patterns are 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 strategies.

The goal of the Quality Connector pattern is similar to that of the Interceptor pattern [26], 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 [30] provides for the inclusion in a running application of meta-objects that provide information about, and control over, the 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 [29] 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.

## **References**

[0] Richard E. Schantz and Douglas C. Schmidt, "Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications," Encyclopedia of Software Engineering, Wiley and Sons, 2002.

[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 J. Russell Noseworthy, "Patterns and Performance of a CORBA Event Service for Large-scale Distributed Interactive Simulations," International Journal of Computer Systems Science and Engineering, CRL Publishing, 2001.

[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 2.5. http://www.omg.org/cgi-bin/doc?formal/01-09-61

[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 Controlling 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 Comp uter 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] Douglas C. Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann, *Pattern-Oriented Software Architecture,: Patterns for Concurrent and Networked Objects*, Wiley and Sons, 2000.

[27] The Programmable Composition of Embedded Software (PCES) Project, DARPA Information Technology Office. http://www.darpa.mil/ito/research/pces/index.html

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

[29] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[30] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, Wiley and Sons, 1996.

[31] Timothy H. Harrison and David L. Levine and Douglas C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service," Proceedings of OOPSLA '97, Atlanta, GA, October 1997.

[32] Joseph K. Cross and Patrick J. Lardieri, *Proactive and Reactive Resource Reallocation in DoD DRE Systems*, submitted to the 9th Annual Conference on Pattern Languages of Programs focused topic session "Towards Patterns and Pattern Languages," Monticello, IL, September, 2002.

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

[34] Greg Bollella and James Gosling, "The Real-Time Specification for Java", IEEE *Computer*, June, 2000, pp. 47-54.

[35] Nanbor Wang, Douglas C. Schmidt, Ossama Othman, and Kirthika Parameswaran, "Evaluating Meta-Programming Mechanisms for ORB Middleware," IEEE Communication Magazine, special issue on Evolving Communications Software: Techniques and Technologies, 2001.

[36] Bill Gallmeister, *POSIX.4 Programming for the Real World*, O'Reilly, 1995.

[37] Carlos O'Ryan and Angelo Corsaro and Raymond Klefstad and Douglas C. Schmidt, "Virtual Component: : a Design Pattern for Memory-Constrained Systems," submitted to the 9th Annual Conference on the Pattern Languages of Programs", focused topic session "Towards Patterns and Pattern Languages," Monticello, Illinois, September 2002.