Review of Support for Real-Time in CORBA-based Architectures

Calin Curescu and Simin Nadjm-Tehrani
Department of Computer and Information Science
Linköping Univeristy
[calcu/simin@ida.liu.se]

October 18, 2000

1 Introduction

An increasingly important class of distributed applications require end-to-end support for various stringent Quality of Service (QoS) guarantees. Applications like telecommunication systems, multimedia systems (video-conferencing), distributed command and control systems, interactive simulations, and avionics, rely on QoS aspects like bandwidth, latency, jitter, dependability, timeliness [14].

On the other hand, the need for flexible designs, rapid development of new applications based on reuse of existing modules and COTS, and distributed system development based on multiple platforms and programming languages, has led to a need for technologies based on open achitectures and standards.

A major such effort is the well-established standard supporting distributed objects via Common Object Request Broker Architecture (CORBA) [21]. It is obvious that any research around methods to build distributed systems with real-time or dependability properties can not ignore the above needs and trend. In particular, the above mentioned class of distributed applications would greatly benefit from a CORBA system with or without a dedicated Object Request Broker (ORB) which supports real-time (RT) or fault-tolerance (FT) requirements. In this report we review the case for the real-time aspect, as well as its extension towards QoS. The fault-tolerance issues are naturally discussed within the distributed systems research community, as a major part of the area of middleware design. A separate report would be needed to cover the area of fault-tolerant middleware design.

Here we consider the ways that a CORBA based architecture can be enriched with mechanisms to deal with real-time. This can in principle be done in three different ways. The first, paragmatic approach, is to add new modules which directly affect (steer) the real-time behaviour of objects in between the CORBA (ORB) layer and the operating system. This approach denoted by the *interceptor* technique, has obvious advantages from an industry point of view. One does not have to change an ORB or an application module but only do some "diversion" in between.

The second approach is the more tailor-made technique: designing a specific ORB which is specifically equipped with primitives to enforce real-time. This is the approach taken by several influential research groups, but has the disadvantage that makes the strict separation of the application and the middleware somewhat diluted. In order to enforce hard real-time requirements in an application, several modules such as scheduling mechanism, priority policy, etc. are fixed within the ORB. However, the benefits of open systems in terms of independence form the platform and the underlying development environments are preserved.

The third approach is to add a layer on top of the ORB, "in between" the application and the middleware. More precisely, it means separating the application's functional code from the specification of its requirements. The QoS requirements for an application, for example, can be used to automatically generate parameters for local objects whose task is to manage system resources so that the QoS requirements can be met.

In this report we provide a short survey of several works in the area of real-time CORBA, illustrating each of the above approaches by at least one representative. Since a large number of publications in the last few years have been devoted to explain the details of each method, we believe there is a need for a short survey which captures the fundamental aspects of each method, and provides a quick exposure to the area. It is not claimed that this survey is a complete survey of all the works in the area, but classes of similar works are well-represented.

In 1995 a special interest group (SIG) was formed within OMG with the goal of extending the CORBA standard with support of RT applications. Their definition of RT-CORBA: *"RT-CORBA deals with the expression and enforcement of real-time constraints on end to end execution in a CORBA system"* [18]. In this report we look back at the characteristics needed for introducing real-time in CORBA architectures and explain how the choices in the recent Standard were influenced. We further explain some recent activities which are likely to influence future extensions towards more dynamic solutions to support QoS.

The paper is organised as follows. Section 2 is a high-level overview of the CORBA architecture. In section 3 we present the characteristics needed by a CORBA system in order to support real-time specification and enforcement. Section 4 presents two of the implementations on which much of the RT-CORBA specification relies. In section 5 we present the RT CORBA 1.0 specification and the proposal received by OMG for dynamic scheduling support in RT CORBA. Section 6 is devoted to two of the frameworks for specifying and enforcing application-centered QoS in a CORBA environment. Section 8 briefly presents some of the latest research and section 9 concludes the report.

2 CORBA Overview

The *Object Management Group* (OMG) is an international industry consortium that promotes the theory and practice of object oriented software development. Their goal is to provide a common architectural framework, across heterogeneous hardware platforms, programming languages, and operating systems, for inter-communication of application objects [21].

An object oriented approach was chosen because of the need for components

that interact with each other through well known interfaces. Through encapsulation reusability and security of the different components are enhanced. Modular production of software is supported, developers can assemble applications from COTS components, which in turn leads to shorter development cycles.

OMG performs no development by itself. The approach is to issue Requests For Proposals (RFP), which solicit specifications of components to fit into a broad *Object Management Architecture* (OMA). Members then propose, review and adopt specifications. Once a specification is adopted the different implementations must comply to the specification. The following section presents the main components of the CORBA Architecture [12, 21, 3].

2.1 The OMA

The *Object Request Broker* (ORB), the Core part of OMA, is the communication bus for Objects. The technology adopted for the ORB is known as *Common Object Request Broker Architecture* (CORBA). It specifies a framework for communicating objects. It allows to transparently invoke operations on distributed objects spanning different:

- programming languages - C, C++, Java, Ada, Smalltalk, Cobol
- operating systems - Win32, Linux, Solaris, RT-OSes hardware platforms
- communication protocols and interconnections - TCP/IP, FDDI, IPX/SPX, ATM, Ethernet

Also it automates tasks like object registration, location and activation, request multiplexing/demultiplexing, error handling, parameter marshaling/demarshaling and operation dispatching.

The *Object Services* are domain independent interfaces to sets of objects which perform fundamental, domain-independent functions. Examples of Object Services:

- *Naming* - allows clients to find objects by name
- *Trader* - allows clients to find objects by the services provided
- *Event* - allows the notification of named events
- *Life Cycle* - manages the creation/destruction of objects
- *Persistence* - makes objects live longer than servants
- *Transactions* - allows the construction of atomic collection of calls
- *Concurrency Control* - allows objects to be locked by clients

The *Common Facilities* and *Domain Interfaces* are services which are also horizontally linked like Object Services, but they are oriented towards user applications respectively specific application domains. The *Application Interfaces* are interfaces developed specifically for a given application and thus not standardized.

2.2 The Object Model

For the specification of the OMA, OMG has defined an *Object Model*. The OMG Object Model defines common object semantics for specifying the externally visible characteristics of objects in a standard and implementation-independent way. This object model is an example of a classical object model, where a client sends a message(request) to an object. A message identifies an object and zero or more actual parameters. A distinguished first parameter is required, which is the operation to perform. CORBA provides a very abstract view of objects. An object is visible only through operations defined by an interface and invoked using an object reference. The *Object Reference* is an object name that reliably denotes a particular object. An *Interface* is a set of possible operations that a client is able to request from an object. It is specified in OMG *Interface Definition Language* (IDL). IDL is a declarative language and is neutral with regard to different programming languages and networks. From IDL definitions it is possible to map CORBA objects into the different programming languages. This is done by an IDL Compiler. An *Operation* is an identifiable entity that denotes a service that may be requested. The signature of an operation consists of:

- The specification of *parameters*. Parameters are characterized by mode and type. The parameters can be *in* (passed from client to server), *out* (from server to client) or *inout* (both). The type of a parameter can be *basic*, *constructed* and *object reference*. CORBA 2.3.3 introduced *value types* which try to emulate the semantic of objects passed by value.
- The specification of a *result*. The result is a distinguished out parameter.
- The specification of *exceptions* that may be raised. The exceptions may carry specific information.
- The specification of additional *contextual information*
- The indication of *execution semantics*. Initially the execution semantics were only *at-most-once* (the default request-reply blocking invocation) and *best-effort* (the *one-way* invocation without replay). The CORBA messaging specification extended the model.

An Interface can also contain *Attributes*. Attributes are logically equivalent to declaring a pair of accessor functions, one for reading the value of an attribute, one for writing it. Attributes may be *read-only*.

Concerning the three ingredients of the object-oriented paradigm, CORBA fully supports encapsulation, does not support polymorfism, and inheritance is restricted to interfaces only (since implementation is no concern). It provides no overriding of operations.

2.3 CORBA ORB Architecture

There are eight components which build the CORBA ORB architecture [12]:

ORB CORE - provides the mechanism for transparently communicating client requests to target object implementations. It simplifies distributed programming by decoupling the client from the details of method invocation,

the requests appear as local procedure calls. The ORB is responsible for finding the server objects, delivering the request and returning the result.

ORB Interface - provides various helper functions for both client and objects, like storing object references into strings or creating argument lists for dynamic invocations.

CORBA IDL stubs and skeletons - are the mediators between the client respectively server and the ORB. They are created automatically by an IDL compiler. Stubs provide the *Static Invocation Interface* (SII), which is strongly typed. The stubs receive the client's request and marshal it into transport-level format. The skeletons demarshal the request and do the upcall to the server implementation.

Dynamic Invocation Interface - if there is no knowledge at compile time of the object interface, the DII is used to directly build the request, usually with information from the Interface Repository, bypassing a stub.This is also the only way to do deferred synchronous invocations.

Dynamic Skeleton Interface - is the DII counterpart on the server side. The DSI delivers requests to an object implementation that has no compile time knowledge of the object interface it is implementing. Dynamic or static, stubs and skeletons are fully interoperable.

Object Adapter - is responsible to assist the ORB in activating objects and delivering requests to objects. It associates servants to objects and builds object references. Different kind of adapters can be used to provide for different policies for object activation, granularity, lifetime.

Interface repository - provides run-time information about IDL interfaces and also additional information associated with interfaces, for example the type library.

Implementation repository - stores information used to find and activate servants and additional information associated with servers.

2.4 ORB Interoperability

The major flaw of CORBA 1.0 was not providing a standard for interoperability between ORBs. So CORBA 2.x specified a single wire format to be used as lingua franca for bridging between *Environment Specific Inter-ORB Protocols* (ESIOP). This is the *General Inter-ORB Protocol* (GIOP) with *Internet Inter-ORB Protocol* (IIOP) being the mapping of GIOP on TCP/IP. In the end vendors chose to implement native GIOP/IIOP to prevent bridging between domains. The GIOP specification consists of the specification of the *Common Data Representation* (CDR), which is the formatting of data on the transport protocol, specification of the *Message Formats*, and the requirement for a connection-oriented protocol.

3 Requirements for a RT CORBA System

The CORBA design principle is to hide most of the underlying, lower-level operating system and communication resources from the user, to provide a transparent way to do method invocations. This policy separates the application programmer from low level management, which makes CORBA well suited for *best-effort* quality of service requirements, but not for RT, deterministic applications.

Schmidt et al. [14] give the following reasons why conventional CORBA implementations are not suited for real-time applications:

- *Lack of QoS specification interfaces.* Clients cannot specify end-to-end QoS requirements. There is no way for clients to indicate the relative priorities, deadlines, importance of their requests to the ORB. Or the period of their invocations. Likewise there are no interfaces for allowing applications to specify admission control policies.

- *Lack of QoS enforcement.* Conventional ORBs do not provide end-to-end QoS enforcement. Most ORBs transmit, schedule and dispatch client requests in FIFO order. FIFO strategies can lead to unbounded priority inversions, for example when a lower level request blocks a higher level request for an indefinite period of time. Likewise conventional ORBs do not allow the specification of priorities for threads that process requests and do not provide fine grained control of servant execution. Mostly they provide ad hoc allocation of resources.

- *Lack of real-time programming features.* No standard language mappings exist in CORBA 2.x to transmit client requests asynchronously and it does not support timed operation invocations.

- *Lack of performance optimization.* Conventional ORBs incur significant latency and throughput [6] overhead as well as many priority inversions and sources of non-determinism. These overheads stem from: non-optimized presentation layers that touch and copy data excessively, internal buffering strategies that produce non-uniform behavior for different message sizes, inefficient demultiplexing and dispatching algorithms, long chains of intra-ORB calls, lack of integration with underlying operating system (OS) and network QoS mechanisms.

In a later paper, Schmidt et.al. evaluate the real-time performance of several ORBs (MT-Orbix, CORBAplus, miniCOOL and TAO) by different benchmarks. They conclude with a list of recommendations, on how non-determinism can be decreased and priority inversion limited [15]. We will return to this in section 4.1.

3.1 Requirements for Real-Time ORB Endsystems

For a CORBA System to be suited for real-time applications it is necessary to let the client *specify* the QoS needed and that the ORB and underlying system are able to *enforce* the QoS parameters specified by the applications.

Wolfe et al. [19] outline different requirements for a RT CORBA System to meet the needs of real-time applications QoS. These requirements fall under

two main categories: Requirements on the operating environment (operating system and networks) and requirements on the CORBA run-time system. We will enumerate them as they appear in the article [19].

Operating environment requirements:

- *Synchronized Clocks.* All the clocks on nodes in a domain should be synchronized within a bounded skew of each other.
- *Bounded Message Delay.* The underlying communication mechanism should ensure a worst-case message delay between the nodes.
- *Priority based Operating Environment Scheduling and Queuing.* All components used in the underlying CORBA environment should support priory based scheduling and queuing. this scheduling should be pre-emptive where possible.
- *Operating Environment Priority Inheritance.* All components used in the CORBA environment that synchronize tasks by blocking one task for another should implement priority inheritance.

CORBA Run-time System requirements:

- *Time Type.* CORBA standard should specify a standard type for absolute time and relative time.
- *Transmittal of Real-time Method Invocation Information.* RT-CORBA should allow the following information to be transmitted from the client to the server together with the method invocation: *deadline, importance, earliest start time, latest start time, period* and other QoS requirements. This information is likely to be needed to enforce real-time constraints in the server node.
- *Global Priority.* The ORB should establish priorities for all executions. These should be *global* across the ORB. This means that priorities should be set in a consistent way, so they *make sense* relative to each other.
- *Priority queuing of all CORBA Services.*
- *Real-time Events.*
- *Priority Inheritance.* All RT CORBA level software that queues one task while the other is executing should use priority inheritance, or a form of priority ceiling protocol.
- *Real-Time Exceptions.* The CORBA exception handling mechanism should be extended to include Real-time exceptions (e.g. missed deadline).
- *Documented Execution Times.* The standard should specify that vendors must publish worst case bounds for their products.
- *ORB Guarantee.* If a client specifies a certain QoS the ORB should be able to either guarantee it or raise an exception.

Likewise, Schmidt et.al., in addition to similar requirements as the ones presented before, identify the following requirements for a *high performance real-time ORB* [14], which address the more low-level, implementation (optimization) features of an ORB:

- *Efficient and predictable real-time communication protocols and protocol engines.* This means a connection and a concurrency architecture that minimize priority inversion and a transport protocol that enables efficient, predictable and interoperable processing and communication across ORB endsystems.
- *Efficient and predictable demultiplexing and dispatching of the incoming requests.* This is possible by reducing the multilayered demultiplexing and dispatching scheme of conventional ORBs through techniques like active demultiplexing or perfect hashing.
- *Efficient and predictable presentation layer.* ORB presentation layer conversions transform application-level typed data into a portable format (CDR). Common case optimization can be applied.
- *Efficient and predictable memory management.* Data copying is consuming a significant amount of resources. Likewise using dynamic memory allocation produces locking overhead and can also induce non-determinism through heap fragmentation.

4 RT CORBA Development

In this section we describe two of the research projects that have provided much of the foundation on which the RT CORBA standard is based.

4.1 TAO at Washington University

The Ace ORB (TAO) is a high performance, real-time ORB developed at Washington University at St. Louis. Their objective is to find patterns and performance optimizations that provide for building a high performance, real-time ORB. This ORB was targeted to support hard real-time systems using an *a priori* static scheduling. Later the Scheduling Service was modified to suit also dynamic environments. The key components of TAO [14] are as follows.

Real-time I/O subsystem TAO's real-time I/O subsystem extends support for RT into the OS. Although some general purpose operating systems now support RT scheduling, they do not provide a RT I/O subsystem. TAO's I/O subsystem assigns priorities to RT I/O threads so that the schedulability of the application can be enforced. At the core of the I/O subsystem lies a network interface consisting of one or more ATM Port Interface Controllers.

Efficient and predictable ORB Cores To address a reliable, deterministic inter-ORB communication TAO's ORB Core supports:

- *A priority based concurrency architecture.* TAO ORB Core can be configured according to the following concurrency policies: thread/priority, thread pool, thread/connection and single thread. In the scope of these policies operations can execute with one of the following models - *Client propagated model* - at the priority at which the client invoked the operation and *Server sets model* - at the priority of the thread in the server's core that received the operation.

- *A priority based connection architecture.* The server ORB pre-establishes different connections for different levels of priority. To get a certain level of priority (thread priority) the client simply connects to the respective connection. This approach is well suited for static rate monotonic scheduling.

- *A Real-time inter-ORB protocol* (RIOP). An application that implements dynamic QoS characteristics, or which requires that priority is propagated from the client to the server, can transfer QoS information as a *Tagged Component* in the *Message Context.* ORBs that do not support RIOP simply ignore the header. In addition to this RIOP is designed to map GIOP on different networks (e.g. ATM network).

Efficient and predictable Object Adapters An Oobject Adapter (OA) demultiplexes incoming requests to servants, which have to locate the right skeleton, which has to find the right operation to call. This layered demultiplexing can lead to priority inversions. TAO's RTOA uses perfect hashing and active demultiplexing [11] to dispatch operations in constant time regardless of number of servants, skeletons, and operations defined in IDL interfaces.

Efficient Stubs and Skeletons and Memory Management Stubs and skeletons are responsible with marshaling and demarshaling of typed operation parameters, an activity which is a major bottleneck for high-performance systems. TAO optimizes performance by reduced use of dynamic memory (problematic due to heap fragmentation and locking on heap allocation), reduced data copying between ORB layers, and reduced function call overhead (inline-ing). Also there is a choice between compiled stubs and skeletons, which are faster, and interpreted ones which are smaller.

TAO's Real-time Scheduling Service The TAO real-time scheduling service was initially designed for static, offline scheduling, on a single CPU, but was later extended to permit distributed, dynamic scheduling. We will briefly explain the features of the Scheduling Service as described in the literature [5, 14].

TAO uses the concept of *RT_Operation* to identify an operation defined in CORBA IDL which has its own timing requirements specified in terms of the attributes of a *RT_Info* IDL struct. The attributes contained in the RT_Info specify information like *worst-case execution time*, the *period* of this operation invocation, *criticality, importance*, and *dependency* on other operations.

The off-line part of the Scheduling Service stores the RT_Info in a repository of RT_Info descriptors to be used at run-time. Then it constructs operation dependency graphs (from the dependency information of RT_Info) and identifies threads of execution by examining the terminal nodes of the graph. Each

thread is identified by an active RT_Operation, which means an RT_Operation that does not depend on another RT_Operation call to be executed. Then it calculates each thread's execution time as a sum of all traversed operations and the threads period as the minimum period of all non-zero periods of the traversed operations. Based on this information and on the scheduling policy used, it assesses schedulability for the thread set. Initially the *Dependency_Info* and the *Execution_Time* for each operation had to be gathered manually but the Scheduling Service was extended to allow configuration runs to gather this information and populate the RT_Info structure.

At run-time, on the server side, the service uses two mechanisms to provide schedulability. First, it sets up a number of queues which are used for dispatching operations and the priority of the thread which serves each queue, second, it determines the dispatching prioritisation of each queue. By different ways of using the information from the RT_Info structure to assign operations to different queues and their position inside the queue, different scheduling policies can be implemented. Mapping for rate-monotonic, earliest deadline first, minimum laxity first and maximum utility first are presented.

TAO also provides a RT Event Service which can be used in conjunction with the Scheduling Service to determine dispatching ordering and preemption strategies.

4.2 RT CORBA at University of Rhode Island

Researchers at University of Rhode Island have developed a prototype of RT CORBA that is designed to support expression and enforcement of dynamic end-to-end timing constraints within a CORBA system. It shows a *best-effort* behavior and uses an *earliest deadline first within importance* scheduling scheme. The system was built as an augmentation to Orbix from Iona Technologies. The components of the real-time CORBA system are implemented as a *Real-Time Daemon* process and library code. The RT Daemon coordinates dynamic aspects of the system as changing global priorities, time synchronization and supporting RT Events.The Key Components of the system [20] are:

Global Time Service For expressing timing constraints in a distributed system, a common global notion of time must be supported. Clients and servers can call it to get the current time.

Latency Service This service allows clients and servers to determine various latency bounds in the system. For example, a specification of 98% means that the returned latency is greater than the actual latency 98% of the time. The Latency Service uses three methods to provide latency bounds: *Estimated Latencies* - which are a priori measurements and very fast; *Measured Latencies* - the Service measures the latency by background calls and requires high overhead; *Analytical Latencies* - the server uses network parameters (like SNMP) to calculate latencies.

Specification of Real-Time Constraints In a *Timed Distributed Method Invocation* (TDMI) a client expresses real-time constraints on a CORBA method invocation as attributes of a *RT_Environment* structure. A RT_Environment

structure contains attributes that include *importance, deadline, period* and is specified in IDL. The RT CORBA run-time system attaches the RT_Environment structure to all executions that result from client's TDMI request. For example the RT_Environment structure is sent as a last argument on a remote operation invocation. Also URI RT CORBA augments the CORBA exception mechanism to handle real-time exceptions (e.g. it raises an exception of the type RT_Exception when the TDMI has not returned before its deadline).

Global Priority Service and Distributed Scheduling The dynamic schedulling is done by establishing a global priority for all executions in the system. Each client communicates its RT parameters to the *Global Priority Service* and receives a *global dynamic priority*. The priority is dynamic because it can change during the lifetime of the activity. For example when a TDMI arrives on a server the deadline for the execution on the sever should pessimistically allow the message delivery from the server to client and the clock skew between nodes. The GPS uses an *Earliest Deadline First Within Importance* algorithm to assign priorities. The RT-Daemon on each node maps the dynamic priority onto local OS priority. Additionally the RT-Daemon enforces *aging* of global priorities so that it remains consistent with the EDF scheduling.

Real-Time Event Service Real-time events are important in a distributed system as means for synchronization and enforcement of real-time constraints. This RT CORBA system implements a *RT Event service* that delivers the time when events occurred and prioritizes the delivery of events depending on the global priority of the producer, or of the consumer, or of both.

Real-Time Concurrency Control Service CORBA 2.x provides a Concurrency Control Service to control consistent access to distributed resources. This system includes a Real-Time Concurrency Control Service that implements *priority inheritance*. When a TDMI holds a lock and blocks a higher priority TDMI it inherits the latter priority. Transitive blocking is avoided by not allowing a child activity to be created under a lock, which means that an activity can only hold locks on one resource at a time.

5 Real-Time CORBA Specification

5.1 RT-CORBA Specification 1.0

The Final Joint Revised Submission to RT CORBA 1.0 [2] was received by OMG in March 1999 and is part of the new CORBA 3.0 standard which will be available in 2000. RT CORBA is defined as an extension to CORBA 2.2 (formal/98-12-01) and Messaging Service specification (orbos/98-05-05). We will recognize many ideas from the systems presented earlier.

The goal of the specification is to support meeting real-time requirements by facilitating end-to-end predictability of activities in the system and by providing support for management of resources. The specification is sufficiently general to span hard and soft real-time, but does not currently address dynamic scheduling. Also, RT CORBA assumes fixed priority based scheduling in the underlying

operating system. The thread as provided by the underlying OS represents the *schedulable entity.*

Activity An *activity* is the design concept used for the end-to-end flow of information between client and server and should be *end-to-end predictable.* But this abstract entity is represented by concrete units: the running thread on the client, the request in memory, the message within the transport protocol, the thread in the server and back. So, in order to assure the end-to-end predictability, RT-CORBA defines standard interfaces and policies to manage:

- processing resources
- communication resources
- memory resources

In the remaining part we will make a short overview of the specification. A more detailed presentation can be found elsewhere [2, 13].

Priority System RT CORBA allows the application to determine the priority of its threads. At any time a thread has a global CORBA priority and a native priority which is the mapping of the global priority on different operating systems. The RT-ORB also offers an interface for mapping global to native and back.

Priority Model The priority model offers two policies of specifying the priority according to which an invocation is processed on the server. The first way is *client-propagated*, that means the priority is transported from the client to server in the message context of invocation. The second is *server-declared*, the server object specifies the priority at which it processes client requests, the priority is specified for all the methods provided by the server and is exported in the *Interoperable Object Reference* (IOR) of the server.

Thread Pools Through the pre-allocation of threads, programmers can reduce priority inversions by ensuring that there are enough threads and by avoiding dynamic thread creation/destruction. Server applications can specify the number of static threads (created initially), the maximum number of dynamic threads, and the default priority of the threads. An Object Adapter is associated only with one thread pool so it might be useful to have threads at different priorities in a thread pool. That's why thread pools with lanes were specified. For each lane the number of static and dynamic threads and their priority can be specified. Also thread pools can specify the number or size of the requests to be buffered if there are no threads available.

RT CORBA Mutexes To ensure a consistent resource allocation, RT CORBA specifies a locality constrained *mutex* interface so that the applications can use the same mutex implementation as the ORB. The mutex implementation has to offer simple priority inheritance or some form of priority ceiling protocol.

Explicit Binding In CORBA 2.x the application had no control over the connections between clients and servers. Now there are two mechanisms available. First, clients are allowed to pre-establish connections. *Priority banded connections* allow clients to specify explicit priorities (bands) for each network connection and to select the appropriate connection based on the thread that made the request. Second, clients are allowed to specify a connection as *private connection*, it means that the requests sent on that connection are not multiplexed, and a second request has to wait for the reply for the first.

Selecting and Configuring Protocols RT CORBA offers two policies to specify the communication protocol used between clients and servers. Using the *ServerProtocolPolicy* the server specifies what kind of protocols are available and different protocol attributes. The server exports its protocol preferences inside its IOR. When a client establishes a connection it uses the *ClientProtocolPolicy*. When no ClientProtocolPolicy is available the ORB chooses a protocol depending on the protocols from the server IOR and the protocols supported by the client ORB.

Invocation Timeouts Developers can specify timeouts on invocations to bound the time a client is blocked in waiting for a reply. It uses one of the timeouts specified by the Messaging Specification.

Scheduling Service RT CORBA 1.0 also specifies a global scheduling service. The service provides an abstraction layer to hide the RT CORBA scheduling parameters. It uses *names* for objects and activities. At run-time the application uses the scheduling service by acting on these names. What are the corresponding low level parameters and how to coordinate these named activities and objects is determined at the design time of the scheduling service.

5.2 Dynamic Scheduling

The Dynamic Scheduling Specification is intended to define the Real-Time CORBA 1.0 fixed priority extensions that are required for a system governed by a dynamic scheduling doctrine. This subsection is based on the initial submission received by OMG [1]. A revised submission has been recently received.

Scheduling doctrines which define thread eligibility as a function of both *thread characteristics* (e.g. fixed priority) and *current system characteristics* (e.g. loading) are regarded as dynamic, and extensions to the RT CORBA 1.0 are required to implement them. This submission presents techniques which are usable without modification of the current generation of real-time OS priority based dispatching systems.

The approach to dynamic scheduling is the following: To select which thread is to be dispatched, the RTOS considers the *priority* of the thread and whether it is *ready-to-run*. Activities waiting for IO, held in dispatch or mutex related queues, or waiting for synchronous messages to complete are not ready-to-run. The dynamic scheduling extensions allow a *Dynamic Scheduler* to manipulate when threads are or are not ready-to-run. The specification defines:

- an interface between the ORB and the scheduler which is used by the ORB to inform the scheduler about request/reply message processing

- a pluggable module for the ORB that permits a "Dynamic Scheduler" object to provide the queuing functions used within the ORB
- a user extendible scheduling parameter to be associated with the *schedulable entity* (the thread) in the same manner CORBA 1.0 associates priority with threads

The *Scheduler::PermissionToProceed* interface provides methods for applications and ORB components to both inform the scheduler about loading and scheduling parameter characteristics and for the scheduler to control the execution progress.

The methods intended for usage by the ORB require the ORB to pass to the scheduler information related to the servant and method being targeted, and the scheduling parameter associated with the request. Policies associated with the request specify whether the ORB should ask permission during:

- client request processing
- during server up-call processing
- not at all

For example, by blocking the request on the client side, the usage of the transport mechanism can be managed.

The methods intended to be used by applications provide the scheduler with information related to activity workload demand and scheduling parameters. Since the calls to this interface are synchronous the scheduler can use them to control the runability of the calling threads.

The *Scheduler::QueueServices* interface provides an interface for an *ORB plug-in* for supporting pushing and popping requests and activities (threads) into a queue provided and managed by the scheduler. It can be used either when resources are unavailable at ORB request up-call or at mutex processing. Application can use the interface also for other synchronization purposes.

The scheduler has three processing phases:

- acquiring information about the system by ORB and applications calling operations on the two interfaces,
- determine an activity dispatch schedule,
- affecting which threads are available for the RTOS to dispatch, by responding to PermissionToProceed or by popping requests or threads out of the queue.

This means that the scheduler is able to perform its scheduling activities at the frequency at which the applications use the two provided Interfaces. The proposal also discusses implementation possibilities for some scheduling doctrines (e.g. rate monotonic, earliest deadline first, least laxity first, maximum accrued utility).

5.3 Remarks about the specification

The goal of CORBA is to provide a high-level environment for the programmer, which is able to make transparent method invocations without concern for location, operating system, hardware and communication mechanism. By providing the developer with handles on managing resources and on enforcing predictability, RT-CORBA sacrifices some of the general purpose character of CORBA by supporting development of RT-Systems.

Also we can see that there are no IDL definitions for entities like RT activities or QoS parameters, which means that this specification offers the developer the possibility to enforce RT constraints, but it puts no restriction on how it is done.

O'Ryan et al.[10] evaluate the implementation of TAO with respect to CORBA 2.x and CORBA 3.0 specifications. Besides the interesting configuration and benchmarking results, some shortcomings of CORBA 3.0 are also revealed:

- There is no standard API to control how thread pools are associated with explicitly bound connections
- There is no standard API that would allow a server application to control how the ORB associates a thread of a pre-specified priority to read from I/O. Thus the thread that performs I/O could be different of the thread processing the request which could lead to priority inversions.

In that paper they present the way these shortcomings were treated in the TAO ORB, as well as an extension for specifying policies for controlling communication protocol buffer limits and flushing.

6 Application-level QoS

RT CORBA 1.0 recognized the need to let the application developer specify non-functional, low-level requirements to the RT CORBA system. The positive side is that now CORBA allows for end-to-end predictability. The negative side is that programmers have to go low-level again, assigning all the needed resource control parameters themselves. In a way this is solved by the RT-CORBA Scheduling Service, which offers the use of *names* for activities and real time objects. How the Scheduling Service interprets these names and acts on the low-level parameters is defined at design time. But this may lead to *inflexible systems* and to *less reusable code* because of the tight binding between the application and its scheduler.

In order to fully support fundamental middleware/CORBA requirements like:

- high-level programming which fast development cycles and
- reusability of code, use of *commercial off the shelf* (COTS)

there is a need for an architecture in which the programmers of different components are able to specify also *non-functional requirements - QoS* in a more abstract, *application-centric* way. To make the code reusable the architecture has to provide a consistent way to specify and enforce the different QoS requirements. Among others, this means a uniform translation of application level QoS into operating system and network level parameters.

Another aspect of interest is *adaptability*, especially in open, dynamic environments. RT QoS applications are usually fragile, since they depend so much on the environment conditions. This means they are not suited for different environments or for environments where system conditions vary a lot. In order to avoid this, components should adopt an adaptive behavior. They should take into account changes in the environment and have some knowledge about their structure, to be able to adapt at run-time and provide the needed QoS. In the rest of this section we will present two approaches to solve this problem.

6.1 Quality of Objects Architecture

The *Quality of Objects* (QuO) architecture is a CORBA QoS architecture developed at *BBN Systems and Technology*, and described by Zinky et.al. [22]. The goal of QuO is to let the application specify the desired QoS in an application-centered way, and by monitoring system and environment conditions to adapt in the best way to deliver the expected QoS.

In QuO the object is the entity responsible for guaranteeing the end-to-end QoS. A part of the object's implementation is moved into the client's address space. Thus, the distributed information needed for providing QoS is gathered and processed on the client node by the *object delegate* (client side proxy). The advantages of this solution is that there is almost no delay between object delegate and client. Also the delegate cannot fail independent of the client.

6.1.1 Specifying QoS in QuO

The specification of QoS is done with help of a set of IDL-like languages, which together form the *Quality Description Language* (QDL), as follows.

The Contract Description Language CDL is used to define the expected usage patterns and the QoS requirements an object should obey. A *contract* is defined in terms of different level QoS regions of operation. To help the application adapt to different system conditions QuO supports multiple behaviors for a functional interface, each of these behaviors is bound to a region. The regions of operation are specified as a predicate over multiple system properties. In this way the client and object have to adapt only when there is a transition between regions, and not every time a system property changes. A *negotiated region* is defined in terms of the system conditions in which both client and server will try to operate. Different negotiated regions correspond to different modes of operation. Within a negotiated region there may be many *reality regions* which are defined in terms of client usage and object delivered QoS that are actually measured by the QuO runtime system. The reality regions are the ones that specify which actions need to be taken to keep the client-server as long as possible in the chosen negotiated region.

The Structure Description Language SDL defines the internal structure of an Object and its delegates on the client side. SDL is a data-flow description of the object. Each method of an object would have its own SDL description. Based on the SDL, at run-time different paths through the object may be taken to provide the necessary QoS.

The Connection Setup Language CSL is used to specify the components of a QuO application and how they are instantiated, initialized and connected with each other [8].

The Real-time Specification Language RSL is part of the QDL family, and has been developed to deal with application-level, real-time requirements in the case of periodic sensor/actuator systems [8].

6.1.2 Monitoring and adapting

An *object delegate* is not a monolithic object, it is built by a web of sub-objects. Some of the objects implement the *functional interface*. Others implement the contract, these are *region objects* and implement the object's behavior in the different reality regions. And a third category implement system properties, the *system condition objects* are objects who provide uniform interfaces to different system resources, mechanisms and managers.

Also, the object delegate on the client side is composed of layers of delegate objects, so each layer is able to mask out some variance of system conditions for the layer above. In this way the client implementation deals only with high-level, application-centric system conditions.

Each layer of delegate objects exports a negotiated region to the layer above. It uses different techniques to mask changing conditions (e.g. by changing policies, changing processing algorithms) and maintain the QoS it provides for the layer above. When it cannot do it anymore it propagates information upward to signal a change in the reality region. Each party tries to adapt and if it cannot it indicates a change in expectations. This triggers a renegotiation of the negotiated region between the client and the object.

There are *observed* and *non-observed* [8] system condition objects. Value changes in the observed system conditions automatically trigger contract reevaluation, which could result in an active region change. The non-observed ones do not propagate a change in their state but provide their value on demand, for example when a contract is reevaluated due to a method call or an event from a observed system condition.

At run-time, QuO provides a Run-Time Kernel that coordinates contract reevaluation and provides run-time QuO services.

The design of QuO also intends to provide reusability and automatic generation of the code. By providing a layered architecture for object delegates, clients can use high-level system conditions. This should improve the reusability of the client code. QuO also offers system condition object from an already implemented collection. In the same way as an IDL compiler, a *QDL compiler* would generate a big part of the code used in object delegates.

6.2 ERDOS QoS Architecture

Sydir et al. [16] describe a QoS driven resource management scheme called *End-to-End Resource Management For Distributed Systems* (ERDoS) for supporting end-to-end QoS in a CORBA environment.

In order to provide a complete QoS driven middleware layer, the ERDoS system incorporates a *QoS-driven resource manager* into the CORBA ORB and Object Adapter. It includes algorithms to:

- decompose end-to-end application-oriented QoS into middleware, OS and network parameters. In supporting this task they defined a *QoS taxonomy* to capture the various facets of QoS in a consistent manner. QoS parameters are expressed in terms of different units at different layers of the system,
- allocate computing, storage and communication resources to applications,
- schedule application on these shared resources,
- gracefully degrade applications QoS when load exceeds capacity.

For capturing the information needed by the previously presented algorithms ERDoS uses the following models:

The Resource Model captures information about the individual resources. Resources represent the smallest grouping of hardware over which the resource manager has control.

The System Model captures information about the structure of the system. It describes the layout of the system resources by using a hierarchical structure. (e.g. different subsystems can be managed by different management schemes)

The Logical Application Stream Model (LASM) captures the structure of the application. The LASM can be represented as a graph in which the nodes represent application components and the edges represent the data flow. Each component represents a service which transforms the functional data and its associated QoS. The end-to-end QoS of the application is a composition of these individual QoS transformations. A service can be realized by one ore more *Logical Realizations of Service* (LRoS) or *Logical Units of Work* (LUoW). Each LRoS is represented also by a graph of data flow through lower-level services which are realized by other LRoSs and LUoWs. The LUoW is a piece of application which is realized entirely on a single resource, and when it is instanced on a resource it is referred as a *Physical Unit of Work* (PUoW) and has a *resource demand model* attached. The resource demand model describes the resource usage requirements depending on the QoS of incoming data and the desired outgoing QoS. At run-time the resource manager *composes* the application by choosing which LRoSs and PUoWs should realize a service.

The Application Invocation Model (AIM) contains the information that the user supplies when invoking a QoS application. It includes the desired values of the QoS parameters, and information on their relative importance in the form of a *benefit function*. The system uses this information to choose the QoS which maximizes benefit and to provide *graceful degradation* when resources are not sufficient.

At run-time a client of a QoS application invokes a top level LRoS, which represents the entire end-to-end application, and provides it with the AIM parameters. Then, based on the LASM information of the available LRoSs and LUoWs, the ORB composes the application, allocates its PUoW to resources

and schedules them in order to provide the best QoS achievable. In the composition phase the ORB communicates with the potential PUoW through a negotiation interface and chooses the ones which provide the better aggregated QoS. Basically this interface provides two negotiation functions: one gets an *output_qos* based on an *input_qos* and a *resource_demand*, the other gets a *resource_demand* based on an *input_qos* and an *output_qos*.

ERDoS also extends IDL to capture the LASM of the different LRoS and LUoW available in the system, information which is stored in the implementation repository and is used at run-time to compose the application. Also the *ERDoS IDL-compiler* generates code for stubs, wrappers and skeletons for the different PUoW used.

The authors also relate their work to BBN's QuO and emphasize on the fact that this solution addresses the QoS problem for entire end-to-end server applications, not only for a client-server connection, and that a central manager for all the activities in the system is more suited to enforce a consistent scheduling policy.

6.3 Comparing QuO and ERDoS

The big difference between these two systems is the localization of the QoS decision maker. On one side we have ERDoS centralized global resource manager, who has to find a suitable scheme for all the QoS applications in the system, applications which have to comply with the manager. On the other side we have QuO clients and objects, the decision making is per application basis and is taken locally on the client node. The client can provide decision making to adapt to the currently measured conditions and the QuO solution is non-obtrusive to the ORB.

It seems that ERDoS is more suited for somehow closed systems, inside an ERDoS ORB domain, with a centralized QoS scheme that provides better coordination between services. QuO is more suitable for large, decentralized domains where local adaptability is the answer.

7 Other approaches

In this section we describe a few other approaches to real-time or QoS treatment. In particular, we briefly describe the *interceptor* approach mentioned in the introduction to the report [9].

The interceptor approach can be seen as a more general approach than the ones described so far (but also a more low-level implementation-deriven approach). It is more general in the sense that several "deficiencies" of current CORBA systems can be overcome by interceptors: non-application components which can alter the behaviour of the application without changing the code or the code of the ORB. For example, Narasimhan et.al. propose that interceptors be used not only to achieve real-time scheduling, but also for several other purposes: profiling and monitoring, as protocol adaptors for protocols other than IIOP, for enhancement of security, and fault-tolerance via object replication. Paradoxically, the interceptor proposed for scheduling purposes (which is supposed to influence the time at which a method for an object is executed) is nothing more than a library interpositioning mechanism for thread creation,

thread release, and thread management via existing operating systems such as Solaris or other POSIX based operating systems.

At University of Rhode Island recent research involves a static scheduling method for their RT CORBA system, with deadline monotonic global priority assignment and the Distributed Priority Scheduling Protocol for concurrency control [4]. They present a new algorithm for mapping potentially many global priorities on few local priorities, the Lowest Overlap First Priority Mapping algorithm. They also extended PERTS to generate global and local priorities using the previous scheme.

Le Tien et al. [17] are building a "*lightweight distributed processing scheme*" to support end-to-end QoS in middleware. They do not claim to introduce new concepts in the field of QoS but want to adapt proven solutions to object environments (CORBA). Their scheme is based on two dedicated objects:

The *Micro QoS Manager* is an application dependent object, in fact it should be generated by a special IDL compiler from the applications QoS specified in IDL. It is responsible for resource reservation, QoS monitoring and renegotiation. It creates the QoS contract which represents the *mapping* of QoS as specified by the application in parameters, and understood by the *The Object Resource manager* (OR-Manager) and a *resource graph* which represents the resource processing order. The OR-Manager is an object which controls a physical resource. It includes policies for resource reservation, admission control and resource scheduling and takes an active part in the request processing.

End-to-end QoS management is provided by the cooperation of Micro QoS Manager with a set of OR-managers. In the admission phase the client sends the request parameters to the associated Micro QoS Manager. The Micro QoS Manager builds the contract, and finds a OR-managers which can provide the needed QoS for each of the elements in the resource graph. In the processing phase the Micro QoS Manager provides monitoring of OR managers (can lead to dynamic renegotiation). The developing of the system seems to be in a fairly incipient phase.

Kalogeraki et al. [7] present a scheduling algorithm to support the execution of tasks in a soft real-time distributed object system. The System Model consists of a Global Scheduler which is responsible for computing the initial laxities of the tasks based on information provided by the programmer. As new tasks enter the system the Global Scheduler uses current system information collected by a Resource Manager to distribute objects onto processors in order to provide a balanced load on the processors, migrating objects when necessary. A task consists of a sequence of object invocations. The laxity of a task is carried from one processor to another with the object invocation. A Local Scheduler is responsible for maintaining a local ordered list (schedule) for the objects. The scheduling algorithm uses the *laxity* of the task and the *importance* of the object to the task. It schedules a more important object before a less important one if the laxity indicates that both will meet their deadlines. The authors emphasize that the usage of laxity and importance allows for a system-wide scheduling strategy that requires only local computations, and that a low importance object does not delay a high importance task. However it is not quite clear how the local schedule list is reordered with the arrival of a new task, and which objects are dropped in overload cases. A profiler on each processor measures the execution times(for a better laxity calculation) and resource utilization and supplies this information to the Resource Manager.

8 Current research directions

Loyall et al. [8] present the collaboration between the TAO and the QuO projects to provide *a "top-down" adaptable policy-driven perspective coupled with a "bottom-up" real-time driven perspective.* The enhancements that QuO brings to the TAO RT ORB are twofold:

First, the applications can specify higher level aspects of QoS (timeliness) requirements, and also can specify tradeoffs between requirements. These are mapped into lower level requirements and mechanisms, which are enforced by TAO's real-time infrastructure.

Second, QuO provides application-adaptive mechanisms to the system. For example, the scheduling in the system is performed on a global basis conforming to an a priori chosen policy, which can be changed only with time-costly mode-changes. But individual applications may want to react to changing conditions, for example they may want to yield some resources, or they may want to find alternative solutions.

As an example is given TAO's Real-time Event Service which has QuO delegates inserted between suppliers and consumers. The delegates can change the frequency, priority and type of events. Since the Event Service is used in scheduling this can provide adaptation without reconfiguring the enforcement mechanisms. This is also presented as a way to implement *migration of processing.* Delegates on the supplier or on the consumer side can relieve the other side by taking over some of the data processing.

In our opinion little work has been done to fully evaluate the suggested approaches by performing analysis, i.e. to illustrate that the goals set out were achieved. In particular, it is not clear what are the overheads associated with the additional features to achieve real-time or QoS guarantees (to the extent that it can be provided by the architecture). Neither is it determined which trade-offs exist in the choice of one scheduling or resource management mechanism against another. Our work plans to study the performance and utilisation trade-offs when achiving real-time or a given level of QoS – in particular, in the context of the three main approaches mentioned in the report.

Another major track of activity is to study the real-time and fault-tolerance trade-off when both services are added. This could also be extended to real-time and security trade-offs.

9 Conclusion

The goal of this report was to provide a short overview over problems encountered when incorporating real-time into CORBA-based applications, and some solutions to overcome these.

We began by identifying the characteristics of a CORBA architecture with support for end-to-end predictability, and two of the CORBA RT systems which provide some solutions for implementing these requirements.

In the current specification of RT CORBA, OMG provided interfaces by which programmers can specify low-level requirements for OS and network resources. They chose not to specify IDL extensions for RT specification to give the developers freedom about the way to implement the desired behavior. As additional help the specification defines the interface for a Scheduling Service

which can be constructed to hide the low level parameters behind "names" for real-time activities and objects.

While this may be enough for closed, static systems it is not enough for large dynamic systems in which different application co-exist and system conditions can have a high variation. The solution is to provide frameworks which let the client specify the non-functional (QoS) requirements in a high-level, application-centric way. To be feasible, the mapping of these QoS requirements into low-level parameters has to be done in a consistent, well-defined way. The solutions oscillate between local adaptability schemes and centralized system-wide managers for enforcing the needed QoS. Two of the solutions presented offer a comprehensive QoS taxonomy, the measurements of different parameters to provide adaptation, and a layered architecture from high level to low level.

The presented solutions have to be regarded from the perspective of reusability of code, use of commercially available components (COTS), quick development of software by shielding the programmer from low-level mechanisms, requirements which motivated the development of middleware. Though CORBA 2.x offered a uniform way to specify functional requirements, this was not enough for some applications to which the timeliness of the result is as important as its value. So, RT CORBA specifications provide support for enforcing non-functional requirements. However, the specification does not provide a common, high-level way to specify them. Therefore systems like QuO and ERDoS are being built with the aim to let the programmer specify non-functional requirements as easily and at the same high level as the functional ones.

References

[1] Dynamic Scheduling Initial Submission. OMG Work In Progress (orbos/99-10-06), March 1999.

[2] Real-Time CORBA. OMG Recently Adopted Specifications (orbos/99-02-12), March 1999. Revision 1.0.

[3] The Common Object Request Broker: Architecture and Specification. OMG Formal Documentation (formal/99-10-07), October 1999. Revision 2.3.1.

[4] Lisa Cingiser DiPippo, Victor Fay Wolfe, Levon Esibov, Gregory Cooper, Ramachandra Bethmangalkar, Russel Johnston, Bhavani Thuraisingham, and John Mauer. Scheduling and Priority Mapping for Static Real-Time Middleware. *Real-Time Systems*, 2000, to appear.

[5] Christopher D. Gill, David L. Levine, and Douglas C. Schmidt. The Design and Performance of a Real-Time CORBA Scheduling Service. *Real-Time Systems*, 2000, to appear.

[6] Annirudha Gokhale and Douglas Schmidt. Measuring and Optimising CORBA Latency and Scalability Over High-Speed Networks. *IEEE Transactions on Computers*, 47(4):391–413, April 1998.

[7] V. Kalogeraki, P.M. Melliar-Smith, and L. E. Moser. Dynamic Scheduling for Soft Real-Time Distributed Object Systems. In *Proceedings of the Third*

IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC), 2000.

[8] Joseph P. Loyall, Alia K. Atlas, Richard E. Schantz, Christopher D. Gill, David L. Levine, Carlos O'Ryan, and Douglas C. Schmidt. Flexible and Adaptive Control of Real-Time Distributed Object Computing Middleware. *Real-Time Systems*, 2000, submitted to.

[9] P. Narasimhan, L. E. Moser, and P.M. Melliar-Smith. Using Interceptors to Enhance CORBA. *IEEE Computer*, pages 62–68, July 1999.

[10] Carlos O'Ryan, Douglas C. Schmidt, Fred Kuhns, Marina Spivak, Jeff Parsons, Irfan Pyaraly, and David L. Levine. Evaluating Policies and Mechanisms for supporting Embedded, Real-Time Applications with CORBA 3.0. In *Proceedings to the Sixth IEEE Real-Time Technology and Applications Symposium*, May 2000.

[11] Irfan Pyarali and Douglas C. Schmidt. An Overview of the CORBA Portable Object Adapter. *ACM Standard View*, 6, 1998.

[12] Douglas C. Schmidt. Overview of CORBA. http://www.cs.wustl.edu/~schmidt/corba-overview.html.

[13] Douglas C. Schmidt and Fred Kuhns. *An Overview of the Real-Time CORBA Specification.* IEEE Computer special issue on Object Oriented Real-Time Distributed Computing. June 2000.

[14] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee. The Design of the TAO Real-Time Object Request Broker. *Computer Communications*, 21(4):294–324, April 1998.

[15] Douglas C. Schmidt, Sumedh Mungee, Sergio Flores-Gaitan, and Aniruddha Gokhale. Alleviating Priority Inversion and Non-determinism in Real-time CORBA ORB Architectures. In *Proceedings of the Fourth IEEE Real-Time Technology and Applications Symposium (RTAS), Denver, Colorado*, June 1998.

[16] Jaroslaw J. Sydir, Saurav Chatterjee, and Bikash Sabata. Providing End-to-End QoS Assurances in a CORBA-Based System. In *Proceedings of the First IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, April 1998.

[17] Didier Le Tien, Olivier Villin, and Christian Bac. CORBA Application Tailored Manager for Quality of Service Support. In *Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, 2000.

[18] Real time Platform Special Interest Group of OMG. Real-time CORBA - A Whitepaper - Issue 1.0. http://www.omg.org/homepages/realtime/real-time_whitepapers.html, December 1996.

[19] Victor Fay Wolfe, Lisa Cingiser DiPippo, Roman Ginis, Michael Squadrito, Steven Wohlever, Igor Zykh, and Russell Johnston. Real-Time CORBA. In *Proceedings of the Real-Time Technology and Applications Symposium*, June 1997.

[20] Victor Fay Wolfe, Lisa Cingiser DiPippo, Roman Ginis, Michael Squadrito, Steven Wohlever, Igor Zykh, and Russell Johnston. Expressing and Enforcing Timing Constraints in a Dynamic Real-Time CORBA System. *Real-Time Systems*, June 1998.

[21] Zonghua Yang and Keith Duddy. CORBA: A Platform for Distributed Object Computing. *ACM Operating Systems Review*, 30(2), April 1996.

[22] John A. Zinky, David E. Bakken, and Richard D. Schantz. Architectural Support for Quality of Service for CORBA Objects. *Theory and Practice of Object Systems*, 3(1), 1997.