Controlling a CORBA Real-time Event Channel with QuO Middleware

Craig Rodrigues
crodrigu@bbn.com
BBN Technologies
10 Moulton Street
Cambridge, MA 02138, U.S.A.

Christopher D. Gill
cdgill@cs.wustl.edu
Department of Computer Science, Washington University
St. Louis, MO 63130, U.S.A.

Abstract

Due to recent advances in the capabilities of MEMS components, it is now feasible to use commodity-off-the-shelf (COTS) middleware to develop high-performance real-time embedded systems. Such systems have stringent requirements for latency, determinism, and priority preservation. The current generation of COTS middleware does not give the application developer the ability to specify and control the level of an application's end-to-end Quality of Service (QoS), which is a requirement for many real-time embedded systems.

In this paper, we describe of how we used QuO, a Quality of Service (QoS) middleware framework developed at BBN Technologies, to control the delivery of events in the CORBA Real-Time Event Service provided by the TAO CORBA ORB. We describe how we configured QuO in an in-band configuration to control event delivery in a Real-Time Event Channel between two CORBA objects collocated in the same process. The overhead of controlling the Event Channel with QuO was measured and compared against sending events through the Event Channel with no QuO control.

We observe that QuO added latency and decreased the Event Channel's throughput when used to control the Event Channel in-band. We conclude that using QuO in an in-band configuration may be not be appropriate for some types of high-performance embedded systems. However, further research must be done to confirm if this is true, especially in more distributed systems where network latency concerns tend to dominate. In addition, further work must be done to evaluate the overhead in using QuO control mechanisms in out-of-band configurations.

Subject Areas: Quality of Service; Real-time CORBA; Distributed and Real-Time Middleware; Embedded Systems

1 Introduction

Real-time and embedded systems are increasingly being used in a wide variety of applications, such as flight avionics, weapons systems, telecommunications, and medical devices. Advances in microelectromechanical systems (MEMS) have made it increasingly feasible to manufacture low-cost micro-controllers, with more capabilities available on a single-chip node. Newer real-time embedded systems will increasingly be distributed as each sensor or actuator will have local intelligence and digital communication capability[1].

A critical capability of many embedded systems applications is the ability to provide real-time information with a known Quality of Service (QoS). Control information in such applications must be transmitted with a known delay and minimal jitter. Such data can change dynamically at any instant, so the timeliness of the data is as important as the accuracy of the measurement. An example of such an application would be the measurement of the pressure of a pipe. The current design of a real-time embedded systems commonly involves the use of low-level API's which are specific to a certain application and set of hardware and real-time operating systems (RTOS). While it is possible to implement QoS using such techniques, this approach usually leads to the design of applications which are inflexible, difficult to port to new types of hardware, and expensive to maintain.

With the increasing hardware and software capabilities of MEMS components in embedded systems it is increasingly feasible to use higher-level frameworks and middleware software, to solve some of the problems inherent in building such systems, such as QoS, fault tolerance, and security.

The Quality Objects (QuO) group at BBN is currently developing such a framework. QuO was originally designed to facilitate the development of complex distributed applications over Wide Area Networks (WAN)[2]. The QuO architecture is currently being used to address such diverse problem domains such as security[3], dependability[4], and survivability[5]. QuO currently works on top of lower-level middleware based on the OMG CORBA standard[6] and also with Java Remote Method Invocation (RMI)[7].

Much effort at BBN has gone into integrating the QoS capabilities of QuO with TAO, a high performance CORBA ORB, developed at Washington University St. Louis and University of California Irvine. TAO has been optimized for high-performance computing tasks[8], and has been successfully deployed in applications with stringent performance requirements such as flight avionics [9], telecommunications[11], and medical imaging[10].

CORBA provides three methods for inter-object communication: (1) a best-effort one-way invocation model, (2) a two-way synchronous request/response model, and (3) a two-way asynchronous method invocation (AMI) model. TAO comes with an implementation of a Real-time Event Service [12], which enables more flexible modes of inter-object communication. The Event Service allows an arbitrary number of event suppliers to send event data to an arbitrary number of event consumers, with the suppliers and consumers not having any knowledge of each other. This facilitates asynchronous and group communication models. The TAO Real-time Event Service improves on the OMG's Event Service specification by adding such capabilities as: event correlation, event filtering, and real-time scheduling. The OMG Notification Service specification is an enhancement to the original OMG Event Service specification that adds event filtering and QoS, but does not specify the use of real-time scheduling [13]. The lack of real-time scheduling guarantees in the OMG Notification Service is not acceptable for many real-time applications, which require events to be transmitted within certain timing constraints. There is currently an effort underway in the OMG to combine with the Notification Service some of the new Real-Time CORBA features newly added to the CORBA specification, 2.4[6]. This effort aims to integrate Real-Time CORBA features such as priorities and scheduling.

The TAO Real-time Event Service provides an ideal inter-object communication mechanism for CORBA objects in systems with stringent performance requirements and timing constraints. This paper will focus on current research on using QuO to control event delivery in the TAO Real-time Event Service, in a configuration which could be used in real-time embedded system, *i.e.*, by collocating QuO and the Event Service within the same process address space. This paper will specifically focus on the overhead added by adding QuO adaptive control to the Event Service.

2 TAO Real-Time Event Service

2.1 Event Service Model

In the typical CORBA model, a client object has a reference to another object which it wishes to invoke a method on. This CORBA object can be local within the same address space, or remote. CORBA allows the client to invoke a method on a CORBA object in an asynchronous fashion (AMI or one-way), or it can invoke a method and then wait for a response from the object (two-way synchronous).

The Event Service acts as a mediator between CORBA objects, allowing a client with a single method invocation to supply data to one or more CORBA objects. The Event Service divides objects into *event suppliers* and *event consumers*. The Event Service has two basic models by which suppliers and consumers can communicate: the *push model* and the *pull model*. In the push model, the event suppliers "push" their events onto a CORBA object called the Event Channel. The Event Channel is then responsible for pushing events to event consumers. In the pull model, a client will try to "pull" an event from the Event Channel. This action will cause the Event Channel to try to pull an event from an event supplier. Event Channels can be configured to accept both pull and push event requests, so pull and push consumers can be combined in complex configurations called *hybrid push/pull models*. Unlike the typical CORBA model, in which objects must have a reference to every object with which they wish to communicate, the Event Service facilitates anonymous, many-to-many communication. Event suppliers do not need to have any knowledge of the consumers of their events, and vice versa. This paper will focus only on the push model of the Event Service.

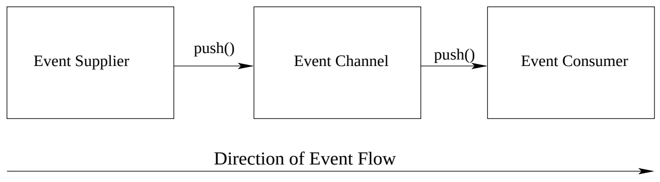

Figure 1: Event Service Push Model

2.2 CORBA interface for the Push Model

The following interface, defined in CORBA IDL, defines the interface for the push model:

```
module RtecEventComm
{
 interface PushConsumer
 {
   oneway void push (in EventSet data);
   void disconnect_push_consumer ();
 };

 interface PushSupplier
 {
   void disconnect_push_supplier ();
 };

};
```

An object that performs the role of an event consumer must implement the *PushConsumer* interface, and an event supplier must implement the *PushSupplier* interface. The consumers and suppliers connect to each other via methods defined in the *RtecEventChannelAdmin* module.

```
module RtecEventChannelAdmin
{
 interface ProxyPushSupplier:
      RtecEventComm::PushSupplier
 {

   void connect_push_consumer(
       in RtecEventComm::PushConsumer
                          push_consumer,
       in ConsumerQOS qos)
      raises (AlreadyConnected, TypeError);
 };

 interface ProxyPushConsumer:
    RtecEventComm::PushConsumer
 {

   void connect_push_supplier (
        in RtecEventComm::PushSupplier
                         push_supplier,
        in SupplierQOS qos)
      raises (AlreadyConnected);
 };

 interface ConsumerAdmin
 {
   ProxyPushSupplier
     obtain_push_supplier();
 };

 interface SupplierAdmin
 {
   ProxyPushConsumer
     obtain_push_consumer();
 };

 interface EventChannel
 {
   ConsumerAdmin for_consumers ();

   SupplierAdmin for_suppliers ();
 };
};
```

2.2.1 PushConsumer setup

An object which implements the PushConsumer interface must first acquire an object reference to an EventChannel. The reference can be obtained by such means as the CORBA Naming Service, or the reference could be a local object if the EventChannel is instantiated in the same process. The PushConsumer must then invoke the *for_consumers()* method on the EventChannel to obtain a reference to a ConsumerAdmin object. The PushConsumer must then invoke the *obtain_push_supplier()* method on the ConsumerAdmin object to obtain a reference to a ProxyPushSupplier. The PushConsumer registers itself with the EventChannel by invoking the *connect_push_consumer()* method on the ProxyPushSupplier. After this point, whenever the EventChannel has an event, it will invoke the *push()* method on the PushConsumer to supply the event.

2.2.2 PushSupplier setup

The setup sequence for a PushSupplier is very similar to that for a PushConsumer. The PushSupplier will invoke the *for_suppliers()* method on the EventChannel and obtain a SupplierAdmin object. The PushSupplier will then invoke the *obtain_push_consumer()* to obtain a ProxyPushConsumer object. Next, the *connect_push_supplier()* method on the ProxyPushConsumer object must be invoked. This will register the PushSupplier with the EventChannel. Whenever the PushSupplier has an event, it must call the push() method on the ProxyPushConsumer. This will send the event to the EventChannel, which will in turn invoke the push() methods of any PushConsumers which are attached to it.

The main difference between a PushSupplier and a PushConsumer is that a PushSupplier explicitly invokes a push() method on a ProxyPushConsumer to send an event, while the PushConsumer never directly invokes a method to receive an event. Instead, the EventChannel will invoke the PushConsumer's push() method to send it an event.

3 QuO Contract for Event Channel

The TAO Real-Time Event Channel allows a user to specify QoS parameters to the *connect_push_consumer()* and *connect_push_supplier* methods in the RtecEventChannelAdmin module. The ConsumerQOS and SupplierQOS parameters in

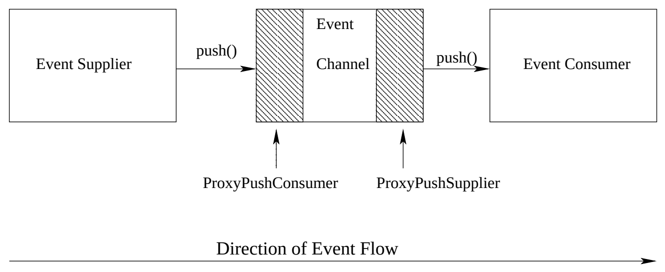

Figure 2: Event Service Push Model, showing proxy interfaces

these methods contain information which is used by the TAO Real-Time Scheduler. The Scheduler uses this information to dispatch events according to event priorities and scheduling strategies [16]. While effective for many use-cases, this implementation of QoS offers limited flexibility, since it 1) only influences the behavior of the Scheduler, and 2) is controlled at the application level. In particular, it does not have a direct interface to resource managers which could change system parameters which influence QoS, nor does it specify adaptive behaviors for the event suppliers and event consumers.

The QuO middleware framework allows an application to (1) specify its level of QoS, (2) interface with instrumentation probes which can measure QoS and system performance, (3) interface with resource managers which can tune system parameters which influence QoS, and (4) specify behaviors for adapting to QoS variations at run-time. These attributes make QuO an ideal choice for controlling QoS for a communications mechanism like the TAO Real-Time Event Channel. One possible integration of QuO with the Event Channel is illustrated in the following figure.

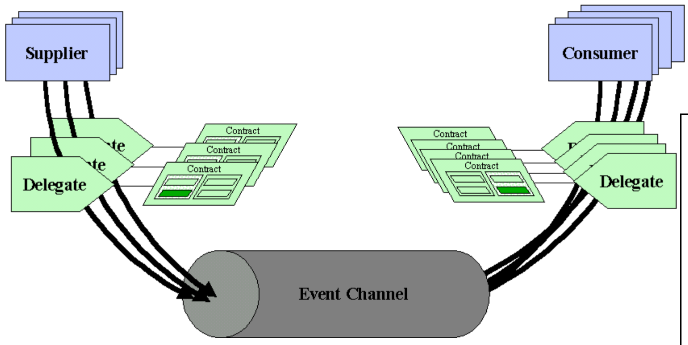

Figure 3: QuO control of the TAO Event Channel

QuO can perform both *in-band* adaptation, in which QuO inserts itself into the normal invocation path of a CORBA method, and *out-of-band* adaptation, in which it performs its evaluations outside the invocation path, and achieves an indirect effect by modifying *aspects* of the invocation path. To perform in-band adaptation, QuO defines a *delegate* object which inherits from the interface of a regular CORBA object. The *delegate* keeps a reference to the CORBA object for which it acts as a surrogate. A client then invokes a method on the delegate, as if it was the "real" object. The *delegate* can then decide whether to pass the invocation down to the object to which it has a reference, drop the invocation, or engage in some other adaptive behavior.

The *delegate* bases its decisions on information given to it by a *contract* object. The *contract* specifies a set of operating regions and conditions for the system. The state of the system at any time is determined by information given to the *contract* by *system condition objects*, which measure attributes of a system such as network or CPU load. When the system leaves one operation region and enters another, the contract will execute a transition function. The transition function can have various effects, such as changing the *in-band* behavior of the delegate, or instructing a resource manager *out-of-band* to change the allocation of system resources.

For the experiment described in Section 4, we created a delegate class for the ProxyPushConsumer. Instead of invoking a push() method on a ProxyPushConsumer and directly sending an event to the Event Channel, a PushSupplier would thus instead invoke a push() method on a QuO delegate class. Based on conditions specified in the contract, the pushed event would either be passed along to the Event Channel, or dropped.

Quality Description Languages QuO uses a set of *Quality Description Languages*[19] to describe the structure of delegates and contracts. QuO comes with a set of code generators which can convert these descriptions to C++ or Java code. The *Structure Description Language (SDL)* describes a delegate's interface and its adaptive behaviors. The SDL for the ProxyPushConsumer used for this experiment consists of the following:

```
delegate behavior for interface
 RtecEventComm::PushConsumer and
 contracts PushConsumerEC_Contract is

 quo::ValueSC event_counter
    bind with cplusplus_code { 0; };

 call push :
 region Pass:
  cplusplus_code {
     event_counter->longValue(
     event_counter->longValue() + 1);
    };
  pass_through;
 region  Drop:
  cplusplus_code {
      event_counter ->longValue (
      event_counter ->longValue () + 1) ;
  };

 default :
   pass_through ;


end delegate behavior ;
```

This SDL definition specifies that a delegate object which implements the PushConsumer interface will be created. There will be two operating regions that this delegate will respond to: Pass and Drop. For each region, a counter will be incremented each time the push() method is invoked. In the Pass region, the method invocation will then be passed down to the actual ProxyPushConsumer that this delegate has a reference to. In the Drop region, no such invocation will take place, and the event will be "dropped".

A contract which specifies the operating regions for a system is specified by a *Contract Definition Language (CDL)* definition. The following CDL definition was used in the experiment:

```
contract  PushConsumerEC_Contract (
 syscond  quo::ValueSC  ValueSCImpl
  eventCounter )

{
 region  Pass
  ( ((long)eventCounter) >= 0
   and ((long)  eventCounter  <= 500))
    { }

  transition  any->Pass {
    synchronous {
         eventCounter.longValue();
    }
  }

  region  Drop
    ( (long)eventCounter  > 500)
  { }

  transition  any->Drop {
    synchronous {
        eventCounter.longValue();
    }
  }
};
```

This contract defines two operating regions: Pass and Drop. It also defines a system condition object, event_counter. When the event_counter is between 0 and 500, the system is in the Pass region. The system is in the Drop region when the event_counter is greater than 500.

4 Description of Experiment

Placing a delegate in the method invocation path, while necessary for in-band forms of adaptation, may introduce performance overhead. Therefore, in this section we describe experiments conducted to quantify the overhead of in-band adaptation using the Event Channel and QuO configurations described in Section 2 and Section 3.

For the experiments described in this section, all tests were run on a Compaq Deskpro Proliant 550 Mhz Pentium class machine, under the Redhat Linux 6.2 operating system. The example was compiled using the TAO version 1.1.8 ORB. A pre-release version of QuO version 3 was used to develop the contract and delegate. All tests were run as the root superuser, to allow threads to run in the real-time scheduling class.

In the TAO software distribution, there is a test program, `ECT_Throughput`, to test the throughput of an Event Channel with a PushSupplier and a PushConsumer collocated in the same process. This example was modified to provide QuO adaptive control. In the unmodified example, the PushSupplier acquired a reference to a ProxyPushConsumer, and invoked this object's push() method to send an event. This in turn dispatched the event to the Event Channel, which then transmitted the event to the PushConsumer. The `ECT_Throughput` test program measured the latency in sending the message to the Event Channel, and the latency in sending the event from the Event Channel to the PushConsumer. This test was designed to measure the overhead of sending an event through the ORB, versus passing the event through a local C++ function.

In the modified version of the example, the PushSupplier acquired a reference to a QuO delegate version of the ProxyPushConsumer. To send an event, the PushSupplier invoked the push() method on the QuO delegate. The QuO delegate then sent the event to an actual ProxyPushConsumer, or dropped the event, depending on the system's operating region at that point.

For each test, 1000 events were sent through the Event Channel, and the average event latency and throughput were measured. Because when events are all dropped there is no throughput to the client, the most useful comparison to assess performance degradation is between the case with no QuO involvement (*i.e.*, the normal CORBA method invocation path) and the case with QuO in the loop but with no events being dropped. Therefore, following results were obtained by slightly modifying the contract so that no events would be dropped by the delegate, even though transitions between the regions could occur as described before. The results obtained are summarized in the following table.

Type of run	Average Latency (μs)	Throughput (events/sec)
Consumer (without QuO)	31	10932
Supplier (without QuO)	45	10304
Consumer (with QuO, 2 contract evals)	278	2362
Supplier (with QuO, 2 contract evals)	298	2348
Consumer (with QuO, 1 contract eval)	280	2591
Supplier (with QuO, 1 contract eval)	301	2526

The first and second rows of results were obtained by running the `ECT_Throughput` program without QuO adaptation. The third and fourth rows of results were obtained by running `ECT_Throughput` with the default QuO contract and delegate added. In the default implementation of the QuO delegate, the QuO contract is evaluated two times per method invocation. To obtain the fifth and sixth rows of results, the QuO delegate was manually modified so that the contract was evaluated only once per method invocation.

A subsequent set of tests were run, modifying the QuO contract so that it would drop events after 500 events had gone through the Event Channel. The QuO delegate successfully dropped all events after the first 500 events were sent through the event channel. Measured latency and throughput results for the Supplier side in this case were similar to the results shown above, although no events were delivered to the Client during the event dropping phase.

5 Analysis of Results

Without QuO adaptation added, the `ECT_Throughput` program just measures the latency of sending an event through the ORB in a single process with two CORBA objects. This is on the order of 30-40 μs, with an event throughput of about 10,000 events per second. With in-band QuO adaptation added, the average latency in the supplier and consumer increased by approximately 240 μs, and the event throughput decreased by a factor of approximately 4.5. This would indicate that *in-band* adaptation using a QuO delegate adds significant overhead in comparison to the overhead added by sending an event through the Event Channel and ORB. This is not surprising given the highly optimized nature of the TAO Event Channel, the lack of communication overhead in the collocated example, and the small amount of data in the events. QuO, which was originally targeted toward wide-area, distributed environments, has not been optimized for the local event delivery environment.

The current implementation of the QuO delegate evalates the QuO contract two times: once before invoking the method on the delegated object, and once after invoking the method on the delegated object. The fifth and sixth sets of results were obtained by removing the second contract evaluation which is not needed for the one-way event delivery. The latency of these results was not significantly different (*i.e.*, 2 μs *more*) than the results obtained with two contract evaluations per delegate method invocation. However, event throughput increased by approximately 200 events per second. This indicates that the time to evaluate the contract is on the order of, but smaller than, the time to deliver the event and provides evidence that *out-of-band* QuO adaptation (using a contract that is not in the path of event delivery) is a viable alternative for the collocated event channel example. More experiments are needed to test this.

6 Conclusions

The experiment described in this paper illustrated that the QuO framework could successfully be inserted in the normal CORBA method invocation path of the TAO Real-Time Event Channel, between two CORBA objects collocated in the same process. The QuO delegate successfully could control the passing or dropping of events, based on the type of contract used. The delegate could have also performed other types of adaptation, such as changing the event type or the event priority.

The current implementation of the QuO framework added a significant amount of latency and reduced the throughput of the Event Channel, when used to perform QoS control in-band on the same processor. For some classes of high-performance embedded systems, this may be unacceptable. However, for a more distributed system, where network latency concerns tend to dominate, the increased latency added by QuO may be negligible. Additional experiments need to be conducted to test this. Of particular interest are embedded systems, such as dynamic avionics mission planning platforms [18], in which some invocation paths require in-band adaptation and can tolerate the overhead of a delegate, while other invocation paths cannot tolerate the overhead and may only be adapted via out-of-band techniques [18].

The QuO framework should also be analyzed and instrumented, to identify bottleneck areas that can be eliminated. These bottlenecks may be due to inefficient use of thread mutexes, unnecessary copying of method parameters, etc.

Further Directions: The application of QoS middleware to embedded systems is a dynamic area of research. The QuO group at BBN is actively researching the application of the QuO framework to such systems. Further research directions include integrating QuO with TAO's implementation of Notification [13] and Real-Time Notification [14] services, investigating the use of Portable Interceptors [17] to increase the performance of QuO delegates, and further comparisons in the efficiency of using QuO mechanisms in-band versus out-of-band in the CORBA method invocation path, and their impact on various classes of applications.

References

[1] S. Aslam-Mir, "Smart Transducer Interface Request for Proposal," Object Management Group, November 2000, http://cgi.omg.org/cgi-bin/doc?orbos/2000-11-04

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

[3] J. Loyall, P. Pal, R. Schantz, and F. Webber, "Building Adaptive and Agile Applications Using Intrusion Detection and Response," *Proceedings of NDSS 2000, the Network and Distributed System Security Symposium*, February 2-4 2000, San Diego, CA.

[4] R. Schantz, J. Zinky, D. Karr, D. Bakken, J. Megquier, J. Loyall. "An Object-level Gateway Supporting Integrated-Property Quality of Service," *Proceedings of ISORC '99, The 2nd IEEE International Symposium on Object-oriented Real-time distributed Computing*, May 2-5, 1999, Palais du Grand Large 35 407 Saint-Malo, FRANCE.

[5] P. Pal, J. Loyall, R. Schantz, J. Zinky, F. Webber. "Open Implementation Toolkit for Building Survivable Applications," *Proceedings of DISCEX 2000, the DARPA Information Survivability Conference and Exposition*, January 25-27, 2000, Hilton Head Island, SC.

[6] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.4 ed., October 2000. http://cgi.omg.org/cgi-bin/doc?formal/00-10-1

[7] Sun Microsystems, Inc, *Java Remote Method Interface (RMI) Specification*, October 1998. http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html

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

[9] D. L. Levine, C. D. Gill, D. C. Schmidt. "Dynamic Scheduling Strategies for Avionics Mission Computing," *Proceedings of the 17th IEEE/AIAA Digital Avionics Systems Conference (DASC)*, Nov. 1998

[10] I. Pyarali, T. H. Harrison, and D. C. Schmidt, "Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging," *USENIX Computing Systems*, vol. 9, November/December 1996.

[11] D. C. Schmidt, "A Family of Design Patterns for Application-level Gateways," *The Theory and Practice of Object Systems (Special Issue on Patterns and Pattern Languages)*, vol. 2, no. 1, 1996.

[12] T. H. Harrison, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-time CORBA Event Service," in *Proceedings of OOPSLA '97*, (Atlanta, GA), ACM, October 1997.

[13] Object Management Group. *Notification Service specification*, June 2000. http://cgi.omg.org/cgi-bin/doc?formal/00-06-20

[14] Object Management Group. *Real-time Notification Request for Proposal*, June 2000. http://cgi.omg.org/cgi-bin/doc?orbos/00-06-10

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

[16] C. D. Gill, D. L. Levine, and D. C. Schmidt, "The Design and Performance of a Real-Time CORBA Scheduling Service," *Real-Time Systems*: The International Journal of Time-Critical Computing Systems (special issue on Real-Time Middleware) Vol 20:2

[17] Object Management Group. *Portable Interceptor draft document*, March 2000. http://cgi.omg.org/cgi-bin/doc?ptc/00-03-03

[18] J. Loyall, R. Schantz, J. Zinky, P. Pal, R. Shapiro, C. Rodrigues, M. Atighetchi, D. Karr, J. Gossett, C. Gill. "Comparing and Contrasting Adaptive Middleware Support in Wide-Area and Embedded Distributed Object Applications", submitted to the 21st International Conference on Distributed Computing Systems (ICDCS2001).

[19] J. Loyall, D. Bakken, R. Schantz, J. Zinky, D. Karr, R. Vanegas, and K. Anderson. *"QoS Aspect Languages and Their Runtime Integration"*.Lecture Notes in Computer Science, Vol. 1511, Springer-Verlag. Proceedings of the Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers (LCR98), 28-30 May 1998, Pittsburgh, Pennsylvania.