Flexible and Adaptive Control of Real-Time Distributed Object Computing Middleware

Joseph P. Loyall (jloyall@bbn.com), Alia K. Atlas (akatlas@bbn.com) and Richard E. Schantz (schantz@bbn.com)
BBN Technologies

Christopher D. Gill (cdgill@cs.wustl.edu), David L. Levine (levine@cs.wustl.edu), Carlos O'Ryan (coryan@cs.wustl.edu) and Douglas C. Schmidt (schmidt@cs.wustl.edu)
Washington University, St. Louis

Abstract.

Next-generation distributed systems have growing demands for real-time quality of service (QoS), flexibility, and control over the often unpredictable environments in which they are deployed. These demands have been hard to achieve simultaneously. For instance, systems have historically either been real-time, which meant that they were highly-tuned, special-purpose, and fragile, or they were flexible, thereby incurring performance penalties that made it hard to achieve stringent QoS requirements. Achieving both sets of demands simultaneously requires distributed object computing (DOC) middleware that supports dynamic and layered resource management, automated reconfiguration, dynamic scheduling, and application-level interfaces for control and adaptation.

This paper provides three contributions to the study of adaptive real-time middleware for distributed and embedded systems. First, it describes advances in Object Request Broker (ORB) technology that can support stringent real-time requirements. Second, it describes extensions to the DOC model that support the control and measurement of, and adaptation to, changing QoS requirements and conditions. Finally, the paper describes how combining multiple cross-cutting DOC middleware strategies for real-time scheduling, resource management, flexible control, and adaptation can achieve real-time QoS *and* flexibility simultaneously.

Keywords: Adaptive Middleware, Dynamic Scheduling, Real-Time Middleware, Quality of Service

1. Introduction

Emerging trends: Next-generation highly networked and interconnected distributed and embedded real-time systems must collaborate with multiple remote sensors, provide on-demand browsing and actuation capabilities for human operators, and respond flexibly to unanticipated situational factors that arise at run-time (Doerr et al., 1999). Moreover, these systems must perform unobtrusively, shielding operators from unnecessary details, while simultaneously communicating and responding to mission-critical information at an accelerated operational tempo (Levine et al., 1998). In such environments, it is often hard or impossible to predict *a priori*, or even approximate into the near future, system configurations or workload mixes.

The communication infrastructure for these next-generation systems must be sufficiently flexible to support varying workloads at different times during an application lifecycle, yet maintain highly predictable behavior. *Quality of service* (QoS) is a widely accepted term that describes a loosely organized collection of activities and technology initiatives designed to improve and control communication-oriented resource management based on mounting R&D experience with distributed applications and systems (Christopher D. Gill et al., 1999). Controlling the real-time behavior of such distributed computing systems is one important dimension of the delivered quality of service.

Providing effective QoS has always influenced the usability of applications and systems. It is only recently, however, that QoS as a set of *named aspects* (Kiczales, 1997) has emerged as a user-controllable property of distributed system infrastructure. Providing greater user control over QoS is in contrast to the level of service that has been traditionally fixed – identically and uncontrollably for all users and application use-cases – during infrastructure conception.

The recent focus on user control over QoS aspects stems from technology advances in historically challenging research areas, such as allocation policies, synchronization of streams (Steinmetz, 1990) in distributed multimedia applications, and assured communication in the face of high demand. The focus on QoS aspects has led the computing and communication research community to devise a number of proposed and implemented improvements to commonly available distributed computing infrastructures. When coupled with software that can recognize and react to environmental changes, these improvements form the basis for constructing appropriate adaptive behavior for next-generation distributed and embedded systems.

An overview of COTS middleware: Constructing usable distributed and real-time systems has always been challenging. Requirements for faster development cycles, decreased cost, and reusable solutions motivate the use of *middleware*. Middleware is software that resides between applications and the underlying operating systems, protocol stacks, and hardware to enable or simplify how these components are connected and interoperate (Christopher D. Gill et al., 1999). Figure 1 illustrates the general concept of middleware and some of its key layers used for this paper. We can further decompose the middleware layer into two general categories of middleware:

- *Infrastructure middleware:* This layer encapsulates lower-level operating system communication and concurrency mechanisms to provide a higher level DOC programming model that automates common network programming tasks, such as parameter marshaling/demarshaling,

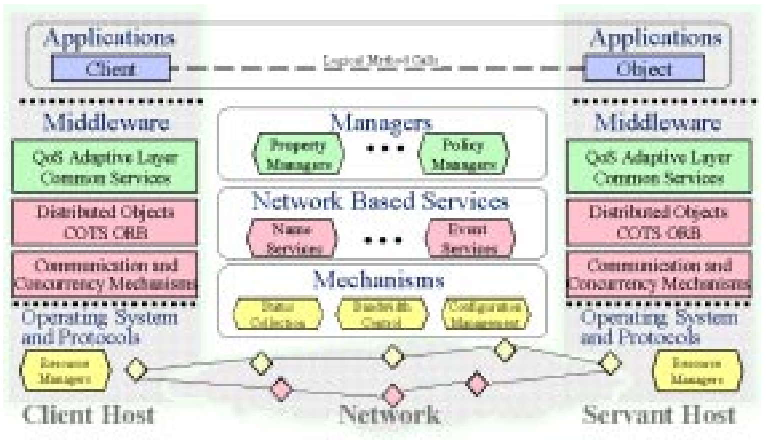


Figure 1. Layers of Middleware

request demultiplexing, and error handling. At the heart of this infrastructure middleware resides some form of Object Request Broker (ORB), such as CORBA (Object Management Group, 1999c) and Real-time CORBA (Object Management Group, 1999b), Java RMI (Wollrath et al., 1996), and Microsoft's DCOM (Box, 1997).

- *Common middleware service components:* Infrastructure middleware itself forms an enabling framework for yet a higher level of common middleware service components. These components provide domain-independent services that can be reused by many applications to manage common distributed system tasks. Example services (Object Management Group, 1995) include persistence (Object Management Group, 1999a), security (Object Management Group, 1998b), transactions (Object Management Group, 1997b), fault tolerance (Object Management Group, 1998a), and concurrency (Object Management Group, 1997a).

In general, middleware provides the following benefits: (1) it shields software developers from low-level, tedious, and error-prone details, such as socket-level programming (Schmidt et al., 1995), (2) it provides a consistent set of higher-level abstractions (Gill et al., 2000; Zinky et al., 1997) for developing distributed systems, (3) it amortizes software lifecycle costs by leveraging previous development expertise and capturing implementations of key design patterns (Schmidt and Cleeland, 1999) in reusable frameworks, rather than building them entirely from scratch for each use.

When middleware is commonly available for acquisition or purchase, it becomes “commercial-off-the-shelf” (COTS). While it is possible *in theory* to develop complex systems from scratch, *i.e.*, without using COTS middleware, contemporary economic and organizational constraints, as well as competitive pressures, are making it implausible to do so *in practice*. Thus, COTS middleware plays an increasingly strategic role in software intensive, real-time distributed systems, which is why we base our adaptive real-time R&D activities on COTS middleware.

Towards an adaptive COTS middleware solution: In addition to the development methodology and system lifecycle constraints outlined above, designers of real-time systems have historically used relatively static methods to allocate scarce or shared resources to system components. For instance, flight-qualified avionics mission computing systems (Harrison et al., 1997) establish priorities for all resource allocation and scheduling decisions very early in the system lifecycle, *i.e.*, *well* before run-time. Static strategies have traditionally been used for real-time applications because

- System resources were insufficient for more computationally-intensive dynamic on-line approaches.
- Simplifying analysis and validation was essential to remain on budget and on schedule, particularly when systems were designed from scratch using low-level, proprietary tools.

Both these factors are changing rapidly. For instance, additional computing and networking resources are becoming available (though statically configured legacy software is often not able to use these new resources effectively). Moreover, complex systems are rarely built from scratch anymore. Thus, although validation and analysis of these systems remains hard, it become more tractable by using pre-analyzable common COTS infrastructure components, rather than using custom proprietary components built in-house.

As network and endsystem performance continues to increase, so too does the demand for more control and manageability of their resources through the middleware interface. In particular, next-generation systems present real-time QoS requirements for shared resources and workloads that can vary significantly at run-time. In turn, this increases the demands on end-to-end system resource management and control, which makes it hard to simultaneously (1) create effective resource managers using traditional statically constrained allocators and schedulers, (2) achieve reasonable resource utilization, and (3) meet individual application tradeoffs and preferences. In addition, the mission-critical processing aspects of next-generation systems require that they (1) respond adequately to both anticipated and unanticipated operational changes in their run-time environment and (2) ensure that critical capabilities acquire the necessary resources.

Meeting the increasing demands of next-generation real-time systems motivates the need for additional adaptive middleware-centric abstractions and techniques. Supporting this adaptive middleware architecture efficiently, predictably, and scalably requires new dynamic and adaptive resource management techniques that extend existing static resource management techniques in areas such as automated reconfiguration, layered resource management, and dynamic scheduling. These techniques are currently being explored in the context of various research activities, in particular the DARPA Quorum program (DARPA, 1999), which is researching solutions to a number of the missing capabilities needed for mission-critical system development, such as predictable performance for network based applications, fault tolerance and dependability characteristics, real-time performance properties, and fine grained distributed systems security.

In this paper, we describe recent advances we have made towards developing adaptive real-time systems within the context of standards-based COTS middleware. We have created advanced, reusable, multi-level middleware that enables a new generation of flexible distributed applications to (1) have greater control over their resource management strategies and (2) be easily reconfigured and adapted dynamically to changing network and computing environments. Our results come through the integration of two complementary perspectives and technologies: a “top down” adaptable policy-driven perspective coupled with a “bottom up” real-time mechanism-driven perspective.

Paper organization: The remainder of this paper is structured as follows: Section 2 describes the individual technologies underlying our work on adaptive real-time COTS middleware; Section 3 then discusses the topics, approach taken, current results, and open research issues related to our integration activities within the DARPA Quorum (DARPA, 1999) program; Section 4 summarizes ongoing work in the field related to our current research; and Section 5 presents concluding remarks.

2. DOC Middleware Technology Overview

Distributed object computing (DOC) is the most advanced, mature, flexible paradigm available today in which tackle the development of next-generation distributed and embedded systems (Henning and Vinoski, 1999) in domains such as national security, military, health care, medical, multimedia, and financial systems. DOC software architectures are composed of relative autonomous objects that can be distributed or collocated throughout a wide-range of networks and interconnects. Client objects invoke operations on target objects to perform interactions and functionality needed to achieve application goals.

As outlined in Section 1, DOC middleware exposes only the functional interfaces of application component and services, thereby shielding applications from many distributed computing complexities. Chief among these complexities include remote location interoperability, heterogeneity, common services, and synchronization. Mission-critical next-generation distributed applications have stringent quality of service (QoS) requirements, however, such as real-time performance, security, and dependability, that require them to react to or control *how* services are delivered, not just *what* services are delivered.

Conventional middleware, based DOC and other programming paradigms, fails to support more stringent end-to-end application requirements because it hides the details necessary to specify, measure, and control QoS. Moreover, it does not support the development of systems that can adapt to changing QoS conditions dynamically. As a result, developers of mission-critical distributed applications must often “program around” the DOC middleware, which provides little or no advantage compared with building systems manually from scratch. These problems are exacerbated when applications are distributed over WANs, which are inherently more dynamic, unpredictable, and unreliable than LANs and real-time bus interconnects.

This section describes extensions we have made to conventional DOC middleware programming models and implementations so they can support real-time QoS properties and simultaneously allow flexible control and adaptation of key application QoS aspects. We have been developing extensions these in the context of the COTS CORBA DOC middleware model (Object Management Group, 1999c), which we summarize first. Next, we summarize the key features and characteristics of TAO and QuO, which leverage CORBA to provide efficient, scalable, and predictable real-time middleware mechanisms and adaptive QoS management policies, respectively.

2.1. Overview of CORBA

CORBA Object Request Brokers (ORBs) allow clients to invoke operations on distributed objects without concern for object location, programming language, OS platform, communication protocols and interconnects, and hardware (Henning and Vinoski, 1999). Figure 2 illustrates the key components in the CORBA reference model (Object Management Group, 1999c) that collaborate to provide this degree of portability, interoperability, and transparency.[1] Each component in the CORBA reference model is outlined below:

Client: A client is a *role* that obtains references to objects and invokes operations on them to perform application tasks. Objects can be remote or collocated relative to the client. Ideally, a client can access a remote object just like a local object, *i.e.*, `object→operation(args)`. Figure 2 shows how the underlying ORB components described below transmit remote operation requests transparently from client to object.

Object: In CORBA, an object is an instance of an OMG Interface Definition Language (IDL) interface. Each object is identified by an *object*

[1] This overview only focuses on the CORBA components relevant to this paper. For a complete synopsis of CORBA’s components see (Object Management Group, 1999c).

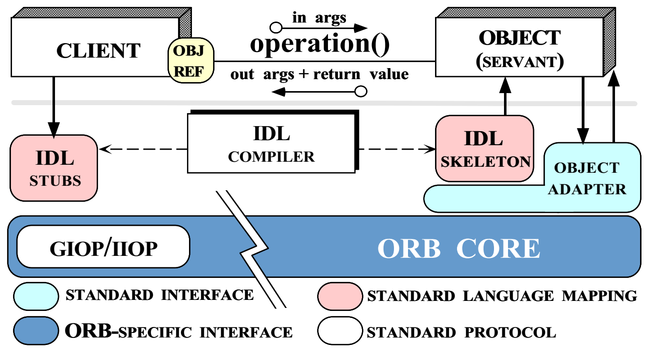


Figure 2. Key Components in the CORBA 2.x Reference Model

reference, which associates one or more paths through which a client can access an object on a server. An *object ID* associates an object with its implementation, called a servant, and is unique within the scope of an Object Adapter. Over its lifetime, an object has one or more servants associated with it that implement its interface.

Servant: This component implements the operations defined by an OMG IDL interface. In object-oriented (OO) languages, such as C++ and Java, servants are implemented using one or more class instances. In non-OO languages, such as C, servants are typically implemented using functions and `structs`. A client never interacts with servants directly, but always through objects identified by object references.

ORB Core: When a client invokes an operation on an object, the ORB Core is responsible for delivering the request to the object and returning a response, if any, to the client. An ORB Core is implemented as a run-time library linked into client and server applications. For objects executing remotely, a CORBA-compliant ORB Core communicates via a version of the General Inter-ORB Protocol (GIOP), such as the Internet Inter-ORB Protocol (IIOP) that runs atop the TCP transport protocol. In addition, custom Environment-Specific Inter-ORB protocols (ESIOPs) can also be defined.

OMG IDL Stubs and Skeletons: IDL stubs and skeletons serve as a "glue" between the client and servants, respectively, and the ORB. Stubs implement the *Proxy* pattern (Gamma et al., 1995) and provide a strongly-typed, *static invocation interface* (SII) that marshals application parameters into a common message-level representation. Conversely, skeletons implement the *Adapter* pattern (Gamma et al., 1995) and demarshal the message-level representation back into typed parameters that are meaningful to an application.

IDL Compiler: An IDL compiler transforms OMG IDL definitions into stubs and skeletons that are generated automatically in an application programming language, such as C++ or Java. In addition to providing programming language transparency, IDL compilers eliminate common sources of network programming errors and provide opportunities for automated compiler optimizations (Eide et al., 1997).

Object Adapter: An Object Adapter is a composite component that associates servants with objects, creates object references, demultiplexes incoming requests to servants, and collaborates with the IDL skeleton to dispatch the appropriate operation upcall on a servant. Object Adapters enable ORBs to support various types of servants that possess similar requirements. This design results in a smaller and simpler ORB that can support a wide range of object granularities, lifetimes, policies, implementation styles, and other properties.

2.2. Overview of the TAO Real-time ORB and Event Service

TAO is a high-performance, real-time ORB endsystem targeted for applications with deterministic QoS requirements, as well as best-effort requirements. The TAO ORB endsystem contains the network interface, OS, communication protocol, and CORBA-compliant middleware components and services shown in Figure 4.

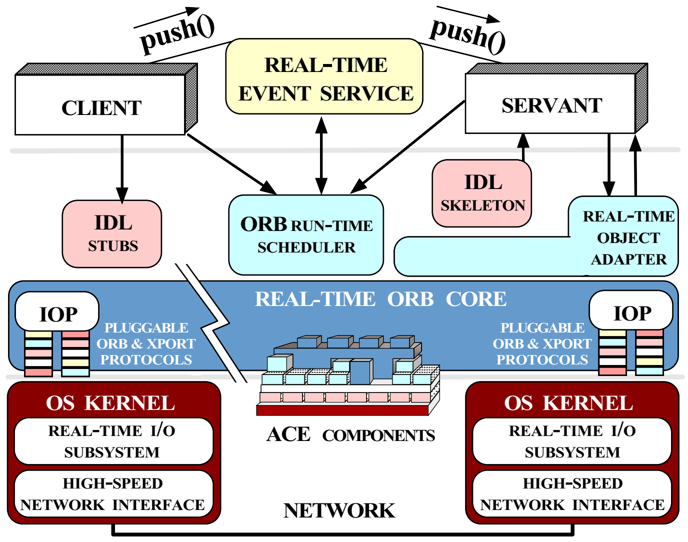


Figure 4. Components in the TAO Real-time ORB Endsystem

TAO supports the standard OMG CORBA reference model (Object Management Group, 1999c) and Real-time CORBA specification (Object Management Group, 1999b), with enhancements designed to ensure efficient, predictable, and scalable QoS behavior for high-performance and real-time applications.

Optimized IDL Stubs and Skeletons: IDL stubs and skeletons perform marshaling and demarshaling of application operation parameters, respectively. TAO's IDL compiler generates stubs/skeletons that can selectively use highly optimized compiled and/or interpretive marshaling/demarshaling (Gokhale and Schmidt, 1999). This flexibility allows application developers to selectively trade off time and space, which is crucial for high-performance, real-time, and/or embedded distributed systems.

Real-time Object Adapter: An Object Adapter associates servants with the ORB and demultiplexes incoming requests to servants. TAO's real-time Object Adapter (Pyarali et al., 1999) uses perfect hashing (Schmidt, 1990) and active demultiplexing (Pyarali et al., 1999) optimizations to dispatch servant operations in constant $O(1)$ time, regardless of the number of active connections, servants, and operations defined in IDL interfaces.

Run-time Scheduler: TAO's run-time scheduler (Object Management Group, 1999b) maps application QoS requirements, such as bounding end-to-end latency and meeting periodic scheduling deadlines, to ORB endsystem/network resources, such as CPU, memory, network connections, and storage devices. TAO's run-time scheduler supports both static (Schmidt et al., 1998) and dynamic (Gill et al., 2000) real-time scheduling strategies.

Real-time Event Service: TAO's Real-time (RT) Event Service (Harrison et al., 1997) extends the standard CORBA Event Service (Object Management Group, 1995) by providing (1) source and type-based filtering, (2) event correlations, (3) event channel federations, (4) hardware and kernel-level filtering based on IP multicast, and (5) large numbers of suppliers and consumers. In addition, TAO's RT Event Service can be integrated with TAO's run-time scheduler outlined above to support applications with stringent end-to-end real-time requirements.

Real-time ORB Core: An ORB Core delivers client requests to the Object Adapter and returns responses (if any) to clients. TAO's real-time ORB Core (Schmidt et al., 2000) uses a multi-threaded, preemptive, priority-based connection and concurrency architecture (Gokhale and Schmidt, 1999) to provide an efficient and predictable CORBA protocol engine. TAO's ORB Core allows customized protocols to be plugged into the ORB without affecting the standard CORBA application programming model.

Real-time I/O subsystem: TAO's real-time I/O (RIO) subsystem (Kuhns et al., 1999b) extends support for CORBA into the OS. RIO assigns priorities to real-time I/O threads so that the schedulability of application components and ORB endsystem resources can be enforced. When integrated with advanced hardware, such as the high-speed network interfaces described below, RIO can (1) perform early demultiplexing of I/O events onto prioritized kernel threads to avoid thread-based priority inversion and (2) maintain distinct priority streams to avoid packet-based priority inversion. TAO also runs efficiently and relatively predictably on conventional I/O subsystems that lack advanced QoS features.

High-speed network interface: At the core of TAO's I/O subsystem is a "daisy-chained" network interface consisting of one or more ATM Port Interconnect Controller (APIC) chips (Dittia et al., 1997). The APIC is designed to sustain an aggregate bi-directional data rate of 2.4 Gbps using zero-copy buffering optimization to avoid data copying across endsystem layers. In addition, TAO runs on conventional real-time interconnects, such as VME backplanes and multi-processor shared memory environments.

TAO internals: TAO is developed using lower-level middleware called ACE (Schmidt and Suda, 1994), which implements core concurrency and distribution patterns (Gamma et al., 1995) for communication software. ACE provides reusable C++ wrapper facades and framework components that support the QoS requirements of high-performance, real-time applications and higher-level middleware like TAO. ACE and TAO run on a wide range of OS platforms, including Win32, most versions of UNIX, and real-time operating systems like Sun/Chorus ClassiX, LynxOS, and VxWorks.

2.3. Overview of QuO

Quality Objects (QuO) is a distributed object computing (DOC) framework designed to develop distributed applications that can specify (1) their QoS requirements, (2) the system elements that must be monitored and controlled to measure and provide QoS, and (3) the behavior for adapting to QoS variations that occur at run-time. By providing these features, QuO opens up distributed object implementations (Kiczales, 1996) to control an applications functional aspects and implementation strategies that are encapsulated within its functional interfaces. To achieve these goals, QuO provides middleware-centric policies and mechanisms for developing DOC applications that can perform the following operations *in addition to* their functional behavior:

Specify operating regions and service requirements: QuO-enabled applications can specify their levels of desired performance or resources (which might change dynamically based upon changes in the environment), operating modes, and operating regions. changes in the environment), operating modes (corresponding to different functional objectives of the application), and operating regions (corresponding to different environment or system conditions, resource availability, etc.).

Measure environmental and system conditions: QuO-enabled applications can insert and use probes in their distributed environment to measure resources, characteristics, and behavior. In addition, these applications can receive information from resource managers, real-time operating systems or ORBs, and other property managers and mechanisms.

Access control interfaces: QuO-enabled applications can access system resource management control interfaces and pass information to resource or property managers to achieve their desired level of service.

Adapt and reconfigure: QuO-enabled applications and systems can adapt to changing conditions at all levels, coordinated through the QuO middleware. For example, in response to changing mission objectives or degraded resources, a QuO-enabled system can respond through adaptation on the part of the resource managers, real-time mechanisms, QuO middleware, and application programs.

The functional path of QuO illustrated in Figure 5 is a superset of the functional path of CORBA illustrated in Figure 2. The components provided by QuO to support the above operations are defined below.

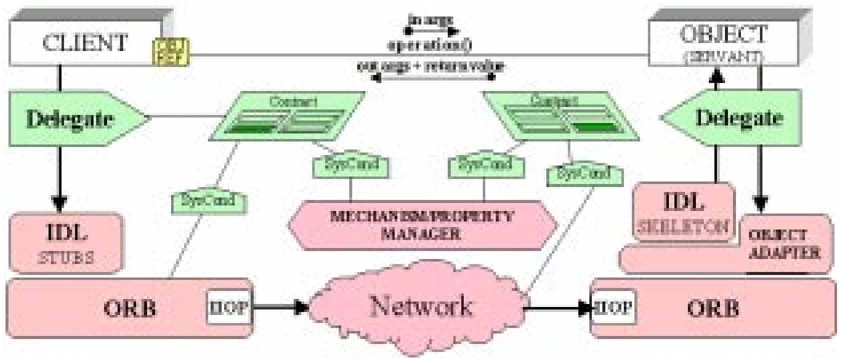


Figure 5. The QuO Distributed Object Computing Model

Contracts: The operating regions and service requirements of the application are encoded in *contracts*, which describe the possible states the system might be in, as well as which actions to perform when the state changes. The possible states are specified as a set of *regions*, which can be nested. Each region is defined by a predicate over a set of system condition objects, which are described below. A contract also defines a set of *transitions* between regions, which specify adaptive behavior that is triggered when the system state (as defined by the contract regions and predicates) changes (as represented by a contract transitioning from one valid region to another).

Delegates: QuO inserts *delegates* in the CORBA functional path. Delegates project the same interfaces as the stub (client-side delegate) and the skeleton (server-side delegate), but support adaptive behavior upon method call and return. When a method call or return is made, the delegate checks the system state, as recorded by a set of contracts, and selects a behavior based upon it.

Contracts and delegates support two means for triggering manager-level, middleware-level, and application-level adaptation. The delegate triggers *in-band* adaptation by making choices upon method calls and returns. The contract triggers *out-of-band* adaptation when region transitions occur which can be caused by changes in observed system condition objects.

System Condition Objects: These objects provide uniform interfaces to multiple levels of system resources, mechanisms, and managers to translate between application-level concepts, such as operating modes, to resource and mechanism-level concepts, such as scheduling methods and real-time attributes. System condition objects are used to measure the states of system resources, mechanisms, and managers that are relevant to contracts in the overall system. In addition, they can pass information to interfaces that control the levels of desired services.

Higher-level system condition objects can interface to other, lower-level system condition objects, forming a tree of system condition objects that translate mechanism data into application data. System condition objects can be either *observed* or *non-observed*. Changes in the values measured by observed system conditions trigger contract evaluation, possibly resulting in region transitions and triggering adaptive behavior.

Observed system condition objects are suitable for measuring conditions that either change infrequently or for whom a measured change can indicate an event of notice to the application or system. Non-observed system condition objects represent the current value of whatever condition they are measuring, but do not trigger an event whenever the value changes. Instead, they provide the value upon demand, whenever the contract needs it, *i.e.*, whenever the contract is evaluated due to a method call or return or due to an event from an observed system condition object.

Observed system condition objects can measure frequently changing system conditions by coding the system condition object to *smooth out* continuous changes. For example, a system condition object measuring the load on a host can be observed. However, it can be programmed to only report periodic events showing average load over some time duration. Likewise, it can be programmed to only report events when the load crosses a certain threshold.

Instrumentation Probes: QuO provides a library of *instrumentation probes* that can be inserted throughout the remote method invocation path. These probes can be used by the QuO infrastructure to gather performance statistics and validation information unobtrusively. To accomplish this, the QuO delegate adds a data structure to each method call and return. This structure can be populated or read by any or all the instrumentation probes along the method call/return path.

Quality Description Languages (QDLs) and Code Generators: QuO provides a suite of QDLs, which are similar to CORBA's Interface Description Language (IDL), and *code generators*, which are similar to the stub and skeleton generators of CORBA IDL compilers. QDLs and code generators describe and automatically output, respectively, the components of QuO applications (Loyall et al., 1998b; Loyall et al., 1998a; Pal et al., 2000). QuO currently provides a contract description language (CDL); a structure description language (SDL) to specify adaptive behavior and adaptation strategies; and a connector setup language (CSL) to specify the components of a QuO application and how they are instantiated, connected, and initialized.

QuO Runtime Kernel and GUI Monitor: QuO provides a *runtime kernel* that coordinates contract evaluation and provides other runtime QuO services (Vanegas et al., 1998). These services include initializing contracts and system conditions, binding them to each other and to delegates, triggering contract evaluation, and triggering adaptive behavior. The runtime kernel also provides a debugging mode that prints trace messages of the QuO middleware behavior during system execution.

In addition, the QuO kernel provides a graphical user interface (GUI) that enables monitoring applications to observe the QuO middleware in action. The GUI displays contracts and regions and indicates the current active region and the previously active regions. It also displays the system condition objects in the system and their values, indicating when region transitions occur and the adaptive behavior triggered by the transition. Finally, it displays statistics showing how much time applications have spent in each contract region.

QuO Gateway: QuO provides a general object gateway component, illustrated in Figure 6, which allows low-level communication mechanisms

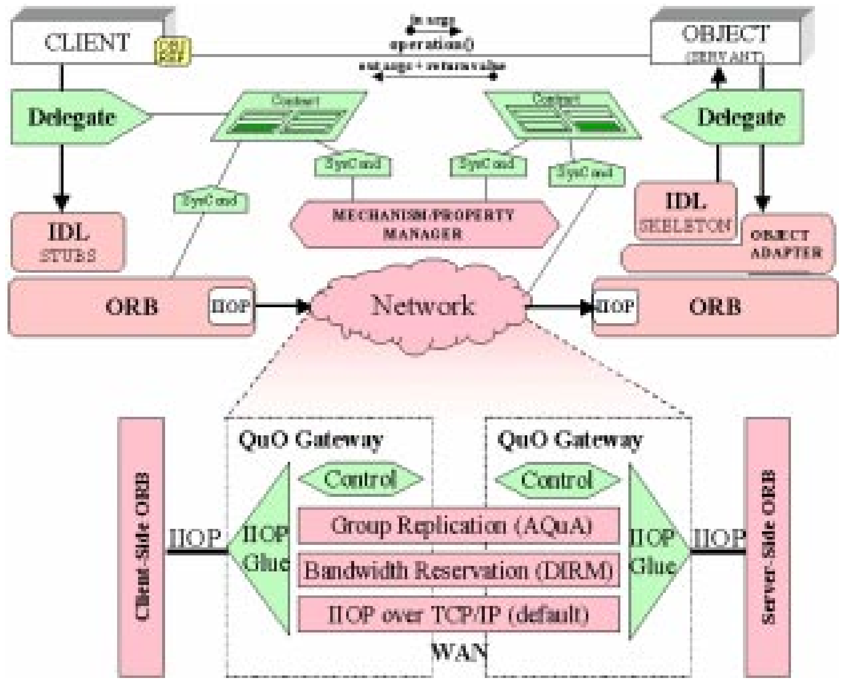


Figure 6. The QuO gateway

and special-purpose to be *plugged into* an application (Schantz et al., 1999). The QuO gateway resides between the client and server ORBs. It is a mediator (Gamma et al., 1995) that intercepts IIOP messages sent from the client-side ORB and delivers IIOP messages to the server-side ORB (on the message return the roles are reversed). On the way, the gateway translates the IIOP messages into a custom transport protocol, such as group multicast in a replicated, dependable system. The QuO gateway is implemented using TAO's pluggable protocol feature (Kuhns et al., 1999a).

The gateway also provides an API that allows adaptive behavior or processing control to be configured below the ORB layer. For example, the gateway can select between alternate transport mechanisms based on low-level message filtering or shaping, as well as the overall system's state and condition objects. Likewise, the gateway can be used to integrate security measures, such as authenticating the sender and verifying access rights to the destination object.

3. QuO Middleware Specification and Control of TAO Real-Time Mechanisms

TAO's CORBA ORB middleware provides a rich set of mechanisms for representing and enforcing real-time requirements in DOC applications. Directly programming TAO's lower-level real-time mechanisms to achieve specific end-to-end quality of service (QoS) goals can be excessively tedious and error-prone, however, particularly for large-scale mission-critical distributed systems. Therefore, higher-level middleware capabilities for end-to-end QoS specification and control are needed.

QuO offers the following two facilities for higher level specification and control of TAO's real-time mechanisms:

1. QuO provides additional mechanisms for application-level and middleware-level adaptation that complements and improves lower-level real-time capabilities of ORB middleware.
2. QuO allows developers to specify higher-level aspects of real-time requirements, such as the type of real-time required – *e.g.*, periodic or end-to-end, the relative priority of events, and the tradeoffs between real-time and other QoS requirements. It then maps these higher-level specifications into QuO and TAO mechanisms that implement, measure, and control them.

The remainder of this section examines various types of application requirements for real-time QoS and describes mechanisms for specifying and enforcing QoS within the TAO and QuO middleware. In addition, we present a sample application that illustrates the power of our integrated approach for specifying and enforcing end-to-end real-time QoS in mission-critical distributed and embedded systems.

3.1. Synopsis of Real-Time Application QoS Requirements

It is important to examine a distributed application's real-time QoS requirements from the perspective of the application itself, *i.e.*, in isolation from various policies and mechanisms that are used to meet those requirements. First, doing so offers insight into the general characteristics of the application, thereby promoting portability and implementation flexibility by decoupling application behavior from specific QoS management implementations. Second, mapping these abstract requirements back into specific implementations offers insight into the general capabilities of various QoS management policies and mechanisms. Finally, the patterns that emerge from comparing application requirements and QoS management capabilities at an appropriate level of abstraction provide a basis for evaluating the suitability of a particular QoS management implementation for a given application.

Below, we outline key requirements for distributed and embedded real-time applications. These requirements present different aspects of the single real-time QoS aspect: *timeliness constraints on applications*. These timeliness constraints can manifest themselves as absolute time requirements or as requirements for synchronization or coordination with other components. For example, a distributed collaboration whiteboard application may require that local display modifications propagate and appear on all other remote displays within 400 milliseconds to avoid excessive display jitter.

The list below is not intended to be complete for all applications. Instead, it motivates and illustrates the generality and flexibility of the adaptive middleware QoS architecture presented in subsequent discussions.

Periodic invocation: Certain distributed real-time applications have specialized timing requirements where specific operations must begin and/or complete within repeated periodic intervals. For example, aircraft sensor devices, such as global position system (GPS) and radar sensors, generate data at regular periodic intervals (Harrison et al., 1997).

Aperiodic invocation: Unlike periodic requests, aperiodic invocation requests are not generated at regular intervals. Instead they are generated in response to specific events, such as threats or alerts. In many cases, aperiodic events must take priority over periodic events due to special circumstances they represent. For example, it may be necessary to respond at high priority to the signal on a fighter aircraft that warns of an enemy radar lock.

Event delivery and consumption: Some distributed applications have real-time requirements that are best defined in terms of *end-to-end* event delivery and consumption. For example, the periodic generation of sensor data described above may generate interrupts to notify various applications, such as heads-up displays (HUDs) or navigation systems, to receive incoming event data.

Event-driven models are used by many networked, embedded systems, including avionics systems, command and control systems, and industrial process control systems. Such applications often have requirements whereby not only must events be delivered to the proper consumers, but consumers must process the events completely within strict time bounds. In a periodic invocation system, this is typically prior to the next arrival of a periodic event. In aperiodic systems, this is often based upon external constraints. For example, nuclear power plant safety applications detect dangerous situations and also initiate automated safeguards to alleviate them. Both the initiation and completion of the safeguard processing must complete within a fixed interval from the detection of the hazardous situation.

Round-trip: Some applications require data to be delivered, processed, and a reply returned within a given time-frame. For example, a stock market trading application must place trades, conduct the trading transaction, and return a confirmation, all within a fixed interval from the user's initiation of a trade.

Pipelined: Pipelined applications require that data and events be delivered and passed to a final destination within specific time constraints. In some cases, these applications, whose timing requirements often pertain only to the final delivery, can be divided into subsidiary real-time constraints pertaining to each segment of the distributed pipeline.

Different policies and mechanisms within middleware architectures can be applied to specify, enforce, and control these real-time aspects and requirements. TAO provides a number of mechanisms for enforcing the above requirements, while QuO provides higher-level capabilities for specifying and controlling these requirements. Together, TAO and QuO help applications adapt to changing requirements and changing environments that affect their ability to meet end-to-end QoS requirements.

3.2. TAO Enforcement Mechanisms

TAO and its real-time (RT) Event Service provides many mechanisms for enforcing QoS requirements related to managing CPU, memory, and network resources (Harrison et al., 1997; O'Ryan et al., 1999). These mechanisms can be categorized by the time-scales on which they provide adaptation to meet application QoS requirements, as follows:

In-band mechanisms: These mechanisms perform shorter time-scale activities to establish and preserve QoS during system operation. TAO's in-band mechanisms for real-time QoS include periodic timer expiration, event correlation, event filtering, priority dispatching, and dynamic queue ordering.

Out-of-band mechanisms: These mechanisms perform longer time-scale mechanisms that configure or reconfigure in-band mechanisms in the ORB and its ORB Services. TAO's out-of-band mechanisms for real-time QoS include scheduling strategies, static priority assignment, static queue configuration, event supplier registration, and event consumer registration.

Below, we examine these mechanisms in detail.

Periodic timer expiration: TAO's RT Event Service allows event consumers to register for periodic timer expiration events. This feature can enforce periodic invocation of operations with periodic timing requirements, *e.g.*, polling values of sensor devices.

Event correlation: TAO's RT Event Service allows event consumers to control their real-time invocation semantics by correlating events using logical conjunction and disjunction. For example, an event consumer might wish to be invoked whenever an event supplier sends a "data ready" event *or* whenever an "end-of-frame" timer expires.

Event filtering: TAO's RT Event Service allows applications to reduce resource demands by filtering out unnecessary events, thereby reducing event traffic. For example, one event consumer may register to receive events only from a particular event supplier. Conversely, another event consumer may register to see only events of a particular type, but from any supplier.

Priority dispatching: Priority dispatching can be implemented efficiently on most modern operating systems through the OS kernel's preemptive thread scheduler. TAO implements the standard Real-time CORBA (Object Management Group, 1999b) policies that allow applications to assign distinct priority to threads in a pool (Schmidt et al., 2000) and to associate each thread with a queue onto which requests can be inserted. Using these mechanisms, operations can be demultiplexed and dispatched at appropriate thread priorities.

Dynamic queue ordering: Different queue ordering policies are useful for different types of real-time dispatching (Gill et al., 2000). For example, operations can be ordered according to their advertised deadlines, also known as earliest deadline first (EDF) (Liu and Layland, 1973) ordering. Moreover, if an application knows the best and/or worst case execution times of its operations, it can use an minimum laxity first (MLF) (Stewart and Khosla, 1992) ordering to ensure the operation with the least "slack" time is scheduled first. Ordering by statically assigned subpriorities is useful to break ties among operations with the same preemptive priority level but having precedence relationships, *e.g.*, due to data dependencies. Finally, if an application does not require other forms of queue ordering, simple FIFO queueing is efficient and straightforward to implement.

Scheduling strategies: TAO's Scheduling Service supports different strategies for static priority assignment and static dispatching queue configuration. Implementations of the well-known RMS (Liu and Layland, 1973), EDF (Liu and Layland, 1973), MLF (Stewart and Khosla, 1992), and MUF (Stewart and Khosla, 1992) scheduling strategies are provided with TAO. The choice of scheduling strategy has significant implications for the real-time behavior of an application. Therefore, scheduling strategies are typically assigned during system design and rarely vary during system execution.

Certain scheduling strategies are better suited for particular types of applications, however, which motivates the ability to configure different strategies flexibly. For example, an RMS strategy may be serve a periodic sensor polling application, where all operations are time-critical and are known in advance. Conversely, an EDF strategy may be more suited to a peer-to-peer gateway application with no bounds on its traffic load.

Static priority assignment: Scheduling strategies that assign distinct static priorities to all operations can help to isolate higher priority operations from the resource demands of lower priority operations. In turn, this allows the application of real-time analysis techniques, such as RMA (Klein et al., 1993), that depend on fixed priorities for all operations. Likewise, other scheduling strategies, such as maximum urgency first (MUF) (Stewart and Khosla, 1992), that assign static priority according to operation criticality can help to isolate mission-critical portions of the system from the resource requirements of non-critical portions. Note that these “static” priorities can be adaptively reconfigured at run-time using TAO’s reconfigurable scheduler implementation (Doerr et al., 1999). However, such reconfiguration occurs on an out-of-band time-scale, reflecting a system mode change (Sha et al., 1989).

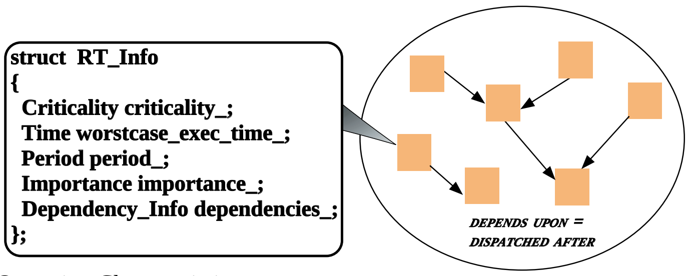
Static queue configuration: Supporting diverse scheduling strategies requires significant *in-band* variation in the run-time enforcement of application QoS. Despite this, the *configurations* of the enforcement mechanisms can be established statically. In particular, the number of dispatching queues, and their policies for ordering operation dispatches, are functions of the scheduling strategy and the characteristics of application operations.

For example, the MUF scheduling strategy requires a dispatching queue and a dispatching thread at the corresponding static priority for each *criticality level*. Each of these dispatching queues use the MLF ordering policy. In contrast, RMS requires a dispatching queue for each *rate* and dispatches operations either in static subpriority order or FIFO order, depending on the strategy implementation.

Event supplier and consumer registration: Event suppliers must provide configuration information when they register with TAO’s RT Event Service. First, each supplier provides an identifier that will accompany all events it sends. Second, each supplier indicates the event types it will *publish*. In addition, event consumers must provide configuration information when they register with TAO's RT Event Service. Each consumer *subscribes* to events of particular types and/or events that are associated with specific supplier identifiers.

3.3. QoS Specification in TAO

TAO supports QoS specification through its real-time scheduling service. An application describes the characteristics of each operation to its Scheduling Service, such as its criticality, period, worst case execution time, and dependencies on other information. The Scheduling Service stores these attributes in `RT_Info` descriptors, which are shown in Figure 7. TAO's Scheduling



```
struct  RT_Info
{
  Criticality criticality_;
  Time worstcase_exec_time_;
  Period period_;
  Importance importance_;
  Dependency_Info dependencies_;
};
```

Figure 7. Operation Characteristics

Service uses these `RT_Info` descriptors to prioritize operations, determine the number and types of dispatching queues, and store derived information about the operations, such as the assigned static dispatching priorities.

TAO's Scheduling Service also provides a specification for the real-time event service's dispatching queue configuration, based on the registered operation descriptors. The real-time scheduling service provides the number of queues, the type of each queue (*i.e.*, its ordering policy), the global priority number for the queue, and the corresponding OS thread priority at which operations will be dispatched from that queue.

3.4. Application Adaptive Mechanisms

Providing mechanisms and policies to specify and enforce QoS requirements in real-time middleware can shield the application developer from many tedious and error-prone details. There are cases, however, when controlled violations of this layering are actually beneficial. For example, a collection of application components that compete for the same set of resources may wish to avoid congestion and blocking delays due to excessive contention. In such cases, the application can play an active part in the management of its QoS requirements, as follows:

1. An application component can simply reduce its impact on other application components. For instance, it could reduce its resource demands or yielding resources frequently.
2. An application component may adjust its real-time behavior. For instance, it could change its rate of execution by registering for a different timer interval or poll a sensor data port that is clocked at a different rate.
3. An application component may behave adaptively with respect to the behavior of other components. For instance, it could set a “need data” timer as described in Section 3.2 and use an older value if the supplier is not ready.

The third technique is particular useful when integrating components whose real-time behavior can vary significantly. As real-time systems continue to be integrated with non-real-time data sources and applications, the need to balance the competing design forces of (1) encapsulating complex programming details and (2) supporting flexible adaptation in the application layer will certainly increase.

3.5. QuO Adaptive Mechanisms

As noted in Section 3.2, adaptation mechanisms may operate either on a shorter *in-band* time-scale or on a longer *out-of-band* time-scale. The QuO adaptive QoS management middleware supports both types of mechanisms. Moreover, QuO acts as a higher level middleware layer that can interoperate both with the application and with lower-level middleware to mitigate the design tension between encapsulation and flexibility described in Section 3.4.

We have inserted QuO middleware control in the path of the TAO RT Event Service suppliers and consumers, as illustrated in Figure 8. Our efforts

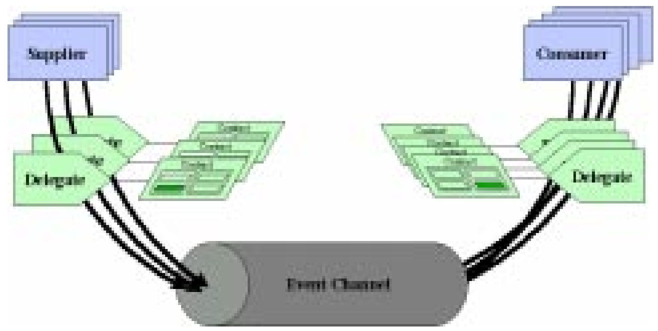


Figure 8. QuO Control of the TAO event channel

have concentrated on the supplier-side of the periodic sensor-actuator event channel examples supported by TAO. Within this context, QuO contracts and delegates support the following types of adaptivity in these types of TAO applications:

Changing the frequency of events: This adaptation varys the rate at which a supplier pushes events by intercepting events in the delegate and either pushing them *faster*, *i.e.*, requiring the delegate to generate events, or *slower*, *i.e.*, requiring the delegate to delay or eliminate events. In addition, the period of the consumer also can be varied by (1) changing the associated `RT_Info` attribute or (2) intercepting delivered events with the consumer-side delegate and pushing them to the actual consumer at a different period.

Changing the priority of events: Depending on the specified scheduling strategy, TAO uses different `RT_Info` attributes to control static priorities. In turn, these priorities are enforced by dispatching threads, and static and dynamic subpriorities which are enforced by dispatching queues. For example, the MUF (Stewart and Khosla, 1992) scheduling strategy implemented in TAO (Gill et al., 2000) uses *criticality* to assign static priority, *worst case execution time* and *period* to assign dynamic subpriority, and *importance* to assign static subpriority. Importance is a lower measure of priority used to break ties. Changing the priority and subpriority of events, therefore, involves changing the criticality, worst case execution time, period, and importance attributes for an event's `RT_Info`. Moreover, priority and worst case execution time are *inherent* aspects of the application's behavior. Therefore, application itself may need to adapt to change these aspects properly.

Changing how the consumer processes events: When an event is delivered to a consumer, the consumer processes it in some manner, *e.g.*, by displaying an image on a HUD or updating navigation data. To invoke different forms of consumer processing, the consumer-side delegate can choose between different methods or alternate arguments to methods based upon the state of the contract.

Change the event type: The supplier-side or the consumer-side delegates can change the event type of a supplied or delivered event. In turn, this changes the way an event will be propagated to, and/or processed by, event consumers.

TAO currently does not support changing of `RT_Info` attributes *in-band*. It does, however, support *out-of-band* reconfiguration and reassignment of priorities. For example, when the time-scale for application mode changes corresponds to the out-of-band time-scale for reconfiguration, *e.g.*, by an Adaptive Resource Manager (J. Huang et al., 1997), it may be sufficient to rely solely on out-of-band adaptation dynamic scheduling to control variations in event-level QoS (Doerr et al., 1999).

These types of *out-of-band* adaptation may prove too costly for certain applications, however, particularly when significant variations in QoS must be enforced on a fine-grain *in-band* time-scale. In such cases, creatively implemented QuO delegates can still achieve adaptation on an in-band time-scale, while preserving the correct operation of all QoS enforcement mechanisms *as they are configured*. For example, changing the priority of events and/or changing the period of events might entail setting up multiple `RT_Info` descriptions beforehand. This would entail creating one for each possible priority and period, and then changing the event type inside the delegate to the event type corresponding to the proper `RT_Info`.

The adaptation techniques outlined above can be combined to produce powerful application- and system-level adaptation strategies. For example, changing the frequency of events, and changing how the consumer processes events, can be generalized to support any arbitrary supplier-side and consumer-side processing of the event prior to (or instead of) delivery to the event channel or consumer. This technique supports filtering of events outside TAO's event service, duplicating events to a remote monitoring framework (Gill et al., 1999), adjusting event data, and so forth.

Changing the timing and distribution of events also supports a potentially powerful adaptation technique: *migration of processing*. This technique allows consumer-side functionality to be *pulled* into the supplier-side if, for example, the consumer was unable to process the data completely within the required time. Conversely, this technique allows supplier-side functionality to be *pushed* out to the consumer-side, *e.g.*, via a replication service. A real-time system can dynamically reconfigure to recover from overloaded hosts or to satisfy real-time requirements in the face of an increased number of aperiodic events. It can do so by increasing the amount of processing performed by the delegate on the supplier side and thus decreasing the processing load on the consumer.

The QuO delegates and contracts also can impose control over the suppliers from which a consumer can receive events. This feature can be used to maintain and enforce real-time requirements. For example, as resources become constrained in a system, a QuO delegate can prohibit delivery of events to a consumer so that another consumer can maintain its period requirement. This level of control is not supported currently by existing TAO RT Event Service mechanisms, and is another example of the "meta-level" programming power provided by QuO.

3.6. QoS Specification in QuO

The QoS specification mechanisms provided by TAO, which were described in Section 3.3, have been shown to be sufficient for building mission-critical applications with stringent real-time requirements (Harrison et al., 1997; Levine et al., 1998; Gill et al., 2000; Doerr et al., 1999). However, these applications were built by researchers and developers with significant experience and expertise both in real-time systems and in distributed object-oriented software engineering. It is a non-trivial exercise for real-time application developers to recognize (1) which strategies and configurations should be used and (2) how they should be applied to achieve the desired real-time behavior.

For example, the code required to (1) create and set up TAO's Scheduling Service and RT Event Service and (2) to connect suppliers and consumers to the Event Service is fairly low-level and specific to the TAO event channel implementation. However, these steps are also relatively common from one Event Service application to the next. Moreover, much of the low-level code dealing with the association of suppliers, consumers, and QoS information is at a lower level of abstraction than the application-level requirements described in Section 3.1. The implementation details also require the specification of unused `RT_Info` data on the supplier and the consumer, as well as the creation of multiple unique names for each event.

In the spirit of CORBA IDL and QuO's QDL, therefore, we have developed an aspect language to deal with the application-level real-time requirements of periodic sensor-actuator systems. This language, called Real-time Specification Language (RSL), allows real-time application programmers to define the supplier and consumer configurations for as many event types and real-time attributes as are of interest. The programmer does so without supplying the remainder of the information necessary to set up the event channel, but that is not relevant to the application requirements.

RSL allows a QoS programmer to specify the following:

- A set of events; and
- A set of event suppliers, the events each will generate; and (optionally) the period at which the event will be generated (or that the period will be specified at runtime); and
- A set of event consumers, the events each consumes, and any or all of the criticality, worst case time, typical time, cached time, and importance values (or that the values will be specified at runtime).

The RSL code generator then outputs the code that (1) creates the scheduling service servant, (2) creates the event channel and register it with the name service, and (3) initializes each server and consumer. This code will generate unique names for the events, encode dummy values for any `RT_Info` attributes not specified in the RSL specification (including unused attributes), specify the values for attributes specified in the RSL, and put the runtime specified attributes into the signature of the routine.

For example, the RSL specification in Figure 9 will generate a class named `QuO_RT_Class` that contains (along with the scheduler and event channel configuration code) the following methods:

```
RT_SPECS QuO_RT_Class {
  EVENTS { event1, event2, event3 }
  EC_SUPPLIER supplier1_setup_routine {
    GENERATES { event1, event2}
    APPLICATION_SPECIFIED { period }
  }
  EC_CONSUMER consumer1_setup_routine {
    CONSUMES { event2 }
    APPLICATION_SPECIFIED { criticality, importance }
  }
  EC_CONSUMER consumer2_setup_routine {
    CONSUMES { event1 }
    APPLICATION_SPECIFIED {}
    CONSTANT { criticality = very high;
               importance = very low;
    }
  }
}
```

Figure 9. Example RSL specification

- The `supplier1_setup_routine` method sets up a supplier that generates `event1` and `event2` (although it will have unique names to refer to them) and generates them at periods that will be passed into the method when it is called at runtime.
- The second and third methods set up two consumers. One method initializes a consumer that consumes `event2` with a criticality and importance passed in at initialization time. The other method initializes a consumer that consumes event1 with a very high criticality and a very low importance[2].

Each of these three methods will encode dummy values for all the other attributes of `RT_Info`.

3.7. Unifying QoS Specification and Enforcement Layers

The TAO and QuO QoS enforcement mechanisms described in Sections 3.2, 3.4, and 3.5, combined with the specification capabilities described in Sections 3.3 and 3.6, provide a powerful framework for meeting the application requirements listed in Section 3.1. To illustrate how the integrate TAO and QuO framework can be used to meet the QoS requirements of mission-critical real-time applications, we describe an example sensor-actuator application, representative of those found in event-driven avionics systems and illustrated in Figure 10.

[2] TAO recognizes criticality and importance values of VERY_HIGH, HIGH, MEDIUM, LOW, and VERY_LOW.

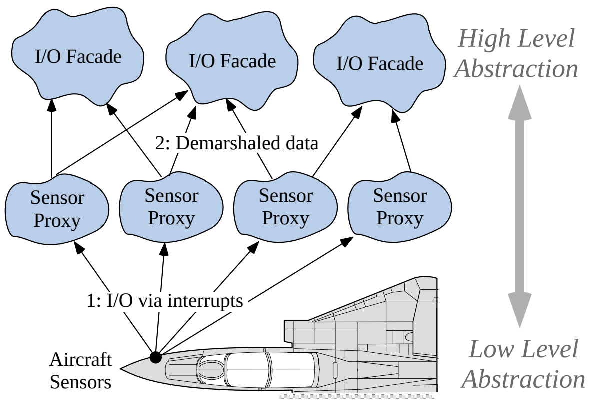


Figure 10. A Real-time Event-Driven Avionics System

Sensor-actuator applications, such as those found in embedded avionics systems (Harrison et al., 1997), contain many subsystems operating in concert, responding to sensor data events, and managing functions of the aircraft. These subsystems include functionality, such as the heads-up display and navigation subsystems. Sensor data can come from a number of sensors on the aircraft, such as a global positioning satellite receivers, or various radar sensors.

In general, these subsystems have crucial QoS requirements, such as real-time response, dependability, and resource utilization. Moreover, the set of QoS requirements that must be satisfied can be highly variable, differing (1) between families of aircraft and between specific products within a family of aircraft, (2) between subsystems within a single aircraft, and (3) even between missions and between operating modes, within a single aircraft subsystem.

Currently fielded avionics systems are designed to be configured between missions, so that pilots can manually switch between mission computer operating modes (Doerr and Sharp, 1999). However, for the most part current avionics software systems are configured statically. Therefore, changes occur in the form of software upgrade cycles and mission reprogramming. These legacy sensor-actuator systems are inflexible because the sensors are tightly coupled to the actuators, and the software is often tightly coupled to special-purpose hardware.

To overcome these limitations, it is necessary to apply new engineering methods to the process of developing these systems. In particular, improving the reliability and flexibility of distributed real-time systems requires advanced techniques, such as leveraging COTS hardware and software, increasing software reuse through middleware, and applying design patterns and adaptive object-oriented programming techniques. Moreover, these techniques serve to manage the monetary and time costs of the overall system development lifecycle.

Adaptable DOC middleware frameworks, such as QuO and TAO, implement the necessary patterns, strategies, and infrastructure needed to build modern, more flexible avionics systems. In the example illustrated in Figure 10, sensors and actuators are decoupled, hidden from one another through sensor proxies and event channels. This allows sensors and actuators to be reconfigured, upgraded, or replaced dynamically without affecting the other subsystems. Furthermore, the avionics software can automatically adapt to changing missions and operational conditions by making tradeoffs between QoS dimensions, and dynamically reallocating resources. For example, an avionics system may temporarily sacrifice progress of non-critical operations for increased performance of critical operations.

This adaptable architecture requires the following combined assets:

- Middleware that can decouple sensors and actuators while providing real-time enforcement, such as that provided by the TAO real-time ORB;
- Dynamic resource managers and mechanisms, such as RT-ARM (J. Huang et al., 1997) or Darwin (Prashant Chandra, 1998);
- Adaptable middleware, such as the QuO system, which can provide application-level control and adaptation based upon changing mission goals, operational modes, environmental conditions, and changing QoS tradeoffs.

These capabilities are complementary. The TAO ORB enables the decoupling of sensor and actuator functionality while guaranteeing real-time delivery of sensor events. Dynamic resource managers enable access to and reallocation of resources in response to changing system conditions and mission needs, while the QuO middleware enables the application- and subsystem-level control to allocate the resources and functionality to the proper mission or operating mode.

3.8. Open Research Issues

This section illustrates how application QoS requirements can be specified to higher-level DOC middleware, such as QoO, which in turn maps these specifications onto efficient, predictable, and scalable mechanisms in lower-level middleware, such as TAO, in order to meet the QoS requirements end-to-end. To make the example concrete, and to document our on-going R&D activities in the DARPA Quorum integration effort, we have focused on a particular use-case in the avionics mission computing domain. We are planning to address the following open research issues, however, to demonstrate the broader applicability of our adaptive multi-level middleware strategy.

Leveraging existing QoS research: The operating system and networking research communities have produced a wealth of techniques, architectures, and empirical information for QoS management issues in the network and OS kernel layers. These techniques must be used as the basis for developing and evaluating middleware QoS management approaches, and wherever possible built into end-to-end middleware solutions. Some middleware solutions leverage particular point-solutions for QoS management, *e.g.*, TAO leverages preemptive thread scheduling in the OS kernel to enforce static priorities. However, a more comprehensive integration of policies and mechanisms *at the middleware level* is needed.

Identifying general-purpose patterns: To leverage existing QoS research at the OS and networking levels effectively, it is necessary to identify the key general-purpose patterns for *composing* the lower level mechanisms end-to-end. For example, identifying different patterns for co-scheduling network and CPU resources along a request-response path between a client and a server will be relevant to many applications. These client-server resource allocation patterns will in turn guide the creation of flexible middleware that is suited to the common requirements of a wide range of QoS-enabled client-server applications.

Identifying domain-specific patterns: Where effective resolutions of common design forces are captured by general-purpose patterns, each individual application domain also produces design forces that are specific to that domain. QoS requirements such as timing, utilization, or reliability constraints may differ between different application domains, *e.g.*, telecommunications and sensor-actuator systems. Additional research is needed to identify the key design forces for each domain, along with the patterns that can resolve those forces.

Building flexible QoS frameworks: After identifying the general-purpose and domain-specific patterns outlined above, along with the necessary lower-level mechanisms for QoS enforcement, it is possible to reify these patterns in flexible QoS frameworks. Implementing key QoS mechanisms, strategies and policies, and embedding these within middleware frameworks, allows DOC middleware to support (1) the common requirements of a wide range of QoS-enabled applications and (2) the specific requirements of individual domains and applications. Moreover, building these frameworks offers practical insights into additional patterns and techniques for QoS management in adaptive DOC middleware for distributed and embedded systems.

4. Related Work

Real-time middleware is an emerging field of study. An increasing number of research efforts are focusing on integrating QoS and real-time scheduling into middleware like CORBA. This section compares our work on TAO with related QoS middleware integration research.

CORBA-related QoS research:

- *Mitre Real-time CORBA:* Krupp, *et al.*, (Thuraisingham et al., 1994) at MITRE Corporation were among the first to elucidate the requirements of real-time CORBA systems. A system consisting of a commercial off-the-shelf real-time OS, a CORBA-compliant ORB, and a real-time object-oriented database management system is under development (USAF-RFI:97, 1997). Similar to TAO's original static scheduling service (Schmidt et al., 1998), their initial static scheduling approach used RMS, though a strategy for dynamic deadline monotonic scheduling support has been designed (Cooper et al., 1997).

- *URI TDMI:* Wolfe, *et al.*, are developing a real-time CORBA system at the US Navy Research and Development Laboratories (NRaD) and the University of Rhode Island (URI) (Wolfe et al., 1997). The system supports expression and enforcement of dynamic end-to-end timing constraints through timed distributed operation invocations (`TDMI`s) (Fay-Wolfe et al., 1995). A `TDMI` corresponds to TAO's `RT_Operation` (Schmidt et al., 1998). Likewise, an `RT_Environment` structure contains QoS parameters similar to those in TAO's `RT_Info`. One difference between TAO and the URI approaches is that `TDMI`s express required timing constraints, *e.g.*, deadlines relative to the current time, whereas `RT_Operation`s publish their resource, *e.g.*, CPU time, requirements.

- *UCSB Realize:* The Realize project at UCSB (Kalogeraki et al., 1997) supports soft real-time resource management of CORBA distributed systems. Realize aims to reduce the difficulty of developing real-time systems and to permit distributed real-time programs to be programmed, tested, and debugged as easily as single sequential programs. Realize integrates distributed real-time scheduling with fault-tolerance, fault-tolerance with totally-ordered multicasting, and totally-ordered multicasting with distributed real-time scheduling, within the context of OO programming and existing standard operating systems. The Realize resource management model can be hosted on top of TAO (Kalogeraki et al., 1997).

- *UIUC Epiq:* The Epiq project (Feng et al., 1997) defines an open real-time CORBA scheme that provides QoS guarantees and run-time scheduling flexibility. Epiq explicitly extends TAO's off-line scheduling model to provide on-line scheduling. In addition, Epiq allows clients to be added and removed dynamically via an admission test at run-time. The Epiq project is work-in-progress and empirical results are not yet available.

- *UCI TMO:* The Time-triggered Message-triggered Objects (TMO) project (Kim, 1997) at the University of California, Irvine, supports the integrated design of distributed OO systems and real-time simulators of their operating environments. The TMO model provides structured timing semantics for distributed real-time object-oriented applications by extending conventional invocation semantics for object methods (*i.e.*, CORBA operations) to include (1) invocation of time-triggered operations based on system times and (2) invocation and time bounded execution of conventional message-triggered operations.

  TAO differs from TMO in that it provides a complete CORBA ORB, as well as CORBA ORB services and real-time extensions. Timer-based invocation capabilities are provided through TAO's Real-Time Event Service (Harrison et al., 1998). Where the TMO model creates new ORB services to provide its time-based invocation capabilities (Kim and Shokri, 1999), TAO provides a subset of these capabilities by extending the standard CORBA COS Event Service. We believe TMO and TAO are complementary technologies since (1) TMO extends and generalizes TAO's existing time-based invocation capabilities and (2) TAO provides a configurable and dependable connection infrastructure needed by the TMO CNCM service. We are currently collaborating with the UCI TMO team to integrate the TAO and TMO middleware as part of the DARPA Quorum integration project.

Non-CORBA-related QoS research:

- *ARMADA:* The ARMADA project (Mehra et al., 1997; Abdelzaher et al., 1997) defines a set of communication and middleware services that support fault-tolerant and end-to-end guarantees for real-time distributed applications. ARMADA provides real-time communication services based on the X-kernel and the Open Group's MK micro-kernel. This infrastructure provides a foundation for constructing higher-level real-time middleware services.

  TAO differs from ARMADA in that most of the real-time infrastructure features in TAO are integrated into its ORB Core (Schmidt et al., 2000) and I/O subsystem (Kuhns et al., 1999b), rather than in a microkernel. In addition, TAO implements the OMG CORBA standard, while also providing the hooks necessary to integrate with an underlying real-time I/O subsystem and OS. Thus, the real-time services provided by ARMADA's communication system can be utilized by TAO to support standards-based applications running over a vertically and horizontally integrated real-time system.

- *CMU Publisher/Subscriber:* Rajkumar, *et al.*, (Rajkumar et al., 1995) at CMU developed a real-time Publisher/Subscriber model that is similar to the TAO's Real-time Event Service (Harrison et al., 1997), *e.g.*, it uses real-time threads to prevent priority inversion within its communication framework. The CMU model does not utilize any QoS specifications from publishers (event suppliers) or subscribers (event consumers), however. Therefore, scheduling is based on the assignment of request priorities, which is not addressed by the CMU model.

In contrast, TAO's Scheduling Service and real-time Event Service utilize QoS parameters from suppliers and consumers to assure resource access via priorities. One interesting aspect of the CMU Publisher/Subscriber model is the separation of priorities for subscription and data transfer. By handling these activities with different threads, with possibly different priorities, the impact of on-line scheduling on real-time processing can be minimized.

- *UCI RED-Linux Scheduling Framework:* Wang, *et al.* (Wang et al., 1999), at the University of California, Irvine, have proposed a general scheduling framework to unify three distinct kinds of scheduling approaches: *priority-based*, *time-based*, and *share-based*. Wang, *et al.*, decompose scheduling behavior into policy (*allocator*) and mechanism (*dispatching*) components, which are similar to the TAO scheduling service framework. They have implemented the dispatching portion of this framework in their real-time extensions to the Linux kernel, called RED-Linux.

  While the RED-Linux approach to scheduling relies on special-purpose extensions to the OS kernel, TAO's scheduling service relies only on commonly available OS features, such as preemptive thread priorities. Therefore, TAO's dispatching mechanisms can leverage standards-based CORBA middleware and it can perform effectively on a wide range of commonly available real-time and general-purpose OS platforms.

- *OSU Share-based Scheduling:* Tyan, *et al.* (Tyan and Hou, 1999), at Ohio State University, have developed a general framework for share-based scheduling. They demonstrate their framework's ability to implement a number of well-known fair queueing algorithms, as well as its ability to implement new kinds of share-based scheduling algorithms.

  TAO's strategized scheduling service differs in that it uses priority based scheduling approaches, in order to address applications with hard real-time requirements. In our future research, we are investigating share-based scheduling and its interaction with priority-based scheduling for various classes of real-time applications.

5. Concluding Remarks

Next-generation distributed and embedded systems requires a wide range of features to support increasingly stringent quality of service (QoS) aspects involving bandwidth, latency, jitter, and dependability. In addition to requiring support for these QoS requirements, next-generation systems are becoming *enabling technologies* for companies competing in markets where deregulation and global competition motivate the need for increased software productivity, quality, and cost-effectiveness. Due to constraints on footprint, performance, and weight/power consumption, however, development of distributed and embedded systems has historically lagged far behind mainstream software development methodologies. As a result, these systems are extremely expensive and time-consuming to develop, validate, optimize, deploy, maintain, and upgrade.

The goal of COTS middleware is to decrease the cycle-time and effort required to develop high-quality distributed and embedded systems by composing applications out of flexible and modular reusable software components and services, rather than building them entirely from scratch using proprietary tools. Achieving this goal is essential in contemporary software development environments, which are increasingly constrained in terms of time and effort. Moreover, COTS middleware helps reduce the tight coupling of systems to their current configuration and operating environment, so they can adapt more readily to new market opportunities, technology innovations, and changes in run-time situational environments.

The adaptive real-time middleware policies and mechanisms based on TAO and QuO described in this paper support applications whose resource requirements can vary significantly at run-time. These capabilities make it possible to develop DOC applications that can respond adequately to changing situational features in their run-time environment. TAO and QuO have been applied successfully to a range of real-time applications, including avionics mission computing systems at Boeing (Harrison et al., 1997; Christopher D. Gill et al., 1999; Doerr et al., 1999), the SAIC Run Time Infrastructure (RTI) implementation (O'Ryan et al., 1999) for the Defense Modeling and Simulation Organization's (DMSO) High Level Architecture (HLA) (Kuhl et al., 1999), and high-energy testbeam acquisition systems at SLAC (SLAC,) and CERN (Kruse, 1997).

The source code and documentation for the TAO ORB and its Real-time (RT) Event Service and Scheduling Service are freely available from URL `www.cs.wustl.edu/~schmidt/TAO.html`. Information about the QuO software release is available at `www.dist-systems.bbn.com/tech/QuO`.

References

Abdelzaher, T., S. Dawson, W.-C.Feng, F.Jahanian, S. Johnson, A. Mehra, T. Mitton, A. Shaikh, K. Shin, Z. Wang, and H. Zou: 1997, 'ARMADA Middleware Suite'. In: *Proceedings of the Workshop on Middleware for Real-Time Systems and Services*. San Francisco, CA.

Box, D.: 1997, *Essential COM*. Addison-Wesley, Reading, MA.

Christopher D. Gill et al.: 1999, 'Applying Adaptive Real-time Middleware to Address Grand Challenges of COTS-based Mission-Critical Real-Time Systems'. In: *Proceedings of the 1st IEEE International Workshop on Real-Time Mission-Critical Systems: Grand Challenge Problems*.

Cooper, G., L. C. DiPippo, L. Esibov, R. Ginis, R. Johnston, P. Kortman, P. Krupp, J. Mauer, M. Squadrito, B. Thuraisingham, S. Wohlever, and V. F. Wolfe: 1997, 'Real-Time CORBA Development at MITRE, NRaD, Tri-Pacific and URI'. In: *Proceedings of the Workshop on Middleware for Real-Time Systems and Services*. San Francisco, CA.

DARPA: 1999, 'The Quorum Program'. http://www.darpa.mil/ito/research/quorum/index.html.

Dittia, Z. D., G. M. Parulkar, and J. R. Cox, Jr.: 1997, 'The APIC Approach to High Performance Network Interface Design: Protected DMA and Other Techniques'. In: *Proceedings of INFOCOM '97*. Kobe, Japan, pp. 179–187.

Doerr, B. S. and D. C. Sharp: 1999, 'Freeing Product Line Architectures from Execution Dependencies'. In: *Proceedings of the 11th Annual Software Technology Conference*.

Doerr, B. S., T. Venturella, R. Jha, C. D. Gill, and D. C. Schmidt: 1999, 'Adaptive Scheduling for Real-time, Embedded Information Systems'. In: *Proceedings of the 18th IEEE/AIAA Digital Avionics Systems Conference (DASC)*.

Eide, E., K. Frei, B. Ford, J. Lepreau, and G. Lindstrom: 1997, 'Flick: A Flexible, Optimizing IDL Compiler'. In: *Proceedings of ACM SIGPLAN '97 Conference on Programming Language Design and Implementation (PLDI)*. Las Vegas, NV.

Fay-Wolfe, V., J. K. Black, B. Thuraisingham, and P. Krupp: 1995, 'Real-time Method Invocations in Distributed Environments'. Technical Report 95-244, University of Rhode Island, Department of Computer Science and Statistics.

Feng, W., U. Syyid, and J.-S. Liu: 1997, 'Providing for an Open, Real-Time CORBA'. In: *Proceedings of the Workshop on Middleware for Real-Time Systems and Services*. San Francisco, CA.

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

Gill, C. D., D. L. Levine, C. O'Ryan, and D. C. Schmidt: 1999, 'Distributed Object Visualization for Sensor-Driven Systems'. In: *Proceedings of the 18th IEEE/AIAA Digital Avionics Systems Conference (DASC)*.

Gill, C. D., D. L. Levine, and D. C. Schmidt: 2000, 'The Design and Performance of a Real-Time CORBA Scheduling Service'. *The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*.

Gokhale, A. and D. C. Schmidt: 1999, 'Optimizing a CORBA IIOP Protocol Engine for Minimal Footprint Multimedia Systems'. *Journal on Selected Areas in Communications special issue on Service Enabling Platforms for Networked Multimedia Systems* **17**(9).

Harrison, T. H., D. L. Levine, and D. C. Schmidt: 1997, 'The Design and Performance of a Real-time CORBA Event Service'. In: *Proceedings of OOPSLA '97*. Atlanta, GA.

Harrison, T. H., C. O'Ryan, D. L. Levine, and D. C. Schmidt: 1998, 'The Design and Performance of a Real-time CORBA Event Service'. *submitted to the Journal on Selected Areas in Communications special issue on Service Enabling Platforms for Networked Multimedia Systems*.

Henning, M. and S. Vinoski: 1999, *Advanced CORBA Programming With C++*. Addison-Wesley Longman.

J. Huang et al.: 1997, 'RT-ARM: A real-time adaptive resource management system for distributed mission-critical applications'. In: *Workshop on Middleware for Distributed Real-Time Systems, RTSS-97*. San Francisco, California.

Kalogeraki, V., P. Melliar-Smith, and L. Moser: 1997, 'Soft Real-Time Resource Management in CORBA Distributed Systems'. In: *Proceedings of the Workshop on Middleware for Real-Time Systems and Services*. San Francisco, CA.

Kiczales, G.: 1996, 'Beyond the Black Box: Open Implementation'. *IEEE Software*.

Kiczales, G.: 1997, 'Aspect-Oriented Programming'. In: *Proceedings of the 11th European Conference on Object-Oriented Programming*.

Kim, K. and E. Shokri: 1999, 'Two CORBA Services Enabling TMO Network Programming'. In: *Fourth International Workshop on Object-Oriented, Real-Time Dependable Systems*.

Kim, K. H. K.: 1997, 'Object Structures for Real-Time Systems and Simulators'. *IEEE Computer* pp. 62–70.

Klein, M. H., T. Ralya, B. Pollak, R. Obenza, and M. G. Harbour: 1993, *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Norwell, Massachusetts: Kluwer Academic Publishers.

Kruse, A.: 1997, 'CMS Online Event Filtering'. In: *Computing in High-energy Physics (CHEP 97)*. Berlin, Germany.

Kuhl, F., R. Weatherly, and J. Dahmann: 1999, *Creating Computer Simulation Systems*. Upper Saddle River, New Jersey: Prentice Hall PTR.

Kuhns, F., C. O'Ryan, D. C. Schmidt, and J. Parsons: 1999a, 'The Design and Performance of a Pluggable Protocols Framework for Object Request Broker Middleware'. In: *Proceedings of the IFIP* 6^{th} *International Workshop on Protocols For High-Speed Networks (PfHSN '99)*. Salem, MA.

Kuhns, F., D. C. Schmidt, and D. L. Levine: 1999b, 'The Design and Performance of a Real-time I/O Subsystem'. In: *Proceedings of the* 5^{th} *IEEE Real-Time Technology and Applications Symposium*. Vancouver, British Columbia, Canada, pp. 154–163.

Levine, D. L., C. D. Gill, and D. C. Schmidt: 1998, 'Dynamic Scheduling Strategies for Avionics Mission Computing'. In: *Proceedings of the 17th IEEE/AIAA Digital Avionics Systems Conference (DASC)*.

Liu, C. and J. Layland: 1973, 'Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment'. *JACM* **20**(1), 46–61.

Loyall, J. P., D. E. Bakken, R. E. Schantz, J. A. Zinky, D. Karr, R. Vanegas, and K. R. Anderson: 1998a, 'QuS Aspect Languages and Their Runtime Integration'. *Proceedings of the Fourth Workshop on Languages, Compilers and Runtime Syste,s for Sclable Components*.

Loyall, J. P., R. E. Schantz, J. A. Zinky, and D. E. Bakken: 1998b, 'Specifying and Measuring Quality of Service in Distributed Object Systems'. In: *Proceedings of The 1st IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 98)*.

Mehra, A., A. Indiresan, and K. G. Shin: 1997, 'Structuring Communication Software for Quality-of-Service Guarantees'. *IEEE Transactions on Software Engineering* **23**(10), 616–634.

Object Management Group: 1995, 'CORBAServices: Common Object Services Specification, Revised Edition'. Object Management Group, 95-3-31 edition.

Object Management Group: 1997a, 'Concurrency Services Specification'. Object Management Group, OMG Document formal/97-12-14 edition.

Object Management Group: 1997b, ‘Transaction Services Specification’. Object Management Group, OMG Document formal/97-12-17 edition.

Object Management Group: 1998a, ‘Fault Tolerance CORBA Using Entity Redundancy RFP’. Object Management Group, OMG Document orbos/98-04-01 edition.

Object Management Group: 1998b, ‘Security Service Specification’. Object Management Group, OMG Document ptc/98-12-03 edition.

Object Management Group: 1999a, ‘Persistent State Service 2.0 Specification’. Object Management Group, OMG Document orbos/99-07-07 edition.

Object Management Group: 1999b, ‘Realtime CORBA Joint Revised Submission’. Object Management Group, OMG Document orbos/99-02-12 edition.

Object Management Group: 1999c, ‘The Common Object Request Broker: Architecture and Specification’. Object Management Group, 2.3 edition.

O’Ryan, C., D. C. Schmidt, and D. Levine: 1999, ‘Applying a Scalable CORBA Events Service to Large-scale Distributed Interactive Simulations’. In: *Proceedings of the* 5^{th} *Workshop on Object-oriented Real-time Dependable Systems*. Montery, CA.

Pal, P., J. Loyall, R. Schantz, J. Zinky, , R. Shapiro, and J. Megquier: 2000, ‘Using QDL to Specify QoS Aware Distributed (QuO) Application Configuration’. In: *Proceedings of The 3rd IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 00)*.

Prashant Chandra, e. a.: 1998, ‘Darwin: Resource Management for Value-Added Customizable Network Service’. In: *Sixth IEEE International Conference on Network Protocols (ICNP’98)*. Austin, TX.

Pyarali, I., C. O’Ryan, D. C. Schmidt, N. Wang, V. Kachroo, and A. Gokhale: 1999, ‘Applying Optimization Patterns to the Design of Real-time ORBs’. In: *Proceedings of the* 5^{th} *Conference on Object-Oriented Technologies and Systems*. San Diego, CA.

Rajkumar, R., M. Gagliardi, and L. Sha: 1995, ‘The Real-Time Publisher/Subscriber Inter-Process Communication Model for Distributed Real-Time Systems: Design and Implementation’. In: *First IEEE Real-Time Technology and Applications Symposium*.

Schantz, R. E., J. A. Zinky, D. A. Karr, D. E. Bakken, J. Megquier, and J. P. Loyall: 1999, ‘An Object-level Gateway Supporting Integrated-Property Quality of Service’. In: *Proceedings of The 2nd IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC 99)*.

Schmidt, D. C.: 1990, ‘GPERF: A Perfect Hash Function Generator’. In: *Proceedings of the* 2^{nd} *C++ Conference*. San Francisco, California, pp. 87–102.

Schmidt, D. C. and C. Cleeland: 1999, ‘Applying Patterns to Develop Extensible ORB Middleware’. *IEEE Communications Magazine* **37**(4).

Schmidt, D. C., T. H. Harrison, and E. Al-Shaer: 1995, ‘Object-Oriented Components for High-speed Network Programming’. In: *Proceedings of the* 1^{st} *Conference on Object-Oriented Technologies and Systems*. Monterey, CA.

Schmidt, D. C., D. L. Levine, and S. Mungee: 1998, ‘The Design and Performance of Real-Time Object Request Brokers’. *Computer Communications* **21**(4), 294–324.

Schmidt, D. C., S. Mungee, S. Flores-Gaitan, and A. Gokhale: 2000, ‘Software Architectures for Reducing Priority Inversion and Non-determinism in Real-time Object Request Brokers’. *Journal of Real-time Systems*.

Schmidt, D. C. and T. Suda: 1994, ‘An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems’. *IEE/BCS Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems)* **2**, 280–293.

Sha, L., R. Rajkumar, J. Lehoczky, and K. Ramamritham: 1989, ‘Mode Change Protocols for Priority-Driven Preemptive Scheduling’. *The Journal of Real-Time Systems* **1**, 243–264.

John A. Stankovic and Krithi Ramamritham, *Advances in Real-Time Systems*, IEEE Computer Society Press, 1992.

SLAC, ‘BaBar Collaboration Home Page’. http://www.slac.stanford.edu/BFROOT/.

Steinmetz, R.: 1990, ‘Synchronization Properties in Multimedia Systems’. *Journal on Selected Areas in Communications* **8**(3).

Stewart, D. B. and P. K. Khosla: 1992, ‘Real-Time Scheduling of Sensor-Based Control Systems’. In: W. Halang and K. Ramamritham (eds.): *Real-Time Programming*. Tarrytown, NY: Pergamon Press.

Thuraisingham, B., P. Krupp, A. Schafer, and V. Wolfe: 1994, ‘On Real-Time Extensions to the Common Object Request Broker Architecture’. In: *Proceedings of the Object Oriented Programming, Systems, Languages, and Applications (OOPSLA) Workshop on Experiences with CORBA*.

Tyan, H.-Y. and J. C. Hou: 1999, ‘A Rate-Based Message Scheduling Paradigm’. In: *Fourth International Workshop on Object-Oriented, Real-Time Dependable Systems*.

USAF-RFI:97: 1997, ‘Statement of Work for the Extend Sentry Program, CPFF Project, ECSP Replacement Phase II’. Submitted to OMG in response to RFI ORBOS/96-09-02.

Vanegas, R., J. A. Zinky, J. P. Loyall, D. Karr, R. E. Schantz, and D. E. Bakken: 1998, ‘QuO’s Runtime Support for Quality of Service in Distributed Objects’. *Proceedings of Middleware 98, the IFIP International Conference on Distributed Systems Platform and Open Distributed Processing*.

Wang, S., Y.-C. Wang, and K.-J. Lin: 1999, ‘A General Scheduling Framework for Real-Time Systems’. In: *IEEE Real-Time Technology and Applications Symposium*.

Wolfe, V. F., L. C. DiPippo, R. Ginis, M. Squadrito, S. Wohlever, I. Zykh, and R. Johnston: 1997, ‘Real-Time CORBA’. In: *Proceedings of the Third IEEE Real-Time Technology and Applications Symposium*. Montréal, Canada.

Wollrath, A., R. Riggs, and J. Waldo: 1996, ‘A Distributed Object Model for the Java System’. *USENIX Computing Systems* **9**(4).

Zinky, J. A., D. E. Bakken, and R. Schantz: 1997, ‘Architectural Support for Quality of Service for CORBA Objects’. *Theory and Practice of Object Systems* **3**(1).