Applying the CORBA Event Service to Real-time Avionics Systems

Timothy H. Harrison, Carlos O'Ryan, David L. Levine, and Douglas C. Schmidt

{harrison,coryan,levine,schmidt}@cs.wustl.edu
Department of Computer Science, Washington University
St. Louis, MO 63130, USA*

This paper will be submitted to the IEEE Computer special issue on Object-Oriented Real-time Distributed Computing, edited by Philip Sheu and Eltefaat Shokri, June 2000.

Abstract

The CORBA Event Service provides a flexible model for asynchronous and group communication among distributed and collocated objects. However, the standard CORBA Event Service specification lacks important features required by applications with real-time requirements, such as low latency, predictability, event filtering, priority, and event correlation. This paper describes the design and performance of an object-oriented, real-time implementation of the CORBA Event Service that is designed to meet these requirements.

This paper makes four contributions to the design and performance measurement of object-oriented real-time systems. First, it illustrates how the standard CORBA Event Service can be tailored for real-time avionics systems that require low latency, periodic rate-based event processing and efficient event filtering and correlation. Second, it describes how to develop object-oriented event dispatching and scheduling mechanisms that can provide real-time guarantees. Third, it describes how to distribute the Event Service effectively and provide low latency for collocated and remote suppliers/consumers. Finally, the paper presents benchmarks that demonstrate empirically the predictability, low latency, and high utilization of our real-time Event Service for real-time avionics mission computing.

Keywords: Real-time CORBA event systems, object-oriented communication frameworks.

1 Introduction

There is a widespread belief in the embedded systems community that object-oriented (OO) techniques are not suitable for real-time systems. However, many real-time application domains, such as avionics, telecommunications, process control, and distributed interactive simulation, can leverage the benefits of flexible and open distributed object computing architectures, such as those defined in the CORBA specification [1]. If these architectures can be implemented in an efficient and predictable manner, then their benefits make them very compelling for real-time systems.

In this paper, we describe the design and performance of a real-time Event Service, which is a key CORBA-based architecture for push-driven real-time event systems. Our results desmonstrate that the efficiency and predictability of our real-time Event Service is sufficient for applications in the domain of real-time systems such as avionics mission computing [2] and high-speed network monitoring [3].

This paper also empirically evaluates the dynamic binding properties of OO programming languages like C++. Historically, dynamic binding has been viewed as antithetical to real-time systems, which require deterministic execution behavior and low latency. However, on modern platforms, compiler optimizations can reduce the cost of dynamic binding to a negligible amount.

This paper is organized as follows: Section 2 outlines the CORBA reference model, the CORBA Event Service, and related work; Section 2.3 describes how the CORBA Event Service model can help to simplify application development in real-time domains like avionics; Section 3.5 discusses the real-time extensions we added to the CORBA Event Service; Section 5 outlines the OO framework for real-time event dispatching and scheduling that forms the core of TAO's Real-time Event Service; Section 6 shows the results of several benchmarks performed on our implementation, under different workloads using Solaris real-time threads; Section 7 discusses our experiences using OO techniques in a real-time context; and Section 8 presents concluding remarks.

*This work was funded in part by Boeing, NSF grant NCR-9628218, and DARPA contract 9701516.

2 Background

This section outlines the CORBA reference model, the CORBA Event Service, and compares this paper with related work.

2.1 Synopsis of CORBA

CORBA is a distributed object computing middleware standard being defined by the Object Management Group (OMG). CORBA is designed to support the development of flexible and reusable distributed services and applications by (1) separating interfaces from (potentially remote) object implementations and (2) automating many common network programming tasks, such as object registration, location, and activation; request demultiplexing; framing and error-handling; parameter marshalling and demarshalling; and operation dispatching.

Figure 1 illustrates the primary components in the OMG Reference Model architecture [4]. At the heart of the OMG Reference Model is the *Object Request Broker* (ORB). CORBA ORBs allow clients to invoke operations on target object implementations without concern for where the object resides, what language the object is written in, the OS/hardware platform, or the type of communication protocols and networks used to interconnect distributed objects [4].

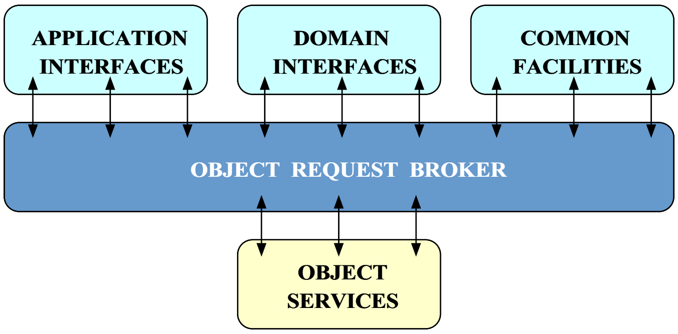

Figure 1: **OMG Reference Model Architecture**

Our prior work on CORBA explored several dimensions of real-time ORB endsystem design including real-time scheduling [5], real-time request demultiplexing [6], real-time I/O subsystem integration [7], and real-time concurrency and connection architectures [8]. This paper extends our previous work [2] on real-time extensions to the *CORBA Event Service* by showing how to distribute the Event Service without incurring extra overhead for collocated supplier/consumer pairs and presenting benchmarks for a federated Event Service configuration.

2.2 Synopsis of the CORBA Event Service

Many distributed applications exchange asynchronous requests using *event-based* execution models [9]. To support these common use-cases, the OMG defined a CORBA Event Service component in the CORBA Object Services (COS) layer in Figure 1. The COS specification [10] presents architectural models and interfaces that factor out common services for developing distributed applications.

The CORBA Event Service defines *supplier* and *consumer* participants. Suppliers generate events and consumers process events received from suppliers. In addition, the CORBA Event Service defines an *Event Channel*, which is a mediator [11] that propagates events to consumers on behalf of suppliers.

The CORBA Event Service model simplifies application software by allowing decoupled suppliers and consumers, asynchronous event delivery, and distributed group communication [12]. In theory, this model seems to address many common needs of event-based, real-time applications. In practice, however, the standard CORBA Event Service specification lacks other important features required by real-time applications such as *real-time event dispatching and scheduling*, *periodic event processing*, and *efficient event filtering and correlation mechanisms*.

To alleviate the limitations with the standard CORBA Event Service, we have developed a *Real-time Event Service* (RT Event Service) as part of the TAO project [5] at Washington University. TAO is a real-time ORB endsystem that provides end-to-end quality of service guarantees to applications by vertically integrating CORBA middleware with OS I/O subsystems, communication protocols, and network interfaces. Figure 2 illustrates the key architectural components in TAO and their relationship to the real-time Event Service.

TAO's RT Event Service augments the CORBA Event Service model by providing source-based and type-based filtering, event correlations, and real-time dispatching. To facilitate real-time scheduling policies, such as rate monotonic (RMS) [13] and maximum urgency first (MUF) [14], TAO's RT Event Channels can be configured to support various strategies for priority-based event dispatching and preemption. This functionality is implemented using TAO's real-time dispatching mechanism that coordinates with its system-wide real-time Scheduling Service [5].

TAO's RT Event Service runs on real-time OS platforms such as VxWorks, LynxOS, and CHORUS/ClassiX, which provide features and performance suited to real-time systems. General-purpose operating systems like Windows NT and Solaris 2.x also provide real-time threads, though they lack certain features required for hard real-time systems [15, 16].

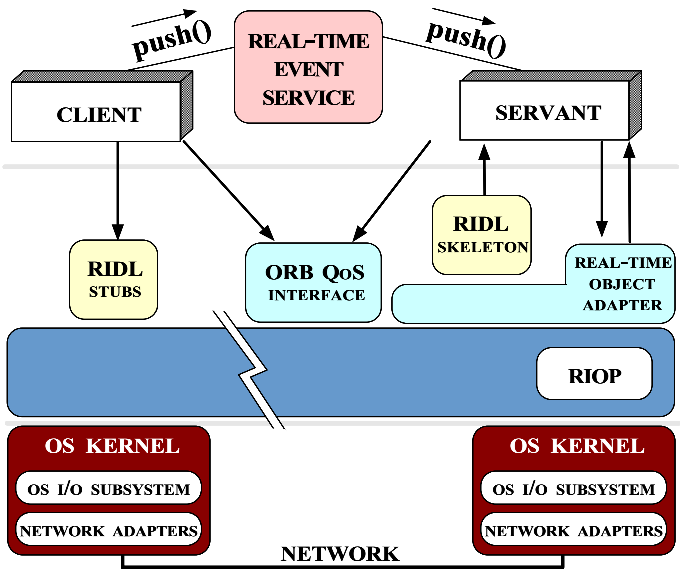

Figure 2: **TAO: An ORB Endsystem Architecture for High-Performance, Real-time CORBA**

2.3 Related Work

Conventional approaches to quality of service (QoS) enforcement have typically adopted existing solutions from the domain of real-time scheduling, [13], fair queuing in network routers [17], or OS support for continuous media applications [18]. In addition, there have been efforts to implement new concurrency mechanisms for real-time processing, such as the real-time threads of Mach [19] and real-time CPU scheduling priorities of Solaris [16].

However, QoS research at the network and OS layers has not necessarily addressed key requirements and usage characteristics of distributed object computing middleware [5]. For instance, research on QoS for network infrastructure has focused largely on policies for allocating bandwidth on a per-connection basis. Likewise, research on real-time operating systems has focused largely on avoiding priority inversions and non-determinism in synchronization and scheduling mechanisms. In contrast, the programming model for developers of OO middleware focuses on invoking remote operations on distributed objects. Determining how to map the results from the network and OS layers to OO middleware is the main focus of our research.

There are several commercial CORBA-compliant Event Service implementations available from multiple vendors, such as Expersoft, IONA, and Borland/Visigenic. IONA also produces OrbixTalk, which is a messaging service based on IP multicast. Since the CORBA Event Service specification does not address issues critical for real-time applications, however, the QoS behavior of these implementations are not acceptable solutions for many application domains.

The OMG has issued a request for proposals (RFP) on a new Notification Service [20] that has generated several responses [21]. The RFP specifies that a proposed Notification Service must be a superset of the COS Event Service with interfaces for the following features: event filtering, event delivery semantics, *e.g.*, at least once, or at most once, security, event channel federations, and event delivery QoS. The organizations contributing to this effort have done excellent work in addressing many of the shortcomings of the CORBA Event Service [22]. However, the OMG RFP documents do not address the implementation issues related to the Notification Service.

Although there has been research on formalisms for real-time objects [23], relatively little published research on the design and performance of real-time OO systems exists. Our approach is based on emerging distributed object computing standards like OMG CORBA ORBs – we focus on the design and performance of various strategies for implementing QoS in real-time ORBs [5].

The QuO project at BBN [24] has defined a model for communicating changes in QoS characteristics between applications, middleware, and the underlying endsystems and network. The QuO architecture differs from our work on RT Event Channels, however, since QuO does not provide hard real-time guarantees of ORB endsystem CPU scheduling. Other research on the CORBA Event Service [25, 26] describe techniques for optimizing event service performance for filtering and message delivery. As with QuO, the focus of this work is not on guaranteeing CPU processing for events with hard real-time deadlines.

Rajkumar, *et al.*, describe a real-time publisher/subscriber prototype developed at CMU SEI [9]. Their Publisher/Subscriber model is functionally similar to the COS Event Service, though it uses real-time threads to prevent priority inversion within the communication framework. An interesting aspect of the CMU model is the separation of priorities for subscription and event transfer so that these activities can be handled by different threads with different priorities. However, the model does not utilize any QoS specifications from publishers (suppliers) or subscribers (consumers). As a result, the message delivery mechanism does not assign thread priorities according to the priorities of publishers or subscribers. In contrast, the TAO Event Service utilizes QoS parameters from suppliers and consumers to guarantee the event delivery semantics determined by a real-time scheduling service.

3 Overview of the OMG CORBA Event Service

3.1 Background

The standard CORBA operation invocation model supports two-way, oneway, and deferred synchronous interactions between clients and servers. The primary strength of the two-way model is its intuitive mapping onto the `object->operation()` paradigm supported by OO languages like C++ and Java. In principle, two-way invocations simplify the development of distributed applications by supporting an implicit request/response protocol that makes remote operation invocations transparent to the client.

In practice, however, the standard CORBA operation invocation models are too restrictive for real-time applications. In particular, these models lack asynchronous message delivery, do not support timed invocations or group communication, and can lead to excessive polling by clients. Moreover, standard oneway invocations are not necessarily reliable and deferred synchronous invocations require the use of the CORBA Dynamic Invocation Interface (DII), which yields excessive overhead for most real-time applications [27].

The Event Service is a CORBA Object Service that is designed to alleviate some of the restrictions with standard CORBA invocation models. In particular, the COS Event Service supports asynchronous message delivery and allows one or more suppliers to send messages to one or more consumers. Event data can be delivered from suppliers to consumers without requiring these participants to know about each other explicitly.

3.2 Structure and Participants for the COS Event Service

Figure 3 shows the key participants in the COS Event Service architecture. The role of each participant is outlined below:

Suppliers and consumers: Consumers are the ultimate targets of events generated by suppliers. Suppliers and consumers can both play active and passive roles. An active push supplier *pushes* an event to a passive push consumer. Likewise, a passive pull supplier waits for an active pull consumer to *pull* an event from it.

Event Channel: At the heart of the COS Event Service is the Event Channel, which plays the role of a mediator between consumers and suppliers. The Event Channel manages object references to suppliers and consumers. It appears as a proxy consumer to the real suppliers on one side of the channel and as a proxy supplier to the real consumers on the other side.

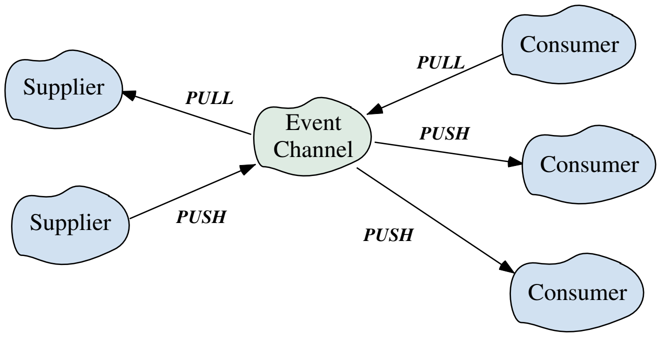

Figure 3: **Participants in the COS Event Channel Architecture**

Suppliers use Event Channels to push data to consumers. Likewise, consumers can explicitly pull data from suppliers. The push and pull semantics of event propagation help to free consumers and suppliers from the overly restrictive synchronous semantics of the standard CORBA two-way communication model. In addition, Event Channels can implement group communication by serving as a replicator, broadcaster, or multicaster that forward events from one or more suppliers to multiple consumers.

The COS Event Service architecture has two models of participant collaborations: *push* and *pull*. This paper focuses on real-time enhancements to the push model, which allows suppliers of events to initiate the transfer of event data to consumers. Suppliers push events to the Event Channel, which in turn pushes the events to consumers.

3.3 Applying TAO's Real-time Event Service to Real-time Avionics Systems

Modern avionics systems are characterized by processing tasks with deterministic and statistical real-time deadlines, periodic processing requirements, and complex data dependencies. Building flexible application software and OO middleware that meets these requirements is challenging because the need for determinism and predictability often results in tightly coupled designs. For instance, conventional avionics mission control applications consist of closely integrated responsibilities that manage sensors, navigate the airplane's course, and control weapon release.

Tight coupling often yields highly efficient custom implementations. As the example below shows, however, the inflexibility of tightly coupled software can substantially increase the effort and cost of integrating new and improved avionics features. For example, navigation suites are a source of continual change, both across platforms and over time. The specific components that make up the navigation suite, *e.g.*, sensors, change frequently to improve accuracy and availability. Many conventional avionics systems treat each implementation as a "point solution," with built-in dependencies on particular components. This tight coupling requires expensive and time consuming development effort to port systems to newer and more powerful navigation technologies.

3.4 Overview of Conventional Avionics Application Architectures

Figure 4 shows a conventional architecture for distributing periodic I/O events throughout an avionics application. This example has the following participants:

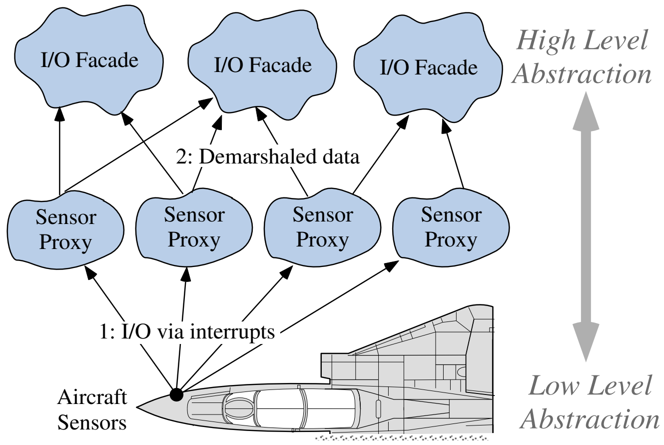

Figure 4: **Example Avionics Mission Control Application**

Aircraft Sensors: Aircraft-specific devices generate sensor data at regular intervals, *e.g.*, 30 Hz (30 times a second), 15 Hz, 5 Hz, etc. The arrival of sensor data generates interrupts that notify mission computing applications to receive the incoming data.

Sensor Proxies: Mission computing systems must process data to and from many types of aircraft sensors, including Global Position System (GPS), Inertial Navigation Set (INS), and Forward Looking Infrared Radar. To decouple the details of sensor communication from the applications, Sensor Proxy objects are created for each sensor on the aircraft. When I/O interrupts occur, data from a sensor is given to an appropriate Sensor Proxy. Each Sensor Proxy object demarshals the incoming data and notifies I/O Facade objects that depend on the sensor's data. Since modern aircraft can be equipped with hundreds of sensors, a large number of Sensor Proxy objects may exist in the system.

I/O Facade: I/O Facades represent objects that depend on data from one or more Sensor Proxies. I/O Facade objects use data from Sensor Proxies to provide higher-level views to other application objects. For instance, the aircraft position computed by an I/O Facade is used by the navigation and weapon release subsystems.

The *push*-driven model described above is commonly used in many real-time environments, such as industrial process control systems and military command/control systems. One positive consequence of this push-driven model is the efficient and predictable execution of operations. For instance, I/O Facades only execute when their event dependencies are satisfied, *i.e.*, when they are called by Sensor Proxies.

In contrast, using a *pull*-driven model to design mission control applications would require I/O Facades that actively acquire data from the Sensor Proxies. If data was not available to be pulled, the calling I/O Facade would need to block awaiting a result. In order for the I/O Facade to pull, the system must allocate additional threads to allow the application to make progress while the I/O Facade task is blocked. However, adding threads to the system has many negative consequences, such as increased context switching overhead, synchronization complexity, and complex real-time thread scheduling policies. Conversely, by using the push model, blocking is largely alleviated, which reduces the need for additional threads. Therefore, this paper focuses on the push model.

3.5 Drawbacks with Conventional Avionics Architectures

A disadvantage to the architecture shown in Figure 4 is the strong coupling between suppliers (Sensor Proxies) and consumers (I/O Facades). For instance, to call back to I/O Facades, each Sensor Proxy must know which I/O Facades depend on its data. As a result, Sensor Proxies must be modified to accommodate changes to the I/O Facade layer, *e.g.*, addition/removal of a consumer. Likewise, consumers that register for callbacks are tightly coupled with suppliers. If the availability of new hardware, such as Forward Looking Infrared Radar, requires a new Sensor Proxy, I/O Facades must be altered to take advantage of the new technology.

3.6 Alleviating Drawbacks with Conventional Avionics Architectures

Figure 5 shows how an Event Channel can alleviate the disadvantages of the tightly coupled consumers and suppliers shown in Figure 4. In Figure 5, Sensor Proxy objects are suppliers of I/O events that are propagated by an Event Channel to I/O Facades, which consume the demarshalled I/O data. Sensor Proxies push I/O events to the channel without having to know which I/O Facades depend on the data. The benefit of using the Event Channel is that Sensor Proxies are unaffected when I/O

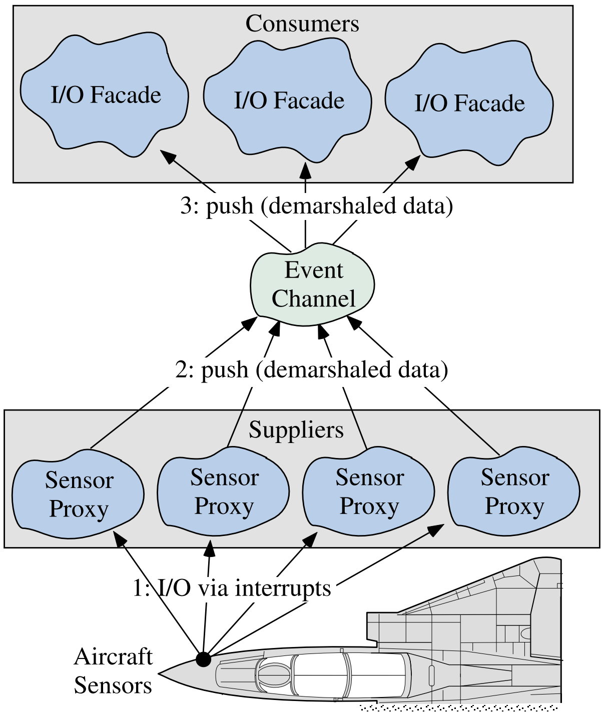

Figure 5: **Example Avionics Application with Event Channel**

Facades are added or removed. This architectural decoupling is described concisely by the Observer pattern [11].

Another benefit of an Event Channel-based architecture is that an I/O Facade need not know which Sensor Proxies supply its data. Since the channel mediates on behalf of the Sensor Proxies, I/O Facades can register for certain types of events, *e.g.*, GPS and/or INS data arrival, without knowing which Sensor Proxies actually supply these types of events (Section 4.2 discusses typed-filtering). Once again, the use of an Event Channel makes it possible to add or remove Sensor Proxies without changing I/O Facades.

4 Overview of TAO's Real-time Event Service

4.1 Overcoming Limitations with the CORBA Event Service

As shown in the previous section, the CORBA COS Event Service provides a flexible model for transmitting asynchronous events among objects. For example, it removes several restrictions inherent in synchronous two-way communication. Moreover, it frees application programmers from the tedious and error-prone details of handling registrations from multiple consumers and suppliers. In addition, the COS Event Service interfaces are fairly intuitive and the consumer/supplier connections and event delivery models are symmetrical and straightforward.

However, the standard COS Event Service Specification lacks several important features required by real-time applications. Chief among these missing features include real-time event dispatching and scheduling, periodic event processing, and centralized event filtering and correlations. To resolve these limitations, we have developed a Real-time Event Service (RT Event Service) as part of the TAO project [5]. TAO's RT Event Service extends the COS Event Service specification to satisfy the quality of service (QoS) needs of real-time applications in domains like avionics, telecommunications, and process control.

The following discussion summarizes the features missing in the COS Event Service and outlines how TAO's Real-time Event Service supports them.

4.1.1 Supporting Guarantees for Real-time Event Dispatching and Scheduling

In a real-time system, events must be processed so that consumers can meet their QoS deadlines. For instance, the Sensor Proxies shown in Figure 5 generate notification events that allow the I/O Facades who depend on the sensor data to execute. To enforce a real-time scheduling policy, higher priority I/O Facades must receive events and be allowed to run to completion before lower priority I/O Facades receive events.

The COS Event Service has no notion of QoS, however. In particular, there is no Event Channel interface that consumers can use to specify their execution and scheduling requirements. Therefore, standard COS Event Channels provide no guarantee that they will dispatch events from suppliers with the correct scheduling priority, relative to the consumers of these events.

TAO's RT Event Service extends the COS Event Service interfaces by allowing consumers and suppliers to specify their execution requirements and characteristics using QoS parameters such as worst-case execution time, rate, etc. These parameters are used by the channel's dispatching mechanism to integrate with the system-wide real-time scheduling policy to determine event dispatch ordering and preemption strategies. Section 5.2.1 describes these QoS parameters in detail.

4.1.2 Supporting Centralized Event Filtering and Correlation

Some consumers can execute whenever an event arrives from any supplier. Other consumers can execute only when an event arrives from a specific supplier. Still other consumers must postpone their execution until multiple events have arrived from a particular set of suppliers, *e.g.*, a correlation of events.

For instance, an I/O Facade may depend on data from a subset of all Sensor Proxies. Furthermore, it may use data from many Sensor Proxies in a single calculation of aircraft position. Therefore, the I/O Facade can not make progress until all of the Sensor Proxy objects receive I/O from their external sensors.

It is possible to implement filtering using standard COS Event Channels, which can be chained to create an event filtering graph that consumers use to receive a subset of the total events in the system. However, the filter graph defined in standard COS increases the number of hops that a message must travel between suppliers and consumers. The increased overhead incurred by traversing these hops is unacceptable for real-time applications with low latency requirements. Furthermore, the COS filtering model does not address the event correlation needs of consumers that must wait for *multiple* events to occur before they can execute.

To alleviate these problems, TAO's RT Event Service provides filtering and correlation mechanisms that allow consumers to specify logical OR and AND event dependencies. When those dependencies are met, the RT Event Service dispatches all events that satisfy the consumers' dependencies. For instance, the I/O Facade can specify its requirements to the RT Event Service so that the channel only notifies the Facade object after all its Sensor Proxies have received I/O. At that time, the I/O Facade receives an IDL `sequence` with all the events from the Sensor Proxies it depends on via a single `push` operation.

4.1.3 Supporting Periodic Processing

Consumers in real-time systems typically require C units of computation time every P milliseconds. For instance, some avionics signal processing filters must be updated periodically or else they will spend a substantial amount of time reconverging. Likewise, an I/O Facade might guarantee regular delivery of its data to higher level components, regardless of whether its Sensor Proxy objects actually generate events at the expected rate.

In both cases, consumers have strict deadlines by which time they must execute the requested C units of computation time. However, the COS Event Service does not permit consumers to specify their temporal execution requirements. Therefore, periodic processing is not supported in standard COS Event Service implementations.

TAO's RT Event Service allows consumers to specify event dependency timeouts. It uses these timeout requests to propagate temporal events in coordination with system scheduling policies. In additional to the canonical use of timeout events, *i.e.*, receiving timeouts at some interval, a consumer can request to receive a timeout event if its dependencies are not satisfied within some time period, *i.e.*, a real-time "watchdog" timer. For instance, an I/O Facade can register to receive a timeout event if its Sensor Proxy dependencies are not satisfied after some time interval. This way, it can make best effort calculations on the older sensor data and notify interested higher level application components.

4.2 TAO's RT Event Service Architecture

Figure 6 shows the high-level architecture of TAO's RT Event Service implementation. The role of each component in the

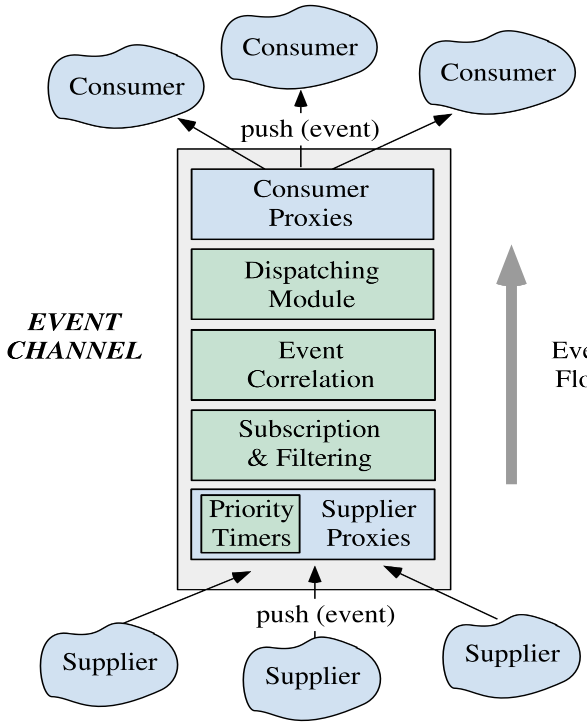

Figure 6: **RT Event Service Architecture**

RT Event Service is outlined below:

4.2.1 Event Channel

In the RT Event Service model, the Event Channel plays the same role as it does in the conventional COS Event Service. Externally, it provides two factory interfaces, `ConsumerAdmin` and `SupplierAdmin`, which allow applications to obtain consumer and supplier administration objects, respectively.

The Event Channel administration objects allow consumers and suppliers to connect and disconnect from the channel. Internally, the channel contains several processing modules based on the ACE Streams framework [28]. As described below, each module encapsulates independent tasks of the channel.

4.2.2 Consumer Proxy Module

The interface to the Consumer Proxy Module is identical to `ConsumerAdmin` interface defined in the COS Event Service `CosEventChannelAdmin` module. It provides factory methods for creating objects that support the `ProxyPushSupplier` interface. In the COS model, the `ProxyPushSupplier` interface is used by consumers to connect and disconnect from the channel.

TAO's RT Event Service model extends the standard COS `ProxyPushSupplier` interfaces so that consumers can register their execution dependencies with a channel. Figure 7 shows the types of data exchanged and the inter-

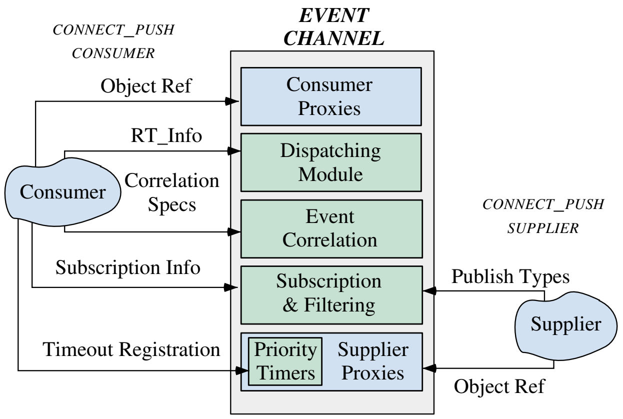

Figure 7: **Collaborations in the RT Event Service Architecture**

object collaborations involved when a consumer invokes the `ProxyPushSupplier::connect_push_consumer` registration operation.

4.2.3 Supplier Proxy Module

The interface to this module is identical to `SupplierAdmin` interface defined in the COS Event Service `CosEventChannelAdmin` module. It provides factory methods for creating objects that support the `ProxyPushConsumer` interface. Suppliers use the `ProxyPushConsumer` interface to connect and disconnect from the channel.

TAO's RT Event Service model extends the standard COS `ProxyPushConsumer` interface so that suppliers can specify the types of events they generate. With this information, the channel's Subscription and Filtering Module can build data structures that allow efficient run-time lookups of subscribed consumers.

`ProxyPushConsumer` objects also represent the entry point of events from suppliers into an Event Channel. When Suppliers transmit an event to the `ProxyPushConsumer` interface via the proxy's `push` operation the channel forwards this event to the `push` operation of interested consumer object(s).

4.2.4 Subscription and Filtering Module

The CORBA Event Service defines Event Channels as broadcasters that forward all events from suppliers to all consumers. This approach has several drawbacks. If consumers are only interested in a subset of events from the suppliers, they must implement their own event filtering to discard unneeded events. Furthermore, if a consumer ultimately discards an event, then delivering the event to the consumer needlessly wastes bandwidth and processing.

To address these shortcomings, TAO's RT Event Service extends the COS interfaces to allow consumers to subscribe for particular subsets of events. The channel uses these subscriptions to filter supplier events, only forwarding them to interested consumers.

There are several reasons why TAO implements filtering in an Event Channel. First, the channel relieves consumers from implementing filtering semantics. Second, it reduces networking load by eliminating filtered events in the channel instead of at consumers. Furthermore, to implement filtering at the suppliers, the suppliers would require knowledge of consumers. Since this would violate one of the primary motivations for an event service (that is, decoupled consumers and suppliers), TAO integrates filtering into the channel.

Adding filtering to an Event Channel requires a well-defined event type system that includes source ID, type, data, and timestamp fields. The complete schema for this type system is beyond the scope of this paper (it is fully described in [29]). The RT Event Channel uses the event type system in the following ways:

Supplier-based filtering: Not all consumers that connect to an Event Channel are interested in the same events. In this case, consumers only register for events generated by certain suppliers. The event type system includes a source ID field that allows applications to specify unique supplier identifiers with each event. The Subscription and Filtering Module uses this field to locate consumers that have subscribed to particular suppliers in *O(1)* worst-case time.

Type-based filtering: Each event contains a type field. This allows consumers to register for events of a particular type. Since the type field is represented as an enumerated type, the Subscription and Filtering Module uses a lookup structure to find type-based subscribers in *O(1)* worst-case time.

Combined supplier/type-based filtering: Consumers can register for any combination of supplier and type-based filtering, *e.g.*, only supplier-based, only type-based, or supplier-based and type-based. To implement this efficiently, the Subscription and Filtering Module maintains type-based subscription tables for every supplier in the system.

When an event enters the Subscription and Filtering Module, consumers that subscribe to combined supplier/type-based IDs are located with two table lookups. The first lookup finds all the type-based subscription tables corresponding to the event's source ID. The second lookup finds the consumers subscribed to the event's type ID.

The Subscription and Filtering Module permits consumers to temporarily disable event delivery by the channel through `suspend` and `resume` operations. These are lightweight operations that have essentially the same effect as de-registering and re-registering for events. Therefore, `suspend` and `resume` are suitable for frequent changes in consumer sets, which commonly occur during mode changes. By incorporating suspension and resumption in the module closest to the suppliers, Event Channel processing is minimized for suspended consumers.

4.2.5 Priority Timers Proxy

The Supplier Proxy Module contains a special-purpose *Priority Timers Proxy* that manages all timers registered with the channel. When a consumer registers for a timeout, the Priority Timers Proxy cooperates with the Run-time Scheduler to ensure that timeouts are dispatched according to the priority of their corresponding consumer.

The Priority Timers Proxy uses a heap-based callout queue [30] provided by ACE. Therefore, in the average and worst case, the time required to schedule, cancel, and expire a timer is *O(log N)* (where *N* is the total number of timers). The timer mechanism preallocates all its memory, which eliminates the need for dynamic memory allocation at run-time. Therefore, this mechanism is well-suited for real-time systems requiring highly predictable and efficient timer operations.

4.2.6 Event Correlation

A consumer may require certain events to occur before it can proceed. To implement this functionality, consumers can specify conjunctive ("AND") or disjunctive ("OR") semantics when registering their filtering requirements, *i.e.*, supplier-based and/or type-based. Conjunctive semantics instruct the channel to notify the consumer when *all* the specified event dependencies are satisfied. Disjunctive semantics instruct the channel to notify the consumer(s) when *any* of the specified event dependencies are satisfied. Consumers can register their filtering requests with a channel multiple times. In this case, the channel creates a disjunction relation for each of its consumer registrations.

Mechanisms that perform filtering and correlation are called *Event Filtering Discriminators* (EFDs). EFDs allow the run-time infrastructure to handle dependency-based notifications that would otherwise be performed by each consumer as *all* events were pushed to it. Thus, EFDs provide a "data reduction" service that minimizes the number of events received by consumers so they only receive events they are interested in.

4.2.7 Dispatching Module

The Dispatching Module determines when events should be delivered to consumers and pushes the events to them accordingly. To guarantee that consumers execute in time to meet their deadlines, this module collaborates with the system-wide Scheduling Service (discussed in Section 5.2).

TAO's Off-line Scheduler initially implements the rate monotonic scheduling policy. Section 5 illustrates how adding new dispatching implementations is straightforward since this module is well-encapsulated from other components in the Event Channel's OO real-time event dispatching framework.

4.3 Federated Event Channels

The original implementation of TAO's RT Event Channel [2] was limited to a single processor configuration. However, modern avionics hardware typically comprises several processor boards connected through a high-speed interconnect, such as a fiber channel network or a VME bus. One way to configure TAO's Event Service is to use a single centralized real-time Event Channel for the entire system. As shown in Figure 8, remote invocations are handled by oneway and two-way CORBA communication mechanisms.

However, the centralized RT Event Channel architecture incurs excessive overhead because the consumer for a given supplier is usually located on the same board. Therefore, if the Event Channel is on a remote board, extra communication overhead and latency are incurred for the common case. Moreover, TAO's collocated method invocation is highly optimized, so it is desirable to exploit this optimization whenever possible.

To address the limitations of the centralized Event Channel architecture, TAO's RT Event Service provides mechanisms to

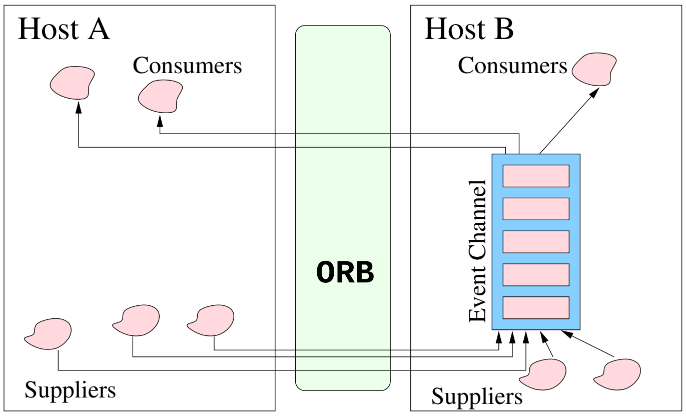

Figure 8: **A Centralized Configuration for the Event Channel.**

connect several Event Channels to form a federation, as shown in Figure 9.

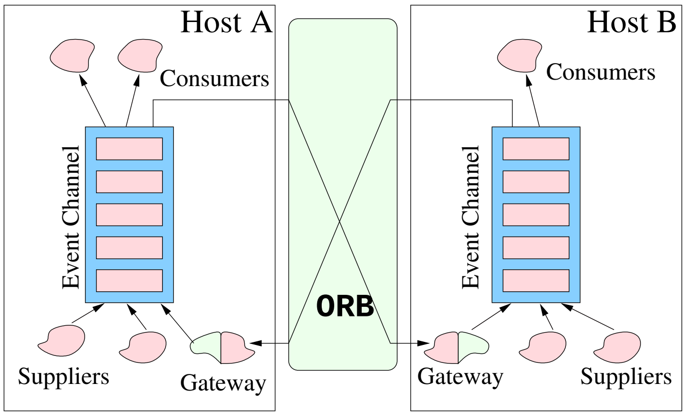

Figure 9: **Federated Event Channels.**

In the federated architecture, multiple Event Channels are connected through a *Gateway*. A Gateway is a servant that connects to one Event Channel as a consumer and forwards all events it receives to other Event Channels. It therefore plays the role of supplier for the second Event Channel. The Gateway usually connects as a supplier to a collocated Event Channel. Thus, TAO's remote communication mechanisms are used only when events are sent from a remote supplier to a local consumer.

A Gateway can be collocated with the Event Channel that supplies events to it. However, the configuration shown in Figure 9 minimizes network traffic since the Gateway only subscribes to events of interest to its consuming Event Channel. The Gateway must subscribe to the disjunction of all the subscriptions in its consuming Event Channel since it is possible that a correlation is satisfied from events arriving from both local and remote consumers.

TAO's current RT Event Service implementation requires that each Event Channel in a federation be connected to every one of its peers. This is not a problem for avionics applications that typically have one Event Channel per processor board. In other domains, however, applications could require hundreds or thousands of Event Channels. In this case, providing a full network would not scale. We are investigating techniques for relaxing this condition.

Other types of distributed applications can benefit from TAO's federated Event Service. For instance, distributed interactive simulations can comprise several nodes in different LANs, where most of the traffic destination is within the same LAN. Configuring an Event Channel on each LAN helps to reduce latency by avoiding round-trip delay to remote nodes.

4.4 Static and Dynamic Event Channel Configuration

The performance requirements of an RT Event Service may vary for different types of real-time applications. The primary motivation for basing the internal architecture of the TAO Event Channel on the ACE Streams framework is to support static and dynamic channel configurations. Each module shown in Figure 7 may contain multiple "pluggable" strategies, each optimized for different requirements. The Streams-based architecture allows independent processing modules to be added, removed, or modified without requiring changes to other modules.

TAO's Event Channel can be configured in the following ways to support different event dispatching, filtering, and dependency semantics:

Full Event Channel: The modules implementing a "full" TAO Event Channel include the Dispatching, Correlation, Filtering, and Consumer/Supplier Proxy modules. A channel that is configured with all these modules supports type and source-based filtering, correlations, and priority-based queueing and dispatching.

Subset Event Channels: As discussed in Section 5, TAO's Event Channel Dispatching Module implements several concurrency strategies. Each strategy caters to the type and availability of system resources, such as the OS threading model and the number of CPUs. TAO's Event Channel framework is designed so that changing the number of threads in the system, or changing to a single-threaded concurrency strategy, does not require modifications to unrelated components in a channel.

The following configurations can be achieved by removing certain modules from an Event Channel:

• **Event Forwarding Discriminator configuration:** Removing the Dispatching Module from the Event Channel yields an Event Forwarding Discriminator (EFD) configuration that supports event filtering and correlations. An EFD configuration is shown in Figure 11(C). Since TAO's Filtering and Correlation Modules have been implemented to guarantee deterministic run-time performance, the EFD configuration is applicable for real-time applications that do not require priority-based queuing and dispatching in the Event Channel. As discussed in Section 5.1.1 below, such systems might implement real-time dispatching in the ORB's Object Adapter level, thereby simplifying the channel.

• **Subscription and Filtering configuration:** Removing the Correlation Module from a full TAO Event Channel yields a Subscription and Filtering configuration. This configuration is useful for applications that have no complex inter-event correlation dependencies, but only want to receive events when they match a simple filter.

• **Broadcaster Repeater configuration:** Removing the Correlation and Dispatching Modules creates a Broadcaster Repeater configuration. This configuration supports neither real-time dispatching nor filtering/correlations. In essence, this implements the semantics of the standard COS Event Channel push model.

In static real-time environments, such as traditional avionics systems, the configuration of an Event Channel is generally performed off-line to reduce startup overhead. In dynamic real-time environments, such as telecommunication call-processing, however, component policies may require alteration at run-time. In these contexts, it may be unacceptable to completely terminate a running Event Channel when a scheduling or concurrency policy is updated. In general, therefore, an RT Event Channel framework must support dynamic reconfiguration of policies without interruption while continuing to service communication operations [31]. Basing TAO's RT Event Channel on the ACE Streams framework supports both static and dynamic (re)configuration.

5 An Object-Oriented Framework for Real-time Event Service Dispatching and Scheduling

Applications and middleware components using TAO's RT Event Service can have deterministic and/or statistical deadlines. As a result, TAO's RT Event Channel utilizes a real-time Scheduling Service [5] to ensure that events are processed before deadlines are missed. Most real-time scheduling policies, such as rate monotonic and earliest deadline first (EDF), require priority-based event dispatching and preemption.

To maximize reuse and allow flexibility between multiple scheduling policies, TAO's Event Channel framework separates its dispatching mechanism from its scheduling policy. The dispatching mechanism implements priority-based dispatching and preemption, but consults a Run-time Scheduler to determine the priorities of objects and events.

This section describes the Dispatching Module and Scheduling Service in TAO's RT Event Channel.

5.1 The Dispatching Module

The Dispatching Module is responsible for implementing priority-based event dispatching and preemption. When the Dispatching Module receives a set of supplier events from the Event Correlation Module, it queries TAO's Run-time Scheduler to determine the priority of the consumers that the events are destined for. With that information, the Dispatching Module can either (1) insert the events in the appropriate priority queues (which are dispatched at a later time) or (2) preempt a running thread to dispatch the new events immediately.

Figure 10 shows the structure and dynamics of the Dispatching Module in the context of the Event Channel. The participants in this figure include the following:

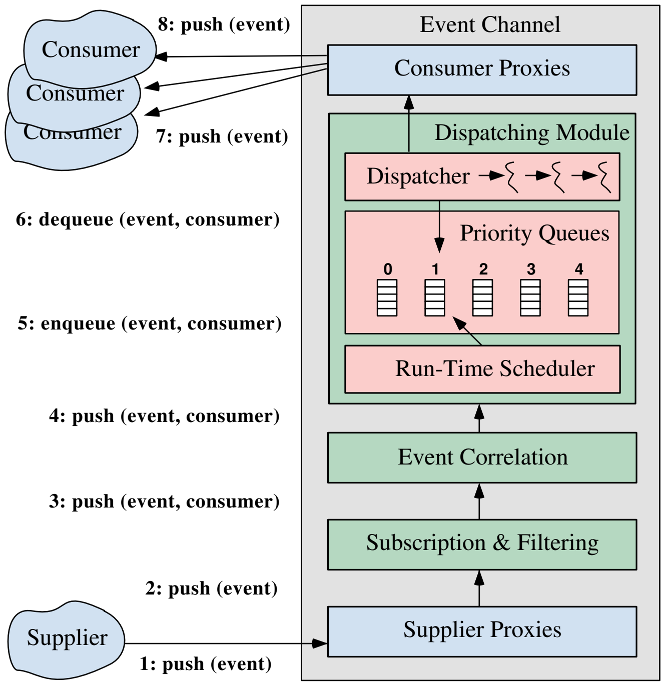

Figure 10: **Event Channel Dispatching**

Consumer and Supplier Proxies: The Event Channel utilizes proxies to encapsulate communication with the consumers and suppliers. For a distributed consumer or supplier, a proxy manages the details of remote communication.

Event filtering and correlation: When events arrive from consumers, the Event Filtering and Correlation Modules determine which consumers should receive the events and when to dispatch the events. These modules forward the events to the Dispatching Module, which handles the details of dispatching each event to its consumer(s) in accordance with the priority of the event/consumer(s) tuple.

Run-time Scheduler: The Dispatching Module collaborates with the Run-time Scheduler to determine priority values of the event/consumer tuples. Given an event and the target consumer, the Run-time Scheduler determines the priority at which the event should be dispatched to the consumer.

The motivation for decoupling the Run-time Scheduler from the Dispatching Module is to allow scheduling policies to evolve independently of the dispatching mechanism. TAO's Run-time Scheduler was initially implemented with a rate monotonic scheduling policy that used the consumer's rate to determine the tuple's priority. Subsequent Run-time Scheduler implementations use dynamic scheduling policies such as EDF and MUF. Thus, by separating the responsibilities of scheduling from dispatching, the Run-time Scheduler can be replaced without affecting unrelated components in the channel.

Priority Queues: Given an event/consumer tuple, the Run-time Scheduler returns a preemption priority and a sub-priority. The Dispatching Module maintains a priority queue of events for each preemption priority used by the Run-time Scheduler. When an event/consumer tuple arrives, it is inserted onto the queue corresponding to the preemption priority returned by the scheduler. The sub-priority is used by the Dispatcher to determine where in the Priority Queue the tuple is placed.

Dispatcher: The Dispatcher is responsible for removing event/consumer tuples from the priority queues and forwarding the events to consumers by calling their `push` operations. Depending on the placement of each tuple in the Priority Queues, the Dispatcher may preempt a running thread to dispatch the new tuple.

For instance, consider the arrival of an event/consumer tuple in a Dispatching Module implemented with real-time preemptive threads. If the Run-time Scheduler assigns the tuple a preemption priority higher than any currently running thread, the Dispatcher will preempt a running thread and dispatch the new tuple. Furthermore, assuming that lower numbers indicate higher priority, the Dispatcher in Figure 10 would dispatch all tuples on queue 0 before dispatching any on queue 1. Similarly, it would dispatch all tuples on queue 1 before those on queue 2, and so on.

To remove tuples from Priority Queues, the Dispatcher always dequeues from the head of the queue. The Run-time Scheduler can determine the order of dequeuing by returning different sub-priorities for different event/consumer tuples. For instance, assume that an implementation of the Run-time Scheduler must ensure that some event E_1 is always dispatched before event E_2, but does not require that the arrival of E_2 preempt a thread dispatching E_1. By assigning a higher sub-priority to event/consumer tuples containing E_1, the tuple will always be queued before any tuples containing E_2. Therefore, the Dispatcher will always dequeue and dispatch E_1 events before E_2 events.

A benefit of separating the functionality of the Dispatcher from the Priority Queues is to allow the implementation of the Dispatcher to change independently of other channel components. TAO's RT Event Channel has been implemented with four different dispatching mechanisms, as described in the Subsection 5.1.1.

5.1.1 Dispatcher Preemption Strategies

An important responsibility of the Event Channel's Dispatcher mechanism is *preemption*. Most real-time scheduling policies require preemption. For example, if consumer A with a priority of 2 is executing when consumer B with a priority of 1 becomes runnable, consumer A should be preempted so that B can run until it completes or is itself preempted by a consumer with a priority of 0. As shown in Figure 11, TAO's

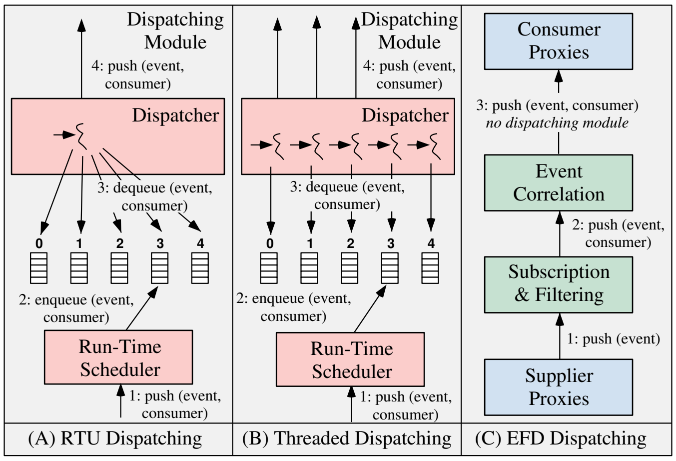

Figure 11: **Dispatcher Implementations**

Event Channel Dispatching Module supports several levels of preemption via the following strategies:

Real-time upcall (RTU) dispatching (with deferred preemption): Figure 11(A) shows a single-threaded implementation where one thread is responsible for dispatching all queued requests. This requires that consumers cooperatively preempt themselves when a higher priority consumer becomes runnable. This model of "deferred preemption" is based on a Real-time Upcall (RTU) concurrency mechanism [32].

The primary benefit of the RTU model is its ability to reduce the context switching, synchronization, and data movement overhead incurred by preemptive multi-threading implementations. However, preemption is deferred to the extent that consumers check to see if they must preempt themselves. This latency may be unacceptable in hard real-time applications, such as avionics mission computing. In contrast, deferred preemption may be acceptable for multimedia applications that have less stringent timing requirements.

Real-time preemptive thread dispatching: Real-time threads are supported by an increasing number of OS platforms, such as VxWorks, LynxOS, CHORUS/ClassiX, Solaris 2.x, and DEC UNIX. Figure 11(B) shows an implementation of the Dispatching Module that allocates a real-time thread (or pool of threads) to each priority queue.

The advantage of this model is that the dispatcher can leverage kernel support for preemption by associating appropriate OS priorities to each thread. For instance, when a thread at the highest priority becomes ready to run, the OS will preempt any lower priority thread that is running and allow the higher priority thread to run. The disadvantage is that this preemption incurs thread context switching overhead. Moreover, applications must identify, and synchronize access to, data that can be shared by multiple threads.

Single-threaded priority-based dispatching: The Dispatching Module can also be implemented with no support for preemption. This is similar to the RTU dispatching mechanism in the sense that a single-thread is used to dispatch events based on priority. However, once a consumer receives an event, it can run to completion regardless of the arrival of events for higher priority consumers.

As with the RTU model, single-threaded dispatching exhibits lower context switching overhead than the real-time thread dispatching model. Moreover, since the channel maintains its own thread of control, it does not borrow supplier threads to propagate events. As a result, the channel is an asynchronous event delivery mechanism for suppliers. However, since the channel's dispatching thread does not implement preemption, consumers run to completion regardless of priority. As a result, single-threaded dispatching can suffer from priority inversion, which results in lower system utilization and non-determinism.

EFD dispatching: As discussed in Section 4.4, the Dispatching Module can be removed from the channel, yielding a purely EFD-based Event Channel. This configuration is shown in Figure 11(C). An EFD channel forwards all events to the consumers without any priority queueing, real-time scheduling, or context switching. Events are dispatched without attention to priority, and there is no preemption of consumers when higher priority event/consumer tuples become available.

EFD channels are appropriate in systems that do not have significant priority-based requirements. In these cases, there is no overhead incurred by a Dispatching Module. However, EFD channels are not always suitable when real-time scheduling policies must be enforced. As shown in Section 5, our performance results show that these drawbacks can cause missed deadlines even under relatively low loads.

5.1.2 Scheduling Enforcement

The real-time scheduling for the version of TAO's Event Channel described in this paper is performed off-line. Therefore, no mechanisms for enforcing component behavior are provided. Consequently, tasks that overrun their allotted resource allocations can cause other tasks to miss their deadlines.

An advantage of a "trust"-based scheduling enforcement policy is there is no overhead incurred by QoS enforcement mechanisms that would otherwise be necessary to monitor and enforce the scheduling behavior at run-time. A disadvantage is that all components must behave properly, *i.e.*, they must use only the resources allotted to them. Though the architecture of our Event Service framework supports QoS enforcement, the decision not to include this mechanism in the Event Channel is motivated by the static scheduling characteristics and stringent performance requirements of real-time avionics applications.

5.1.3 Visualization of Dispatching Module Implementations

To visualize the semantic differences between the four Dispatching Module implementations outlined in Section 5.1.1, we implemented a timeline visualization tool in Java. The timeline tool reads event logs from RT Event Service test runs and displays a timeline of supplier and consumer activity. Figures 12 and 13 show timelines from multi-threaded and single-threaded implementations of the Dispatching Module, respectively. Each test run consists of 3 suppliers and 3 consumers, which are listed on the y-axis. Supplier$_2$ and consumer$_2$ run at the highest frequency (40 Hz), supplier$_1$ and consumer$_1$ run at the next highest frequency (20 Hz), and supplier$_0$ and consumer$_0$ run at the lowest frequency (10 Hz).

The x-axis denotes time in μseconds. Each consumer and supplier outputs a point when it receives an event from the Event Channel. Another point is output when it finishes processing the event. Suppliers receive timeouts and generate a single event for each timeout. Each consumer registers for events from a single supplier. A horizontal line indicates the time span when the respective consumer or supplier runs on the CPU.

Each figure is explained below:

Real-time multi-threaded dispatching: Figure 12 shows how OS real-time thread support for preemption results in supplier_0 and consumer_0 being preempted whenever higher priority tasks become runnable. Our performance results (discussed in Section 6) demonstrate how Dispatching Module implementations like real-time thread dispatching, which support more responsive preemption mechanisms, yield higher resource utilization without missing deadlines.

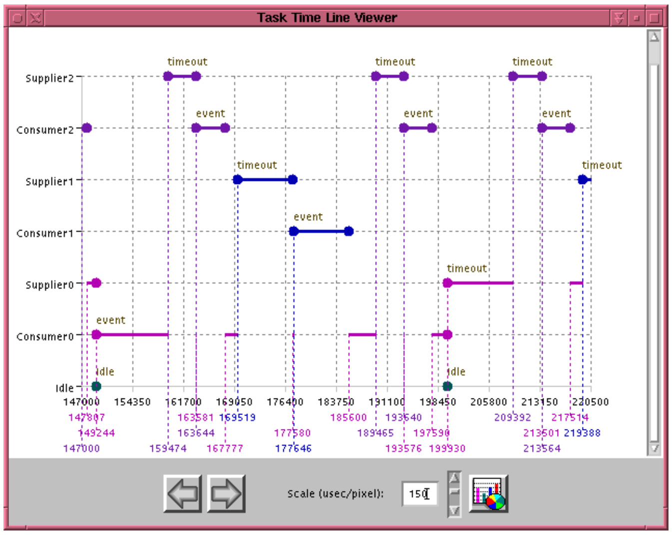

Figure 12: **Timeline from Real-time Multi-Threaded Channel**

Single-threaded dispatching: Figure 13 shows how a single-threaded dispatching module can result in deadlines being missed if lower priority tasks hold the CPU for excessive periods of time. The negative values next to the end times of supplier_2 and consumer_2 show the number of μsecs the deadlines were missed. In other words, consumer_0 held the CPU too long, so that higher rate suppliers and consumers were unable to execute in time to preserve correct application behavior.

5.2 Real-time Scheduling Service

The RT Event Service must guarantee that consumers receive and process events with sufficient time to meet their deadlines. To accomplish this, we have developed a Real-time Scheduling Service. The two primary components in the Real-time Scheduling Service are the Run-time Scheduler and Off-line

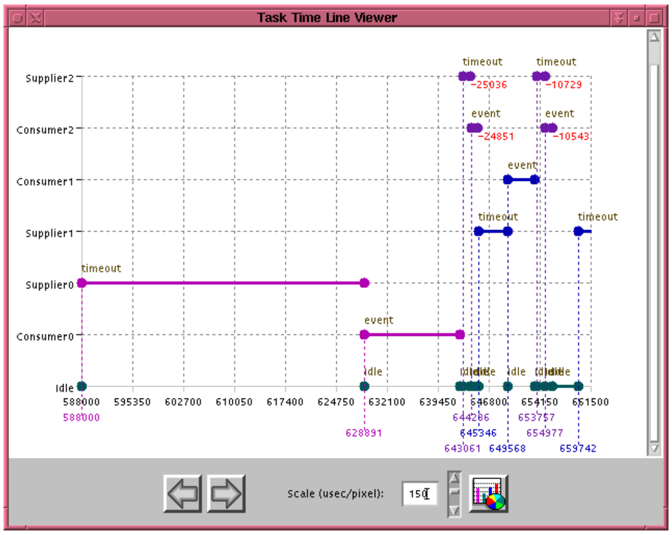

Figure 13: **Timeline from Single-Threaded Channel**

Scheduler, which are summarized below. A complete discussion of these components is beyond the scope of this paper ([29] describes these components in detail).

5.2.1 Run-time Scheduler

The Run-time Scheduler associates priorities with target object implementation operations at run-time. The implementation of the Real-time Scheduling Service described in this paper uses a static scheduling policy. Therefore, thread priorities are determined prior to run-time by the Off-line Scheduler.

Our Real-time Scheduling Service requires that if an object is to be scheduled, each of its operations must export an `RT_Info` data structure describing the operation's execution properties. During scheduling *configuration runs* (described in Section 5.2.2 below), `RT_Infos` contain execution times and rate requirements. At run-time, the Static Scheduler need not know any information about an operation's execution characteristics. Only the operation's priority is needed, so the scheduler can determine how the operation should be dispatched. Thus, at run-time, each operation's `Rt_Info` need only contain priority values for the operation.

At run-time, the Dispatching Module queries the Run-time Scheduler for the priority of a consumer's push operation. The Run-time Scheduler uses a static repository that identifies the execution requirements (including priority) of each operation. The Event Channel's Dispatching Module uses the operation priority returned by the Run-time Scheduler to determine which priority queue an event/consumer tuple should be inserted into.

All scheduling and priority computation is performed off-line. This allows priorities to be computed at run-time efficiently, *i.e.*, located in *O(1)* time. Thus, TAO's Run-time Scheduler simply provides an interface to the results of the Off-line Scheduler, discussed below.

5.2.2 Off-line Scheduler

The Off-line Scheduler has two responsibilities. First, it assigns priorities to object operations. Second, it determines whether a current Event Channel configuration is schedulable given the available resources and the execution requirements of supplier and consumer operations. Both responsibilities require that operation interdependencies be calculated by a *task interdependency compilation* process during during a *configuration run*. Task interdependency compilation builds a repository that records which objects' operations call each other. This can be visualized as a directed graph where the nodes in the graph are object operations and directed edges indicate that one operation calls another, as shown in Figure 14.

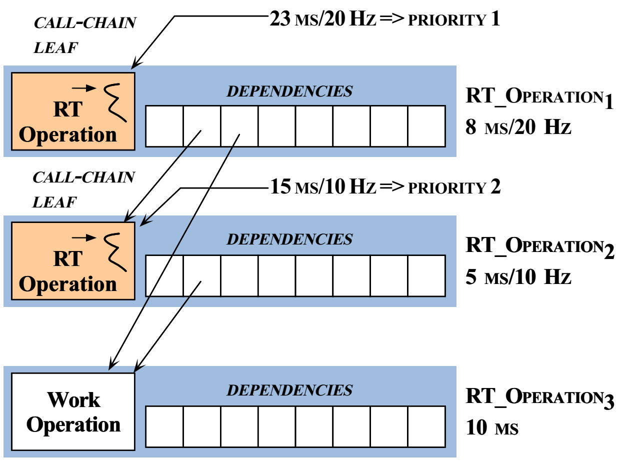

Figure 14: **Scheduling Service Internal Repository**

Once task interdependency compilation is complete, the Off-line Scheduler assigns priorities to each object operation. The implementation of the Event Service described in this paper utilizes a rate monotonic scheduling policy [13, 33]. Therefore, priorities are assigned based on task rates, *i.e.*, higher priorities are assigned to threads with faster rates. For instance, a task that needs to execute at 30 Hz would be assigned to a thread with a higher priority than a task that needs to execute at 15 Hz.

Most operating systems that support real-time threads guarantee higher priority threads will (1) preempt lower priority threads and (2) run to completion (or until higher priority threads preempt them). Therefore, object operations with higher priorities will preempt object operations with lower priorities. These priority values are computed by the Off-line Scheduler and are stored in a table that is queried by the Run-time Scheduler at execution time.

6 Performance Tests

In our previous work [2] we benchmarked the utilization of a single processor prototype of TAO's real-time Event Service. In the following sections, we extend these benchmarks to to illustrate how performance is not affected by a complete ORB and RT Event Channel implementation. We also quantify the utilization of a federated Event Channel architecture.

6.1 Utilization Measurements

For non-real-time Event Channels, *e.g.*, EFD-based and Single-Threaded, correctness implies that consumers receive events when their dependencies are met, *i.e.*, source/type subscriptions and correlations. In contrast, for real-time Event Channels, *e.g.*, RTUs and real-time threads, correctness implies that all deadlines are met. Therefore, correct RT Event Service behavior requires that (1) consumers receive events when their dependencies are satisfied *and* (2) consumers receive these events in time to meet their deadlines.

An important metric for evaluating the performance of the RT Event Service is the *schedulable bound*. The schedulable bound of a real-time schedule is the maximum resource utilization possible without deadlines being missed [32]. Likewise, the schedulable bound of the RT Event Service is the maximum CPU utilization that suppliers and consumers can achieve without missing deadlines.

For TAO's Real-time Scheduling Service to guarantee the schedulability of a system, *i.e.*, that all tasks meet their deadlines, high-priority tasks must preempt lower priority tasks. With rate monotonic scheduling, higher rate tasks preempt lower rate tasks. The performance tests discussed below were conducted on a single-CPU 300 Mhz Sun UltraSPARC 30 workstation with 256 MB RAM running Solaris 2.5.1.

We measured the added overhead of the distribution strategy for TAO's real-time Event Services over a single collocated EC. Two separate experiments were conducted: one measured the utilization of a single Event Channel configuration and another measured the utilization of a federated Event Channel configuration.

6.1.1 Single Event Channel Utilization

The first experiment used a single Event Channel that was collocated with a high-priority Supplier/Consumer pair and a low-priority Supplier/Consumer pair. The processing time for high-priority events was increased until the low-priority task could not meet its deadline. In Figure 15 we plot the aver-

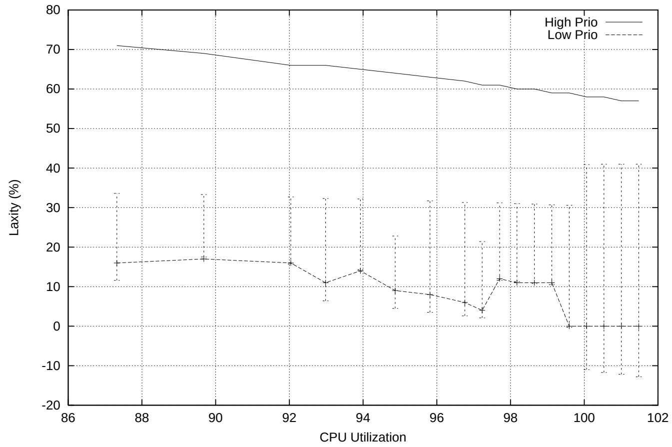

Figure 15: **Utilization for a Single Event Channel**

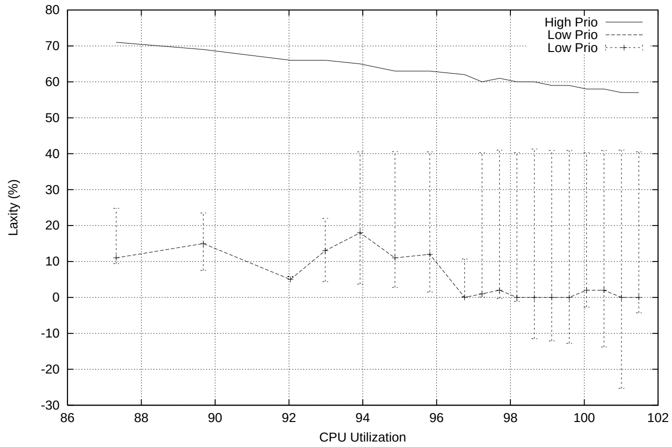

Figure 16: **Utilization for a Federated Event Channel**

age laxity for the high-priority events and for the low-priority events. The error bars represent the minimum and maximum laxity for each experiment. Negative laxity means that a deadline is missed.

Note that the Event Channel achieved over 99% utilization before its low-priority tasks began to miss deadlines. In contrast, the high-priority task laxity also decreases. It is interesting to observe that this decrease in almost linear with the utilization, even when the low-priority task cannot longer meet its deadlines.

6.1.2 Federated Event Channel Utilization

In the second experiment, two Event Channels were configured in separate OS processes on the same host. No work was performed when processing the remote events, but the processing time for high-priority events was increased until deadlines were missed. Figure 16 also depicts the laxity for both high-priority and low-priority tasks, though performance is affected the Event Channel can still maintain high utilization (over 96%) before any deadlines were missed. As shown in Section 6.2, this small ($\sim$3%) loss of utilization yields a performance improvement for the common case, *i.e.*, where events are exchanged with collocated suppliers and consumers.

6.2 Latency Measurements

Another important metric of Event Channel performance is the latency it introduces between suppliers and consumers. To determine Event Channel latency, we developed an Event Latency Test. To obtain the end-to-end supplier $\rightarrow$ consumer latency, this test timestamps each event as it originates in the supplier and subtracts that time from the arrival time at the consumer. The consumer does not do anything with the event other than store the measurement in a preallocated array.

6.2.1 End-to-end Latency Results

The latency tests were run on a Sun UltraSPARC 30 with a single 300 Mhz CPU, running SunOS 5.5.1. The Event Channel and test applications were built with Sun C++ 4.2 with `-fast` optimization. Two tests were performed. In the first test the consumers, suppliers, and Event Channel were collocated in the same process to eliminate ORB remote communication overhead. In the second test two identical processes were created and the consumers in each process subscribed to both local and remote events. In both tests, there was no other significant activity on the workstation during the benchmarking. All tests were run in the Solaris real-time scheduling class so they had the highest software priority (but below hardware interrupts) [16].

In the single process case, we determined the best-case supplier-to-consumer latency was $\sim$60 μsecs. In each case, as the number of suppliers and/or consumers increased, the latency increased as well, as shown in Table 1 and Table 2.

Note how for two processes, the local events exhibit similar latency to the local events in the single process case. In particular, no extra overhead is incurred due to possible remote consumers. In contrast, the remote consumer performance is several times higher than the local events, though it is still close to the performance of a typical oneway RPC request [34].

6.2.2 Minimal Event Spacing Results

Another important performance metric is the *minimum event spacing*, *i.e.*, the maximum rate at which the Event Channel can deliver messages before its overhead introduces extra latency. This test was executed in a single process, where suppliers generate a fixed number (500) of events, and the consumers do no work other than maintain simple statistics. We progressively decreased the event generation period and measured the ratio between the effective event rate and the expected event rate. As seen in figure 17 the Event Channel is able to deliver

		Latency, μsec					
		First Consumer			Last Consumer		
Sup.	Con.	Min	Max	Avg	Min	Max	Avg
1	1	53	93	58	53	93	58
1	5	107	189	114	197	284	206
1	10	171	230	183	379	451	393
1	20	291	340	300	741	817	760
2	1	49	67	51	49	67	51
2	5	95	124	100	180	213	187
2	10	159	281	170	360	498	374
2	20	283	333	299	758	828	781
10	1	51	303	72	51	303	72
10	5	100	211	113	187	1,210	284
10	10	167	222	176	369	2,545	576
10	20	211	310	290	741	4,895	1,137

Table 1: **Event Latency, μsecs, Through Event Channel**

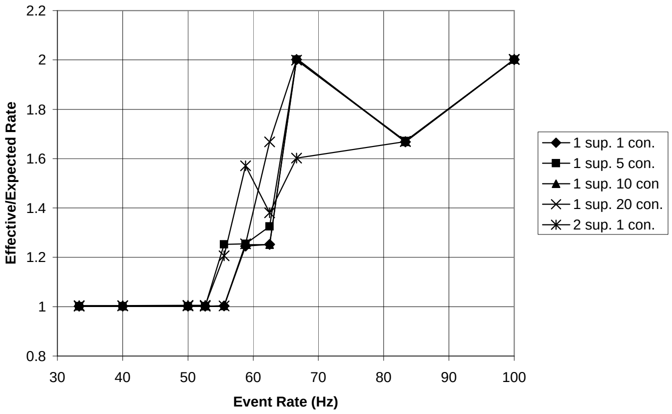

Figure 17: **Minimum Event Spacing**

over 50 messages per second before it experiences any noticeable overhead. We believe this result can be improved and we are currently experimenting with how to reduce the overhead.

		Latency, μsec					
		Local Event			Remote Event		
Sup.	Con.	Min	Max	Avg	Min	Max	Avg
1	1	57	71	63	772	1,078	820
1	5	108	207	155	765	1,995	1,283
1	10	170	598	289	795	5,853	3,544
1	20	303	819	534	756	6,047	3,084
2	1	50	95	57	1,226	2,329	1,297
2	5	101	214	152	1,274	4,577	2,330
2	10	167	421	274	1,226	6,676	3,448
2	20	280	821	519	1,225	20,727	6,038
10	1	49	218	60	1,406	3,969	3,102
10	5	100	1,170	172	1,477	12,773	6,282
10	10	158	2,379	310	1,258	19,153	7,904
10	20	209	4,900	596	1,266	65,449	24,345

Table 2: **Event Latency, μsecs, Local and Remote Events**

7 Evaluating the Use of OO for Real-time Systems

While applying OO technologies to real-time systems we encountered two issues regarding polymorphism that threatened to compromise the predictability and performance of our real-time systems. This section briefly discusses each of the issues and how our systems address the potential problems.

7.1 The Cost of Dynamic Binding Mechanisms

Since our systems are developed using C++, dynamic binding is implemented via virtual method tables. As a result, compilers can implement highly optimized virtual method call mechanisms that impose constant-time overhead. These algorithms typically involve loading the *this* pointer, adjustment of the *this* pointer (for multiple inheritance), lookup of the method offset in the virtual method table, and final calculation of the address before invoking the method. However, these steps still have bounded completion times allowing predictable virtual method call performance regardless of the degree of inheritance used by applications.

We measured the cost of virtual method calls on these platforms: VxWorks 5.3.1 on a 60 MHz Pentium with Cygnus g++

Platform	Call time, μsec Global Function	Non-Virtual Method	Virtual Method	ratio Virtual to Global Function	Virtual to Non-Virtual
VxWorks/g++/60 MHz Pentium	0.300	0.450	0.900	3.0	2.0
VxWorks/GHS/200 MHz Pentium	0.174	0.358	0.542	3.1	1.5
VxWorks/GHS/200 MHz PowerPC	0.021	0.021	0.068	3.2	3.2
Solaris/g++/168 MHz Ultrasparc	0.069	0.061	0.173	2.5	2.8
IRIX/CC/180 MHz SGI Origin200	0.061	0.061	0.084	1.4	1.4
NT/MSVC++/200 MHz Pentium	0.030	0.035	0.035	1.2	1.0

Table 3: **Cost of Virtual Method Calls**

2.7.2-960126, VxWorks 5.3.1 on a 200 MHz Pentium with GreenHills 1.8.8, VxWorks 5.3.1 on a 200 MHz PowerPC with GreenHills 1.8.8D, Solaris 2.5.1 on a dual-CPU 168 MHz Sun UltraSPARC 2 with g++ 2.7.2, Irix 6.4 on a dual-CPU 180 MHz SGI Origin200 with SGI C++ 7.10, and Windows NT 4.0 on a 200 MHz PentiumPro with Microsoft Visual C++ 5.0. As shown in Table 3, a virtual method call costs roughly 2 to 5 times that of a global function or non-virtual method call.

While these ratios seem high, for some platforms, the absolute time penalty (relative to a global function call) for a virtual method call was less than 0.6 μsec on the tested platforms. Our experience has been that this is not an impediment to real-time system performance, though we avoid virtual methods where not needed. Furthermore, modern compilers implement strategies for replacing indirect virtual method calls with direct non-virtual calls [35]. The results for the IRIX C++ and Microsoft VC++ compilers indicate well-optimized virtual method calls.

7.2 The Cost of Polymorphism

Polymorphism facilitates run-time changes in object behavior. Real-time systems often require predictable behavior of all components. Initially, the flexibility of polymorphism seems to be at odds with the requirement for real-time predictability. We resolved this issue using the Off-line Scheduler discussed in Section 5.2. Since scheduling is performed off-line, all objects and operations must be known in advance. Therefore, it is the responsibility of the Off-line Scheduler to determine whether a particular system configuration will meet all of its deadlines. As a result, when a virtual method is called at run-time, the system is not concerned with the actual implementation being invoked. The Off-line Scheduler has already guaranteed that its deadline will be met, based on the published parameters of each schedulable operation.

One advantage of our approach is that operation invocations only pay the overhead of the C++ virtual method call. If the schedule was not determined off-line, a run-time (dynamic) scheduler would need to intercede before any abstract operation was invoked, which incurs additional overhead. For instance, if a rate monotonic scheduling policy is used, the scheduler must determine the rate that each object operation executes in order to calculate its priority. Furthermore, this type of dynamic scheduler must make some type of guarantee, either weak or strong, that deadlines will be met.

One way a scheduler could make strong guarantees is to perform admission control, which permits operations to execute when the necessary resources are available. Admission control requires that object operations export execution properties, such as worst-case execution time. Alternatively, the scheduler might implement a weaker, "best-effort" admission policy. For example, if an Earliest Deadline First policy is used, object operations with the nearest deadlines are given priority over operations with later deadlines. Such a policy would require that object operation deadlines be exported or calculated by the scheduler. This type of support for dynamic scheduling can incur significant overhead, and thus decrease effective resource utilization. As a result, dynamic scheduling solutions are sometimes not viable solutions for systems with hard deadlines and constrained resources.

Since all objects and operations in TAO's Real-time Event Service are determined off-line, one could argue that no real polymorphism exists. Although this is true to a certain extent, there are more benefits to dynamic binding than just changing behavior at run-time. In particular, we found that the ability to develop components independently of applications that use them significantly increases the potential for reuse in the avionics domain. For instance, since the Event Channel pushes to abstract `PushConsumer` interfaces, the code for the Event Channel remains decoupled from the number and type of application `PushConsumer` objects.

8 Concluding Remarks

Many real-time applications require support for asynchronous, event-based communication. The CORBA COS Event Service provides a flexible OO model where Event Channels dispatch events to consumers on behalf of suppliers. TAO'S Real-time Event Service described in this paper augments this model with Event Channels that support source and type-based filtering, event correlations, and real-time event dispatching.

By configuring different Run-time Scheduler strategies, TAO's Event Channels can be configured with multiple scheduling policies, such as rate monotonic scheduling and maximum urgency first. Similarly, channels can be built with varying levels of support for preemption by configuring different Dispatcher preemption strategies, such as EFD, single-threaded, RTU, and real-time thread Dispatchers. This flexibility allows applications to adapt their scheduling and dispatching policies to obtain optimal utilization for different application requirements and platform resource characteristics.

Our performance results demonstrate that using a single Event Channel for distribution yields poor performance and high latency. We present an architecture to build federations of Event Channels that yields near optimal performance for collocated supplier/consumer pairs and does not affect remote event performance significantly. In addition, our measurements indicate that the dynamic binding mechanisms used by our C++ compilers are not fundamentally at odds with the deterministic execution behavior required by real-time applications. In general, our results illustrate that it is feasible to apply CORBA Object Services to develop real-time systems.

The implementation of TAO's Real-time Event Service described in this paper is written in C++ as one of the services in TAO [5]. TAO is a real-time implementation of CORBA based on the ACE framework [28]. ACE is a widely used communication framework that contains a rich set of high-performance and real-time reusable software components. These components automate common communication tasks such as connection establishment, event demultiplexing and event handler dispatching, message routing, dynamic configuration of services, and flexible concurrency control for network services. ACE has been ported to most real-time OS platforms including VxWorks, LynxOS, CHORUS/ClassiX, pSoS, Solaris, Win32, and most POSIX 1003.1c implementations.

TAO's RT Event Service is currently deployed at Boeing in St. Louis, MO, where it is being used to develop operation flight programs for next-generation avionics systems. The source code and documentation for TAO and its RT Event Service implementation is freely available at `www.cs.wustl.edu/~schmidt/TAO.html`.

9 Acknowledgments

This work was funded in part by Boeing. We gratefully acknowledge the support and direction of the Boeing Principal Investigator, Bryan Doerr. In addition, we would like to thank Seth Widoff for building the Java visualization tool that generated the time lines shown in Sections 5 and 6.

References

[1] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.2 ed., Feb. 1998.

[2] 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.

[3] G. Parulkar, D. C. Schmidt, E. Kraemer, J. Turner, and A. Kantawala, "An Architecture for Monitoring, Visualization, and Control and Gigabit Networks," *IEEE Network*, vol. 11, September/October 1997.

[4] S. Vinoski, "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments," *IEEE Communications Magazine*, vol. 14, February 1997.

[5] 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.

[6] A. Gokhale and D. C. Schmidt, "Evaluating the Performance of Demultiplexing Strategies for Real-time CORBA," in *Proceedings of GLOBECOM '97*, (Phoenix, AZ), IEEE, November 1997.

[7] F. Kuhns, D. C. Schmidt, and D. L. Levine, "The Design and Performance of a Real-time I/O Subsystem," in *Proceedings of the* 5^{th} *IEEE Real-Time Technology and Applications Symposium*, (Vancouver, British Columbia, Canada), pp. 154–163, IEEE, June 1999.

[8] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, "Alleviating Priority Inversion and Non-determinism in Real-time CORBA ORB Core Architectures," in *Proceedings of the* 4^{th} *IEEE Real-Time Technology and Applications Symposium*, (Denver, CO), IEEE, June 1998.

[9] R. Rajkumar, M. Gagliardi, and L. Sha, "The Real-Time Publisher/Subscriber Inter-Process Communication Model for Distributed Real-Time Systems: Design and Implementation," in *First IEEE Real-Time Technology and Applications Symposium*, May 1995.

[10] Object Management Group, *CORBAServices: Common Object Services Specification, Revised Edition*, 95-3-31 ed., Mar. 1995.

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

[12] S. Maffeis, "Adding Group Communication and Fault-Tolerance to CORBA," in *Proceedings of the Conference on Object-Oriented Technologies*, (Monterey, CA), USENIX, June 1995.

[13] C. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *JACM*, vol. 20, pp. 46–61, January 1973.

[14] D. B. Stewart, D. E. Schmitz, and P. K. Khosla, "Implementing Real-Time Robotic Systems using CHIMERA II," in *Proceedings of 1990 IEEE International Conference on Robotics and Automation*, (Cincinnatti, OH), 1992.

[15] M. Timmerman and J.-C. Monfret, "Windows NT as Real-Time OS?," *Real-Time Magazine*, 2Q 1997. http://www.realtime-info.be/encyc/magazine/97q2/winntasrtos.htm.

[16] S. Khanna and et. al., "Realtime Scheduling in SunOS 5.0," in *Proceedings of the USENIX Winter Conference*, pp. 375–390, USENIX Association, 1992.

[17] L. Zhang, "Virtual Clock: A New Traffic Control Algorithm for Packet Switched Networks," in *Proceedings of the Symposium on Communications Architectures and Protocols (SIGCOMM)*, (Philadelphia, PA), pp. 19–29, ACM, Sept. 1990.

[18] G. Coulson, G. Blair, J.-B. Stefani, F. Horn, and L. Hazard, "Supporting the Real-time Requirements of Continuous Media in Open Distributed Processing," *Computer Networks and ISDN Systems*, pp. 1231–1246, 1995.

[19] H. Tokuda, T. Nakajima, and P. Rao, "Real-Time Mach: Towards Predictable Real-time Systems," in *USENIX Mach Workshop*, USENIX, October 1990.

[20] Object Management Group, *Notification Service Request For Proposal*, OMG Document telecom/97-01-03 ed., January 1997.

[21] Object Management Group Telecommunications Domain Task Force, "Notification Service RFP (Telecom RFP3)," 1997.

[22] D. C. Schmidt and S. Vinoski, "Object Interconnections: Overcoming Drawbacks in the OMG Events Service," *C++ Report*, vol. 9, July-August 1997.

[23] I. Satoh and M. Tokoro, "Time and Asynchrony in Interactions among Distributed Real-Time Objects," in *Proceedings of 9th European Conference on Object-Oriented Programming*, Aug. 1995.

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

[25] Y. Aahlad, B. Martin, M. Marathe, and C. Lee, "Asynchronous Notification Among Distributed Objects," in *Proceedings of the* 2^{nd} *Conference on Object-Oriented Technologies and Systems*, (Toronto, Canada), USENIX, June 1996.

[26] C. Ma and J. Bacon, "COBEA: A CORBA-Based Event Architecture," in *Proceedings of the* 4^{rd} *Conference on Object-Oriented Technologies and Systems*, USENIX, Apr. 1998.

[27] A. Gokhale and D. C. Schmidt, "The Performance of the CORBA Dynamic Invocation Interface and Dynamic Skeleton Interface over High-Speed ATM Networks," in *Proceedings of GLOBECOM '96*, (London, England), pp. 50–56, IEEE, November 1996.

[28] D. C. Schmidt, "ACE: an Object-Oriented Framework for Developing Distributed Applications," in *Proceedings of the* 6^{th} *USENIX C++ Technical Conference*, (Cambridge, Massachusetts), USENIX Association, April 1994.

[29] D. C. Schmidt, D. L. Levine, and T. H. Harrison, "An ORB Endsystem Architecture for Hard Real-Time Scheduling," Feb. 1997. Submitted to OMG in response to RFI ORBOS/96-09-02.

[30] R. E. Barkley and T. P. Lee, "A Heap-Based Callout Implementation to Meet Real-Time Needs," in *Proceedings of the USENIX Summer Conference*, pp. 213–222, USENIX Association, June 1988.

[31] J.-B. Stefani, "Requirements for a real-time ORB," tech. rep., ReTINA, 1996.

[32] R. Gopalakrishnan and G. Parulkar, "Bringing Real-time Scheduling Theory and Practice Closer for Multimedia Computing," in *SIGMETRICS Conference*, (Philadelphia, PA), ACM, May 1996.

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

[34] A. Gokhale and D. C. Schmidt, "Measuring the Performance of Communication Middleware on High-Speed Networks," in *Proceedings of SIGCOMM '96*, (Stanford, CA), pp. 306–317, ACM, August 1996.

[35] S. Porat, D. Bernstein, Y. Fedorov, J. Rodrigue, and E. Yahav, "Compiler Optimization of C++ Virtual Function Calls," in *Proceedings of the* 2^{nd} *Conference on Object-Oriented Technologies and Systems*, (Toronto, Canada), USENIX, June 1996.