Patterns and Performance of Distributed Real-time and Embedded Publisher/Subscriber Architectures

Douglas C. Schmidt and Carlos O'Ryan
{schmidt,coryan}@uci.edu
Department of Electrical & Computer Engineering
University of California, Irvine, CA 92697 *

A subset of this paper appeared in the Journal of Systems and Software, Special Issue on Software Architecture – Engineering Quality Attributes, edited by Lars Lundberg and Jan Bosch, Elsevier, 2002.

Abstract

This paper makes four contributions to the design and evaluation of publisher/subscriber architectures for distributed real-time and embedded (DRE) applications. First, it illustrates how a flexible publisher/subscriber architecture can be implemented using standard CORBA middleware. Second, it shows how to extend the standard CORBA publisher/subscriber architecture so it is suitable for DRE applications that require low latency and jitter, periodic rate-based event processing, and event filtering and correlation. Third, it explains how to address key performance-related design challenges faced when implementing a publisher/subscriber architecture suitable for DRE applications. Finally, the paper presents benchmarks that empirically demonstrate the predictability, latency, and utilization of a widely used Real-time CORBA publisher/subscriber architecture. Our results demonstrate that it is possible to strike an effective balance between architectural flexibility and real-time quality of service for important classes of DRE applications.

Keywords: Real-time CORBA Event Systems, Object-Oriented Middleware Frameworks, Publisher/Subscriber Architectural Patterns.

1 Introduction

1.1 Challenges for DRE Applications

Distributed, real-time, and embedded (DRE) applications are becoming increasingly widespread and important. There are many types of DRE applications, but they have one thing in common: *the right answer delivered too late becomes the wrong answer*. Common DRE applications include telecommunication networks (*e.g.*, wireless phone services), tele-medicine (*e.g.*, remote surgery), manufacturing process automation (*e.g.*, hot rolling mills), and defense applications (*e.g.*, avionics mission computing systems).

*This work was supported in part by DARPA ITO, BBN, Boeing, DMSO, SAIC, and Siemens.

Due to constraints on weight, power consumption, memory footprint, and performance, the development techniques for DRE application software have historically lagged behind those used for mainstream desktop and enterprise software. As a result, DRE applications are costly to develop, maintain, and evolve. Moreover, they are often so specialized that they cannot adapt readily to meet new functional or quality of service (QoS) requirements, hardware/software technology innovations, or market opportunities.

Programming DRE applications is also hard because QoS properties must be supported along with the application software and distributed computing middleware functionality. DRE applications have historically been custom-programmed to implement these QoS properties. Unfortunately, this tedious and error-prone manual development process has not adequately addressed the following challenges:

Isolating DRE applications from the details of multiple platforms and varying operational contexts. Modern DRE applications must invest an ever-increasing proportion of functionality in software. Rapidly emerging technologies, together with the flexibility required for diverse operational contexts, force deployment of multiple versions of software on various platforms, while simultaneously preserving key properties, such as real-time response and end-to-end priority preservation.

Reducing total ownership costs. Custom software development and evolution is labor-intensive and error-prone for complex DRE applications and can represent a substantial amount of total system acquisition and maintenance costs. Since DRE applications are often upgraded multiple times during their lifetime, it can be hard to maintain the QoS properties of custom-made systems as new components and capabilities are added (1).

Sheltering application architectures from obsolescence trends. Incommensurate lifetimes between long-lived DRE applications (20 years or more) and commercial off-the-shelf (COTS) platforms and tools (2–5 years) lead to pervasive software obsolescence and multiply the total ownership costs by requiring periodic software redevelopment and reengineering (2).

Minimizing personnel risks. Acquiring, retaining, and training personnel to maintain and upgrade proprietary, custom-made DRE applications is risky since it can make organizations overly reliant on a small group of specialized software developers.

While considerable R&D effort has focused on how to meet these challenges for desktop and enterprise business applications (3), comparatively little effort has focused on how to meet these challenges for DRE applications with stringent QoS requirements.

1.2 Candidate Solution: Publisher/Subscriber Architectures

Addressing the challenges outlined above requires software architectures that can support a changing set of requirements and environments gracefully. Too often, developers find themselves reengineering existing software interfaces and implementations in response to planned and unplanned changes. In fact, most software costs occur *after* initial deployment (4). *Publisher/subscriber architectures* can help overcome many of these problems by reducing software dependencies and inflexibility. In this architecture, the components of a system are separated into the following three roles, in accordance with the Publisher/Subscriber pattern (5):

- **Publishers** are event sources, *i.e.*, they generate the events that are propagated through the system. Depending on architecture implementation, publishers may need to describe the type of events they generate *a priori*.
- **Subscribers** are the event sinks of the system. Some architecture implementations require subscribers to declare filtering information for the events they require.
- **Event channels** are components in the system that propagate events from publishers to subscribers. In distributed systems, event channels can propagate events across distribution domains to remote subscribers. Event channels can perform event filtering and routing, QoS enforcement, and fault management.

Figure 1 illustrates the relationships and information flow between these three components.

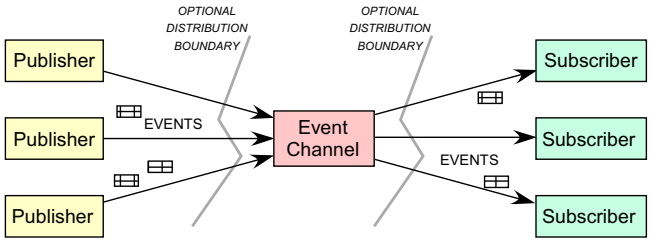

Figure 1: **Relationships Between Components in a Publisher/Subscriber Architecture**

Publisher/subscriber architectures can be used to address many of the challenges outlined in Section 1.1. In particular, they can help to:

Isolate DRE applications from the details of multiple platforms and varying operational contexts. Publisher/subscriber defines a communication model that can be implemented over many networks, transport protocols, and OS platforms. Developers of DRE applications can therefore concentrate on the application-specific aspects of their systems, and leave the communication and QoS-related details to developers of publisher/subscriber middleware.

Reduce total ownership costs. Publisher/subscriber architectures define clear boundaries between the components in the application, which reduces dependencies and thus maintenance costs associated with replacement, integration, and revalidation of components. Likewise, core components of these architectures can be reused, thereby helping to reduce development, maintenance, and testing costs.

Shelter application architectures from obsolescence trends. Publisher/subscriber architectures strongly decouple event sources from event sinks. Application developers can take advantage of this separation of concerns during the lifetime of the system, *e.g.*, new sources of events can be added to the system over time. In a tightly-coupled, monolithic design such changes would also require modifications to event sinks.

Minimize personnel risks. The roles and relationships of publisher/subscriber architectures are well documented and relatively easy to understand, which can help minimize personnel training costs. Solutions based on proprietary publisher/subscriber systems, however, fail to address this challenge completely. Fortunately, publisher/subscriber architectures based on standard middleware technologies, such as CORBA, Java/RMI, or COM+, are more effective at addressing this challenge.

1.3 Unresolved Challenges: Ensuring the QoS of Standard Publisher/Subscriber Architectures.

If implemented properly, publisher/subscriber architectures satisfy many DRE application challenges. However, many implementations of publisher/subscriber architectures rely on non-standard distribution and infrastructure middleware (6; 7). Comparatively little research has focused on applying (and potentially augmenting) standard middleware to publisher/subscriber architectures to meet the needs of DRE applications. Moreover, research that has focused on this topic (8; 9) has not documented the key patterns and frameworks required to implement publisher/subscriber architectures that can (1) preserve a clean separation of concerns while (2) simultaneously satisfying stringent DRE application QoS requirements.

In fact, there is a widespread belief in the commercial embedded systems community that modern software abstraction and composition techniques, such as object-oriented design and programming, application frameworks, and standard COTS components, are not suitable for DRE applications. Yet, many DRE application domains can leverage the benefits of flexible and open distributed object computing architectures, such as those defined in the CORBA specification (10). If publisher/subscriber architectures can be implemented in an efficient and predictable manner, therefore, the benefits outlined in Section 1.2 will make them a compelling choice for new and planned DRE applications.

In this paper, we describe the patterns, framework design, and performance of a publisher/subscriber architecture that supports real-time QoS for event-driven DRE applications. This architecture is

based on the Real-time CORBA standard (10) and the TAO real-time ORB (11). TAO is an open-source[1] implementation of Real-time CORBA that supports efficient, predictable, *and* flexible DRE applications. Our prior work on TAO has explored many dimensions of high-performance and real-time ORB design and performance, including optimal request demultiplexing (12), I/O subsystem (13) and protocol (14) integration, connection architectures (15), asynchronous (16) and synchronous (17) concurrent request processing, adaptive load balancing (18) and meta-programming mechanisms (19), and IDL stub/skeleton optimizations (20).

Our previous work (21) on publisher/subscriber architectures focused on the patterns and performance of a *highly scalable* CORBA Event Service implementation. This paper extends our previous work by describing how to augment the CORBA Event Service specification to satisfy the *real-time* needs of event-driven applications in many DRE domains, such as avionics mission computing (22), mission-critical distributed audio/video processing (23), and large-scale distributed interactive simulation (21). TAO and its Real-time Event Service are used in many production DRE applications. Two particular relevant examples include:

- **HLA RTI-NG** (21), which is the next-generation Run-time Infrastructure (RTI) implementation for the Defense Modeling and Simulation Organization's (DMSO) High Level Architecture (HLA) and
- **Boeing Bold Stroke** (24; 25), which uses COTS hardware and middleware to produce a standards-based component architecture for military avionics mission computing capabilities, such as navigation, data link management, and weapons control.

Both these DRE systems are examples of the need to simultaneously support multiple software qualities, such as maintainability, reusability, performance, availability, usability and time to market.

1.4 Paper Organization

The remainder of this paper is organized as follows: Section 2 describes how publisher/subscriber architectures can be mapped to standard middleware, in particular Real-time CORBA and the CORBA Event Service; Section 3 describes how to augment the CORBA Event Service to create a Real-time Event Service that can satisfy key QoS and flexibility requirements of DRE applications. Section 4 explains the design challenges we addressed and optimizations we applied when implementing TAO's Real-time Event Service; Section 5 shows the results of benchmarks conducted using TAO's Real-time Event Service to validate the design described in Section 3; Section 6 compares our results to related research on publisher/subscriber architectures; and Section 7 presents concluding remarks.

[1]The source code and documentation for TAO and its Real-time Event Service are freely available at URL `http://deuce.doc.wustl.edu/Download.html`.

2 CORBA and the CORBA Event Service

The TAO Real-time Event Service was not designed in a vacuum. Instead, it was designed to overcome limitations with standard CORBA ORBs and services. This section describes how the CORBA Event Service was designed to overcome limitations with CORBA ORB invocation models. Section 3 then describes how TAO's Real-time Event Service overcomes limitations with the CORBA Event Service.

2.1 Overview of CORBA

CORBA is a distributed object computing middleware specification (10) being standardized by the Object Management Group (OMG). CORBA supports the development of flexible and reusable service components and distributed applications by separating interfaces from (potentially remote) object implementations. Figure 2 illustrates the primary components in the CORBA architecture that

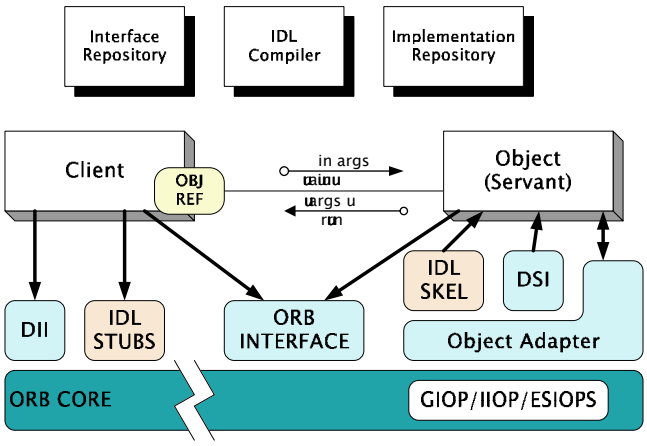

Figure 2: **Components in the CORBA ORB Architecture**

automate many common network programming tasks, such as object registration, location, and activation; request demultiplexing; framing and error-handling; parameter marshaling and demarshaling; and operation dispatching. Although it is beyond the scope of this paper to examine how the CORBA ORB architecture affects software quality and quality of service, these topics are explored in depth in our other publications.

The standard CORBA operation invocation model supports synchronous and asynchronous two-way, one-way, and deferred synchronous interactions between clients and servers. Two-way synchronous calls are the default invocation model. The primary strength of this model is its intuitive mapping onto the `object->operation()` paradigm supported by object-oriented languages.

2.2 Limitations with CORBA

In principle, two-way synchronous invocations simplify the development of distributed applications by supporting an implicit request/response protocol that makes remote operation invocations largely transparent to clients. In practice, however, the standard CORBA operation invocation models are overly restrictive for many DRE applications, due to their following limitations:

Tight coupling of client and server lifetimes. For a CORBA client request to be successful, the server must be available to process the request. If a request fails because the server is unavailable, the client receives an exception and must take corrective action, such as notifying an end-user or system administrator. In 2001 the OMG integrated the Messaging specification (26) into the CORBA 2.4 standard. This specification gave application developers more control over QoS parameters and introduced time-independent invocations (TII). Although these features solve several problems not addressed by earlier versions of CORBA, they require event suppliers to have explicit knowledge of the consumers of those events.

Synchronous invocation semantics. By default, a CORBA client waits until the server finishes processing a synchronous two-way request and returns the result to the client. This blocking can cause problems for DRE applications with stringent real-time constraints. CORBA therefore provides various non-synchronous invocation models, such as one-way invocations, deferred synchronous invocations, and asynchronous method invocations (AMI) (16). However, standard one-way invocations are not required to implement reliable delivery, deferred synchronous invocations yield excessive overhead for DRE applications since they use the CORBA Dynamic Invocation Interface (DII), and AMI still requires the server to be available when a client invokes a request.

Point-to-point communication. A CORBA operation invoked by a client is destined for a single target object on a particular server. The CORBA Fault Tolerance (27) specification relaxes this point-to-point protocol, but is not widely implemented nor widely used at this time. Moreover, the standard replication protocols used in Fault Tolerant CORBA are too heavyweight for many DRE applications.

2.3 Overview of the CORBA Event Service

To address the problems described in Section 2.2, developers often employ some type of publisher/subscriber architecture based on the abstractions outlined in Section 1.2. In a publisher/subscriber architecture, the data and control flow through *events*, rather than via parameterized operation invocations. Event-driven applications that use publisher/subscriber architectures possess several key requirements:

1. The events must be transferred efficiently from event sources to event sinks and
2. The event sources and sinks must be anonymous and decoupled, *i.e.*, the number and characteristics of the event sinks must be transparent to event sources and vice-versa.

To support these requirements, the OMG CORBA specification defines a standard Event Service (28) that provides 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 each other's identities explicitly. The CORBA Event Service can be used in conjunction with the CORBA Messaging specification to achieve fine grained control over QoS parameters, such as timeouts and request priorities. Figure 3 illustrates the relationships between components in the CORBA Event Service architecture.

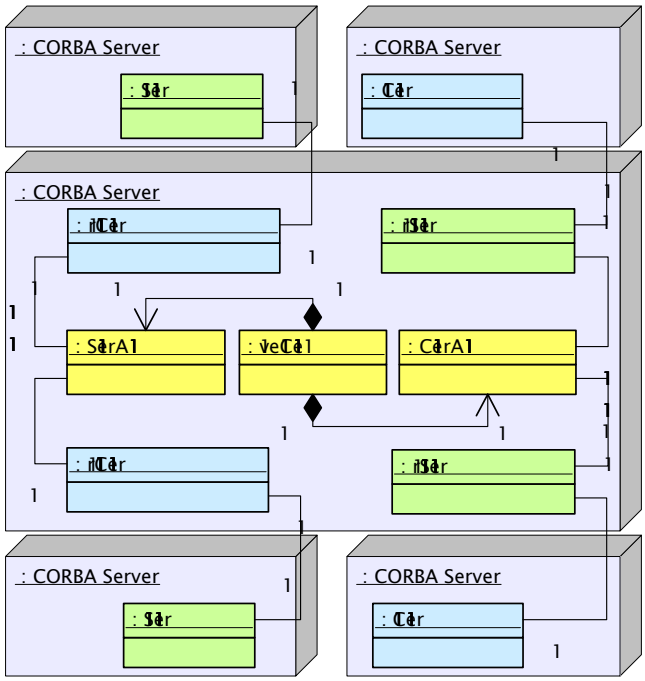

Figure 3: **CORBA Event Service Architecture**

The CORBA Event Service defines the following three roles:

- *Suppliers*, which produce event data, *i.e.*, they play the publisher role in the publisher/subscriber architecture,
- *Consumers*, which receive and process event data, *i.e.*, they play the subscriber role in the publisher/subscriber architecture, and
- *Event channels*, which are mediators (29) through which multiple consumers and suppliers communicate asynchronously, *i.e.*, they play the same role as the event channels in the publisher/subscriber architecture.

There are two models (*i.e.*, *push* vs. *pull*) of participant collaborations in the CORBA Event Service. 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. Events are transferred via standard CORBA operations from suppliers to an event channel, which in turn disseminates the events to consumers.

3 The TAO Real-time Event Service

3.1 Overcoming CORBA Event Service Limitations with TAO

Although the CORBA Event Service described in Section 2.3 provides a standard way to decouple event suppliers and event consumers and support asynchronous communication, it does not specify the following important features required by event-driven DRE applications:

1. Low latency/jitter event dispatching
2. Support for periodic processing
3. Centralized event filtering and event correlations and
4. Efficient use of network and computational resources.

To support these important features, we have developed a *Real-time Event Service* as part of the TAO project. As shown in Figure 4, TAO's Real-time Event Service is a component implemented atop TAO that augments the CORBA Event Service specification to satisfy the QoS needs of DRE applications. The following discussion

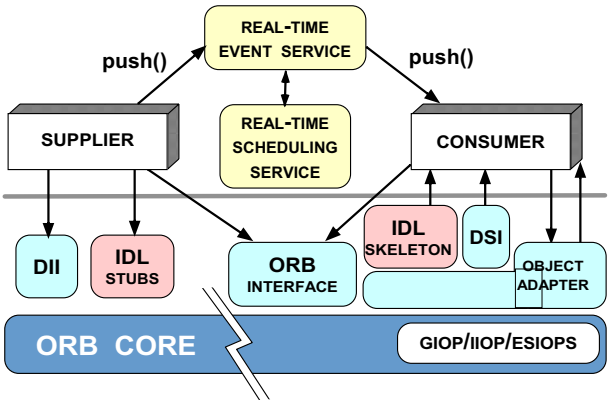

Figure 4: **Relationship of TAO and Its Real-time Event Service**

summarizes the features missing in the CORBA Event Service and outlines how they are supported by TAO's Real-time Event Service.

1. Low latency/jitter event dispatching. To minimize latency and jitter in DRE applications, it is desirable to have multiple threads within an event channel forwarding events to their consumers. The CORBA Event Service specification does not designate a standard threading model, much less a real-time threading model. As a result, DRE applications may not be able to rely on threading support.

TAO's Real-time Event Service can be configured with an application-specified strategy to assign the number and priority of real-time threads that will dispatch events. TAO also predefines several application-specified strategies to cover the most common cases. The same component can be used to assign events to a thread at the appropriate priority, avoiding priority inversions in the event channel and thus achieving greater predictability by enforcing scheduling decisions at run-time.

2. Support for periodic processing. Consumers in periodic real-time systems typically require C units of computation time every P milliseconds. For instance, some real-time avionics signal processing filters must be updated periodically or else they will spend a substantial amount of time reconverging (24). In this case, consumers have strict deadlines by which time they must execute the requested C units of computation time. The standard CORBA Event Service defines no interface that allows consumers to specify their temporal execution requirements. Periodic processing is therefore not supported in conventional CORBA Event Service implementations.

TAO's Real-time Event Service allows consumers to specify event dependency timeouts. It uses these timeout requests to propagate temporal events in coordination with system-wide scheduling policies. In addition to the canonical use of timeout events, *i.e.*, receiving timeouts at some interval, a consumer can request to receive a timeout event via a real-time "watchdog" timer if its dependencies are not satisfied within some time period.

3. Support for centralized event filtering and event correlation. In DRE applications, consumers may not be interested in all events generated by suppliers. Although it is possible to let each application perform its own filtering, this solution wastes network and computing resources and requires extra work by application developers. Ideally, an event service should send an event to a particular consumer only if the consumer has subscribed for it explicitly. Care must be taken, however, to ensure that the algorithms used to support filtering do not cause undue burden on DRE system resources.

To alleviate scalability and performance problems, TAO's Real-time Event Service provides filtering and correlation mechanisms that allow consumers to specify the type of events they are interested in via conjunction ("AND") and disjunctive ("OR") event dependencies. 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. Suppliers can also provide information about the types of events they generate. Based on the *subscriptions* provided by the consumers and the *publications* described by the suppliers, the event channel will dispatch events only to consumers that have expressed interest in them.

4. Efficient use of network and computational resources. Conventional implementations of an Event Service send one message for each remote consumer interested in an event. This point-to-multipoint design requires excessive network resources, since the same data is transmitted multiple times, often to the same target computer. TAO's Real-time Event Service can therefore be configured to minimize network traffic by:

- Using IP broadcast and multicast protocols to avoid duplicate network traffic, and
- Building federations of Event Services that eliminate the transmission of duplicate events, and even eliminate the need to send unwanted events by pushing filtering to the source.

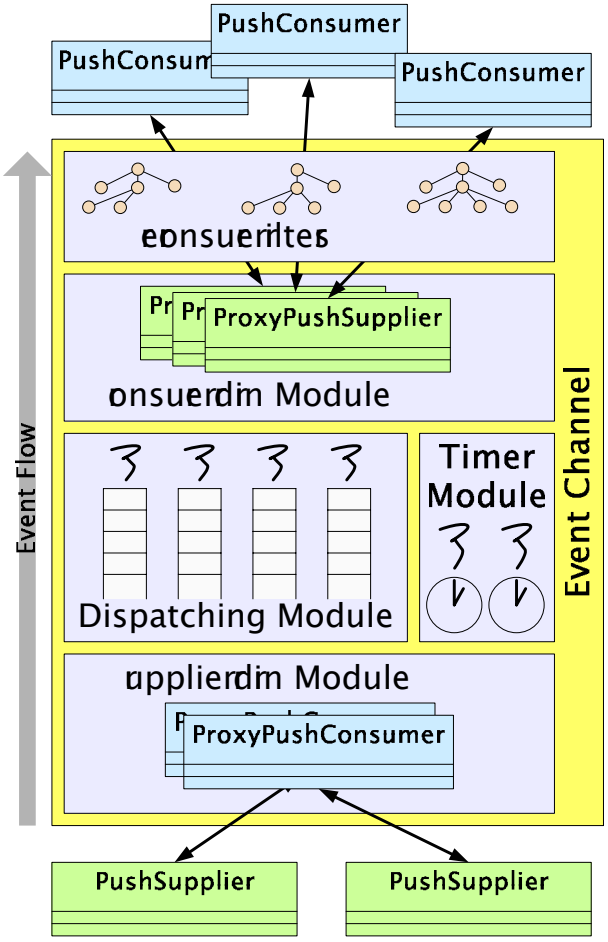

Figure 5: **TAO Real-time Event Service Architecture**

This paper focuses largely on features 1 and 2 described above since they are the most essential for real-time applications. Our previous work in (21) focuses on the scalability issues associated with features 3 and 4.

3.2 TAO's Real-time Event Service Architecture

Figure 5 shows the architecture of TAO's Real-time Event Service. At the heart of this architecture is the *event channel*, which provides two factory interfaces, `ConsumerAdmin` and `SupplierAdmin`, that allow applications to obtain consumer and supplier administration objects, respectively. These administration objects allow consumers and suppliers to connect/disconnect from the channel and to specify their QoS needs, such as their event dependencies and filtering types. Internally, the event channel is an object-oriented framework (30) that contains a series of processing modules linked together in accordance with the Pipes and Filters pattern (5), which provides a structure for systems that process a stream of event data. Each module encapsulates independent tasks of the channel, as described below.

Consumer admin module. The interface to the consumer admin module is identical to `ConsumerAdmin` interface defined by the `CosEventChannelAdmin` module in the CORBA Event Service shown in Figure 3. This interface provides factory methods for creating objects that support the `ProxyPushSupplier` interface. In the CORBA Event Service, the `ProxyPushSupplier` interface is used by consumers to connect and disconnect from the channel.

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

To address these shortcomings, TAO's Real-time Event Service extends the standard `ProxyPushSupplier` interfaces so that consumers can register their event dependencies with a channel. Consumers can specify conjunctive ("AND") or disjunctive ("OR") semantics when registering their supplier-based and/or type-based filtering requirements. TAO also allows consumers to subscribe for particular subsets of events. The channel uses these subscriptions to filter supplier events, only forwarding them to interested consumers.

Supplier admin module. The interface to this module is based on the `SupplierAdmin` interface defined in the CORBA 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 Real-time Event Service also extends the standard CORBA `ProxyPushConsumer` interface so that suppliers can specify the types of events they generate. With this information, the channel's correlation and filtering mechanism 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(s).

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 TAO's Real-time Scheduling Service (11; 22). For instance, consider the arrival of an event into a dispatching module implemented with real-time preemptive threads. If TAO's Real-time Scheduling Service assigns the event a preemption priority higher than any currently running thread, the dispatching module will preempt a running thread and dispatch the new event.

The dispatching module always dequeues events from the head of the queue. TAO's Real-time Scheduling Service can determine the order of dequeueing by returning different sub-priorities for different events. For instance, assume that an implementation of the 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 an event containing E_1, the event will always be queued before any event containing E_2. The dispatcher will therefore always dequeue and dispatch E_1 events before E_2 events.

The dispatching module can be configured to implement several concurrency strategies, such as real-time upcalls and preemptive multi-threading (31). 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.

Priority timers proxy. The supplier admin 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 ensures that timeouts are dispatched according to the priority of their corresponding consumer.

The priority timers proxy uses a heap-based callout queue, where the average and worst case time required to schedule, cancel, and expire a timer is *O(log N)* (*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 and is therefore well-suited for real-time systems requiring highly predictable and efficient timer operations.

3.3 Alternative Event Service Configurations

Although TAO's Real-time Event Service architecture described in Section 3.2 provides many powerful capabilities, the true test of its object-oriented framework arises when using it to support a diverse set of DRE applications with a wide range of functional and QoS requirements. The remainder of this section describes how TAO's Real-time Event Service can be configured both externally and internally. The flexibility of its architecture helps developers of DRE applications meet their requirements without incurring time and space overhead for capabilities they do not use.

3.3.1 Centralized vs. Federated Event Channels

The original implementation of TAO's Real-time Event Service (31) was limited to a single processor configuration. However, modern DRE applications typically connect multiple computers via high-speed interconnects, 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 6. A centralized real-time event channel architecture incurs

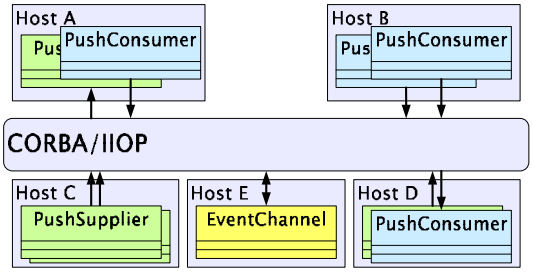

Figure 6: **A Centralized Configuration for the Event Channel**

excessive overhead, however, because the consumer for a given supplier is usually located on the same computer. If the event channel is on a remote computer, therefore, extra communication overhead and latency are incurred for the common case. Moreover, TAO's collocated operation invocation path is highly optimized (19), so it is desirable to exploit this optimization whenever possible.

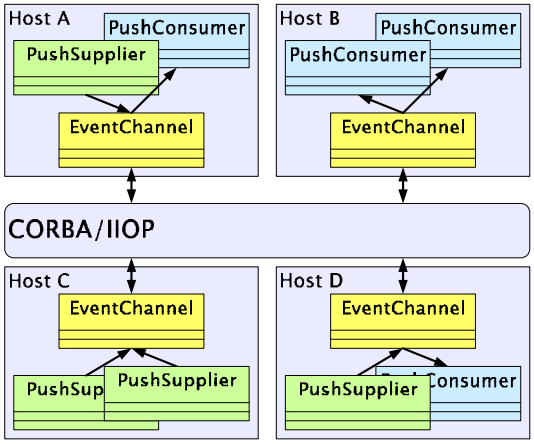

Figure 7: **Federated Event Channels**

To address the limitations of the centralized event channel architecture, TAO's Real-time Event Service provides mechanisms to connect several event channels to form a *federation*, as shown in Figure 7. In a federated group of event channels, suppliers and consumers just connect to their local event channel, while event channel instances talk to each other via CORBA. This design reduces average latency for all consumers in the system because consumers and suppliers exhibit locality-of-reference, *i.e.*, most consumers for any event are on the same computer as the supplier generating the event. Moreover, if multiple remote consumers are interested in the same event only one message is sent to each remote event channel, thereby minimizing network utilization.

A straightforward and portable way to implement this architecture is to use a *gateway* between each event channel. As shown in Figure 8, such gateways play both the consumer and supplier roles and mediate between two event channels. They connect (in their consumer role) to one of the event channels, ideally subscribing only for

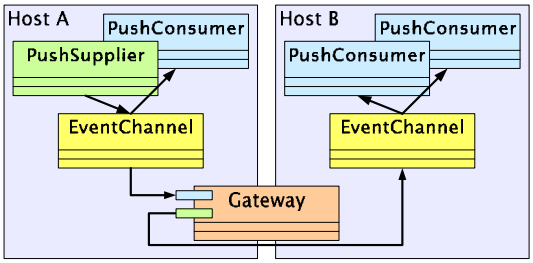

Figure 8: **Using a Gateway to Connect Two Event Channels**

the events that are interesting for the participants in the second event channel. When a gateway receives an event it forwards the event to the second event channel, where it has connected (in its supplier role).

Application developers can configure the location of the gateway with respect to its event channels to minimize the utilization of network resources. For example, collocating the gateway with its sink event channel, *i.e.*, the one it connects to as a supplier, eliminates the need to transmit events that are not interesting for the sink event channel. Collocating the gateway with its source event channel can avoid event cycles more efficiently than the previous configuration, however, as described in (21).

TAO's Real-time Event Service requires that each event channel in a federation be connected to every one of its peers. This is not a problem for DRE applications that have one event channel per computer. In certain domains, however, DRE applications could require hundreds or thousands of event channels. In this case, providing a full network would not scale. We are investigating techniques to address this situation based on related work (7).

Many types of distributed applications can benefit from TAO's federated Event Service. For instance, both distributed interactive simulations (21) and avionics mission computing systems (24; 25) can comprise several computers in different networks, where most of the traffic destination is within the same network. Configuring an event channel on each network helps to reduce latency by avoiding round-trip delay to remote computers.

3.3.2 Event Channel Configurations

Since the QoS requirements of DRE applications can vary considerably, the internals of event channels in TAO's Real-time Event Service can also vary accordingly. In particular, the internal architecture of the TAO event channel is based on the Pipes and Filters (5) and Builder (29) patterns to support different channel configurations optimized for different DRE application requirements. This architecture allows TAO's event channels to be configured in the following ways to support a wide range of event dispatching, filtering, and dependency semantics:

Full event channel configuration. A full event channel includes the dispatching module and consumer/supplier proxy modules, along with their full correlation and filtering mechanisms. A channel configured with all these modules supports type and source-based filtering, correlations, and priority-based queueing and dispatching. Figure 5 illustrates a full event channel configuration.

Subset event channel configurations. TAO's Real-time Event Service supports subset configurations that allow processing modules and mechanisms to be added, removed, or modified with minimal impact on the overall system. The following configurations can

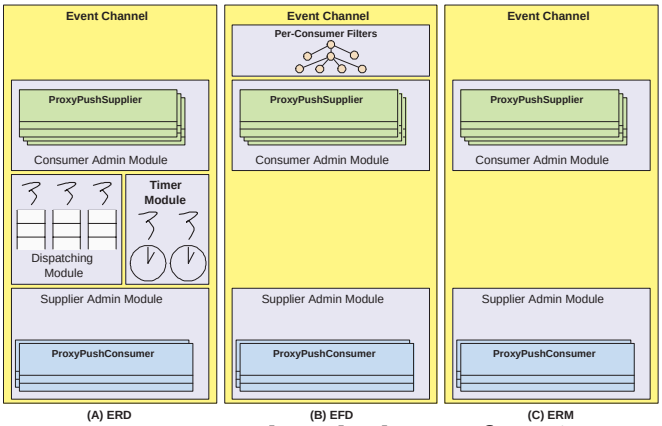

Figure 9: **Event Channel Subset Configurations**

be achieved by removing certain modules and mechanisms from an event channel:

- **Event real-time dispatching (ERD) configuration.** Removing the correlation and filtering capabilities creates an ERD configuration, which is shown in Figure 9 (A). This configuration supports "classic" real-time applications that require no correlation or filtering.
- **Event forwarding discriminator (EFD) configuration.** Removing the dispatching module from the event channel yields an EFD configuration that supports event filtering and correlations, as shown in Figure 9 (B). EFDs provide a "data reduction" mechanism that minimizes the number of events received by consumers so they only receive events of interest. An EFD configuration helps improve the scalability of applications that do not require priority-based queueing and dispatching in the event channel.
- **Event registration multiplexer (ERM) configuration.** Removing both the correlation/filtering *and* dispatching capabilities creates an ERM configuration, which is shown in Figure 9 (C). This configuration supports neither real-time dispatching nor filtering/correlations. In essence, this implements the semantics of the standard CORBA event channel push model.

In mission-critical DRE environments with stringent QoS requirements, such as avionics mission computing systems, the configuration of an event channel is performed off-line to minimize startup overhead. TAO's Real-time Event Service framework uses the Pipes and Filters pattern and the Builder pattern (29) (described in Challenge 4 in Section 4) to configure the event channel off-line. 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. TAO's Real-time Event Service framework uses the Component Configurator pattern (32) (described in Challenge 5 in Section 4) to support dynamic reconfiguration of event channels without interruption while continuing to process events.

4 Designing and Optimizing TAO's Real-time Event Service

Although publisher/subscriber architectures have many benefits, they can also have some disadvantages:

- Their modularity can introduce excessive overhead, *e.g.*, inefficiencies may arise if buffer sizes are not consistently sized and aligned between and within event channels, thereby causing additional segmentation, reassembly, and transmission delays
- Information hiding within components can make it hard to manage resources predictably and dependably in DRE applications and
- Communication between suppliers and consumers must be designed and implemented properly to avoid introducing subtle source of errors.

Publisher/subscriber *architectures* alone are therefore not sufficient to resolve key challenges of DRE applications. What is needed is a deeper understanding of the patterns and optimization techniques necessary to develop flexible *and* QoS-enabled publisher/subscriber software.

Architectural flexibility has historically been associated with excessive time and space overhead, which is antithetical to DRE applications. Fortunately, new generations of hardware and advances in patterns and optimizations are becoming mature enough to compensate for much of the time and space overhead traditionally associated with architectural flexibility and modern software abstraction and composition techniques (1). To reify this point, this section describes the patterns and idioms we applied to address the following design and performance challenges encountered when developing TAO's Real-time Event Service:

1. Ensuring timeliness in event processing
2. Minimizing interference between the event channel components
3. Optimizing the performance of the CORBA Any type
4. Optimizations for footprint reduction and
5. Customizing event channels for particular deployment environments.

These challenges and our solutions are discussed below. To enhance the generality of our solutions, we describe them in terms of the patterns (29; 32) we used to resolve the challenges. Section 5 presents empirical results that quantify the benefits of these patterns and optimizations.

Challenge 1: Ensuring Timeliness in Event Processing

Context. The dispatching module of TAO's real-time event channel ensures that priorities are respected by enqueueing events in an internal buffer according to their priority.

Problem. Long-duration operations, such as complex filter evaluation, can starve other events in the queue and prevent them from being processed in a timely manner.

Solution → the Command Object pattern. This pattern encapsulates an event as an object, thereby allowing parameterization of different events (29). The actual representation of the event is hidden by the command object interface. Different concrete implementation of this interface implement the event and provide semantics to it. This pattern can be used to decompose the internal event processing within an event channel into stages to ensure timeliness by avoiding long-duration operations that would otherwise incur "head of line blocking."

Applying the Command Object pattern. The filter evaluation, subscription lookup, and event dispatching operations in TAO's event channel are encapsulated as command objects. As shown in Figure 10 instead of performing these operations synchronously, their

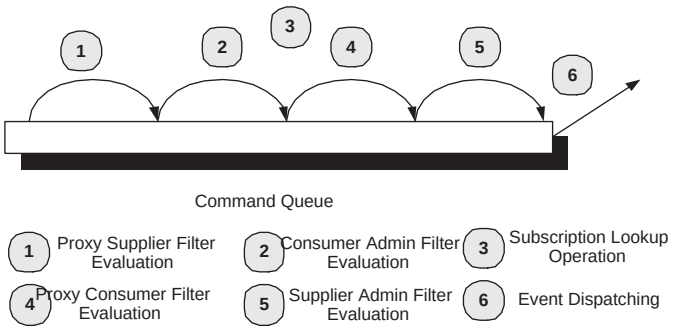

Figure 10: **Processing Command Objects to Ensure Timeliness**

evaluation is split into discrete steps. If the event is still eligible for further processing after executing an command operation, it is replaced into the command queue and dequeued subsequently when further processing is possible. The dispatching command is responsible for sending the event to its consumer.

Challenge 2: Minimizing Interference Between Event Channel Components

Context. An event channel should be able to (1) handle event delivery from suppliers and (2) perform event forwarding with the minimum possible latency, *i.e.*, suppliers delivering an event to the channel should not have to wait while the channel forwards events to recipient consumers. Similarly, when an event channel forwards events to multiple consumers, each consumer might spend an unbounded amount of time in the implementation of its `push()` operation. Since events are forwarded by the event channel via CORBA

two-way operations, the channel's dispatching thread must wait until the consumer returns from its upcall. This blocking overhead affects the event channel's event processing time.

Problem. A real-time event channel implementation must minimize the interference between different components of the event channel.

Solution → the Active Object pattern. This pattern decouples method execution from method invocation to simplify synchronized access to an object that resides in its own thread (32). The Active Object pattern allows one or more independent threads of execution to interleave their access to data modeled as a single object.

Applying the Active Object pattern. Using the Active Object pattern at the various stages of event processing enables the minimization of the interference between TAO's real-time event channel components. As shown in Figure 11 and discussed in Challenge 1 above, events and operations performed on them are encapsulated as com-

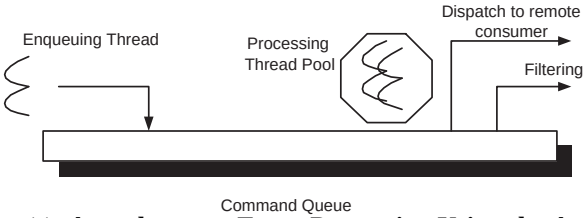

Figure 11: **Asynchronous Event Processing Using the Active Object Pattern**

mand objects. Enqueueing thread(s) place event command objects into a command queue according to a buffering order policy. An active object with worker threads dequeues and executes the command objects in the queue.

At a lower-level of abstraction, the TAO real-time ORB itself can be configured to increase concurrency using the Leader/Followers pattern (32), which provides an efficient concurrency model where multiple threads take turns sharing a set of event sources in order to detect, demultiplex, dispatch, and process service requests that occur on the event sources. In this case, each ORB invocation is handled by a separate thread, allowing multiple events to be delivered concurrently to a real-time event channel.

Challenge 3: Optimizing the Performance of Anys

Context. A CORBA "Any" is a dynamically typed data structure that can contain the typecode and data for any type of data supported by CORBA. A CORBA-compliant Real-time Event Service must process events containing Anys.

Problem. In CORBA, Anys are expensive data types because they can have many levels of nesting. For example, an Any can contain a structure, which can itself be contain a sequences of Anys and so forth. When a CORBA demarshaling engine decodes this expensive type, it must make a copy of the entire data buffer containing the Any. Likewise, copying an Any can require several memory allocations and buffer copys to obtain a new representation. Moreover, the C++ mapping of CORBA Anys requires them to be responsible for any memory returned to the application. Optimized ORBs should share the Any contents even if there are multiple copies of the Any object.

Solution → Reference counting via the Handle/Body idiom. This idiom presents multiple logical copies of the same data while sharing the same physical copy (33). In C++, this idiom is often used to automate the memory management in conjunction with reference counting and smart pointers.

Applying the Handle/Body idiom. The TAO demarshaling engine does not copy data into an Any. Instead, it reference counts its data buffers and the Any only increments the reference count to maintain a logical copy of the buffer. Likewise, after the contents of the Any are extracted by an application, the Any object itself is responsible for deallocating the extracted object. This extracted object can be shared by multiple instances of the Any object, minimizing the cost of copying and extracting the contents repeatedly. The use of the Handle/Body idiom implements this optimization without changing the semantics required by the standard CORBA C++ mapping.

Challenge 4: Optimizations for Footprint Reduction

Context. TAO's Real-time Event Service has many features that may not be required by all applications. For example, deeply embedded systems may not want to incur the increase in memory footprint for unused features, such as correlation and filtering.

Problem. A required set of real-time event features should be "composable" by users.

Solution → the Pipes and Filters pattern and the Builder pattern. The Pipes and Filters pattern (5) provides a structure for systems that process a stream of data. The Builder pattern (29) separates the construction of a complex object from its representation so that the same construction process can create different representations.

Applying the Pipes and Filters pattern and the Builder pattern. TAO's real-time event channel uses the Pipes and Filters pattern to configure alternative implementations of its modules described in Section 3.2. Likewise, it uses the Builder pattern to configure its correlation and filtering mechanisms. For example, a configuration containing *no correlation/filtering* + *AnyProxySupplier* + *AnyProxyConsumer* + *reactive dispatching strategy* would yield the default semantics of the CORBA Event Service.

These features are separated into libraries as follows:

- The three different pairs of proxy supplier and proxy consumer types are separated into different libraries. In a specific configuration, only the required type, *e.g.*, the `PushProxySupplier`, may be loaded by a builder at startup. Hence, the other types of proxy implementations are not loaded since they are not needed.

- An application may not require filtering, in which case the filtering engine library is not configured by the builder.
- The dispatching module can be configured to use simple reactive dispatching, multi-threaded dispatching, or even omitted altogether to create one of the subset event channel configurations described in Section 3.3.2.

Challenge 5: Customizing Event Channels for Particular Deployment Environments

Context. TAO's CORBA Real-time Event Service is configurable in the following manner:

1. A user can specify features required by configuration.
2. A specific class implementation can be modified by the user to enhance or customize behavior.
3. Users can vary default properties, such as thread pool size and locking strategy.

Problem. A mechanism is needed to allow application developers to change various configurable option in TAO's real-time event channel at run-time.

Solution → the Component Configurator pattern. This pattern decouples the behavior of component services from the point in time at which service implementation are configured into an application (32). This pattern can be implemented using *explicit dynamic linking*, which allows an application to obtain, use, and/or remove the run-time address bindings of certain function- or data-related symbols defined in DLLs. Common explicit dynamic linking mechanisms include the POSIX/UNIX functions `dlopen()`, `dlsym()`, and `dlclose()` and the Win32 functions `LoadLibrary()`, `GetProcAddress()`, and `FreeLibrary()`.

Applying the Component Configurator. All objects in TAO's Real-time Event Service implementation are created via factory objects. These factories can be loaded statically or dynamically. Figure 12 shows how the Component Configurator pattern can be used to dynamically configure TAO's real-time event channel. The configuration file contains a script with directives that designate which libraries, such as the filter library, dispatching library, and proxy library, to dynamically link into the address space of TAO's real-time event channel.

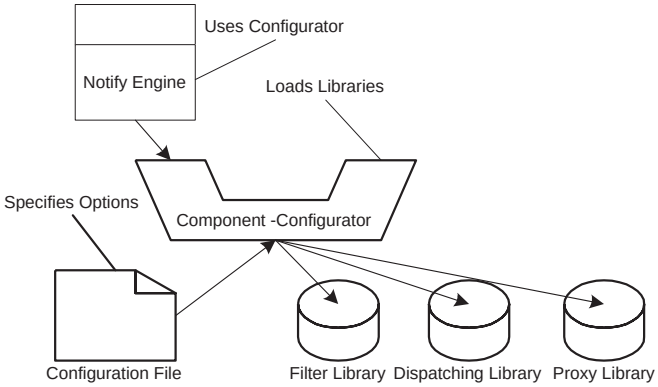

Figure 12: **Apply the Component Configurator Pattern in TAO's Real-time Event Channel**

5 Empirical Results

Our previous work (31) benchmarked a prototype of TAO's Real-time Event Service that did not run on a CORBA-compliant ORB. This section extends these benchmarks and also demonstrates the performance of mature Real-time CORBA and Real-time Event Service implementations. We first measure the CPU utilization of two event channel configurations described in Section 3.3.1. We then measure several aspects of event channel latency using the ERD configuration described in Section 3.3.2. The benchmarks reported in this section are based on performance requirements gleaned from our extensive work (11; 12; 20; 22) on real-time avionics mission computing systems (24; 25). Since our focus in this paper is on real-time properties, rather than the scalability properties described in (21), we do not report correlation or filtering performance here.

All benchmarks were conducted on a single-CPU 300 Mhz[2] Sun UltraSPARC 30 workstation with 256 MB RAM running Solaris 5.7. Version 1.1 of the TAO ORB, TAO's Real-time Event Service, and the test application were built with SunC++ 5.2 with the highest level (`-fast`) of optimization enabled. We focus our experiments on a single CPU hardware configuration to factor out network interface driver overhead and isolate the effects of ORB middleware and application latency, predictability, and utilization. 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) (34).

5.1 CPU Utilization Measurements

Overview. For non-real-time event channels, such as the EFD configuration described in Section 3.3.2, correctness implies that consumers receive events when their source/type subscription and correlation dependencies are met. In contrast, for real-time event channels, such as the full event channel and ERD configurations, correctness implies that all deadlines are met. An important metric for evaluating the performance of a real-time system is its *schedulable bound*, which is the maximum resource utilization possible without missing deadlines (35). The schedulable bound of TAO's Real-time Event Service is therefore the maximum CPU utilization that suppliers and consumers can achieve without missing deadlines.

With rate monotonic scheduling, higher rate tasks are supposed to preempt lower rate tasks. For TAO's Real-time Scheduling Ser-

[2]We chose a 300 Mhz CPU for the benchmarks since it is similar to the CPU speeds on many DRE platforms, such as avionics mission computing (24; 25).

vice to guarantee the schedulability of a system (*i.e.*, that all tasks meet their deadlines), its high-priority tasks must therefore preempt its lower priority tasks. We therefore devised tests to (1) determine whether this is indeed the case and (2) to measure the overhead of TAO's federated event channel configuration compared with a single collocated event channel. Two experiments were conducted: the first measured the utilization of a single event channel configuration and the second measured the utilization of a federated event channel configuration.

Single event channel utilization results. This 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 13 we plot the

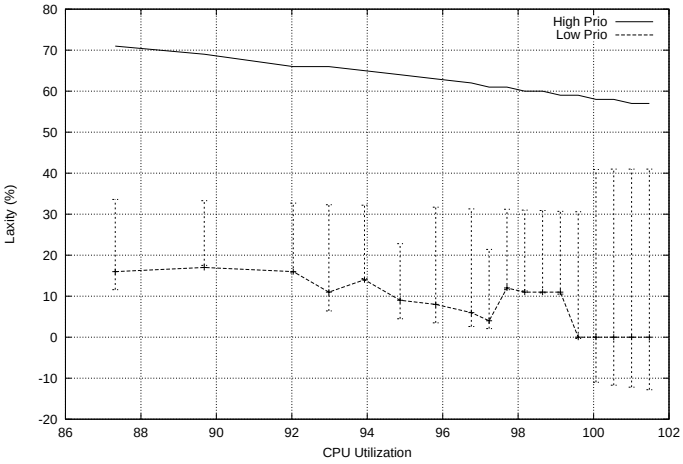

Figure 13: **CPU Utilization for a Single Event Channel**

average laxity[3] for high-priority and low-priority events. The error bars represent the minimum and maximum laxity for each experiment. Negative laxity means that a deadline was missed.

Figure 13 shows that the event channel achieved over 99% utilization before its low-priority task began to miss deadlines. The high-priority task never misses its deadlines, though its laxity decreases slightly as utilization increases. It is interesting to observe that this decrease in almost linear with the utilization increase, even when the low-priority task no longer meets its deadlines.

Federated event channel utilization results. In this experiment, two event channels were configured in separate OS processes on the same computer. No work was performed when processing remote events, but the processing time for high-priority events was increased until deadlines were missed. Figure 14 depicts the laxity for the high-priority and low-priority tasks. Although the performance of the federated event channel is lower than the single event channel, it still maintains high utilization (over 96%) before missing deadlines. As shown in Section 5.2, this small ($\sim$3%) loss of utilization yields a substantial performance improvement for the common case, where

[3]Laxity is defined as the time-to-deadline minus the execution time.

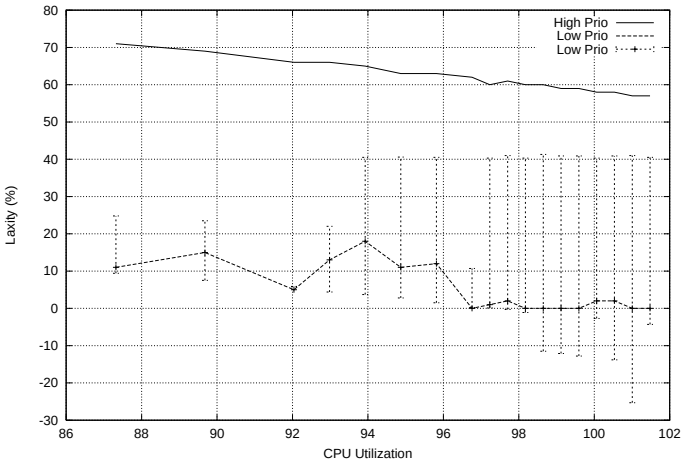

Figure 14: **CPU Utilization for a Federated Event Channel**

events are exchanged between local (collocated) suppliers and consumers.

Analysis of results. The results of these two experiments indicate the following:

- Both event channel configurations properly enforce the real-time distinctions between low- and high-priority suppliers and consumers. This enforcement stems from the design of TAO's Real-time Event Service dispatching module described in Section 3.2.
- The optimizations described in Section 4 are effective at minimizing the overhead of the object-oriented framework used to implement TAO's event channel so as not to degrade its utilization unduly.

5.2 Latency Measurements

5.2.1 End-to-end Latency Test

Overview. Another important measure of event channel performance is the latency it introduces between suppliers and consumers. To determine this latency for TAO's Real-time Event Service, we developed a test that measures the end-to-end supplier $\rightarrow$ consumer latency using the IIOP communication mechanism, which uses point-to-multipoint event delivery rather than IP multicast. 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 nothing with the event other than store the measurement in a preallocated array.

Single process latency results. In this test, the consumers, suppliers, and event channel were collocated in the same process to eliminate ORB remote communication overhead. In the single process case, the best-case supplier-to-consumer latency was $\sim$50 μsecs. In each case, as the number of suppliers and/or consumers increased, the latency also increased, as shown in Table 1.

		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 for Collocated Event Processing**

		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 for Local and Remote Event Processing**

Two process latency results. In this test, two identical processes were created and the consumers in each process subscribed to both local and remote events.[4] Table 2 shows the results of this test. For two processes, the local events exhibit similar latency to the local events in the single process case. In particular, no significant overhead is incurred due to possible remote consumers. In contrast, the remote consumer performance is much higher than the local events, though it is close to the performance of a remote operation invocation.

Analysis of results. The results of the experiments above indicate the following:

- Table 1 illustrates the efficiency of the optimizations described in Section 4. In particular, as the number of suppliers and consumers increase, the increase in latency is less than linear, due in large part to the Handle/Body idiom used to optimize the processing of Any data types. Naturally, the use of IP multicast should even further reduce latency, though TAO's event channel only supports unreliable multicast currently.
- Table 2 illustrates the benefits of the federated event channel configuration described in Section 3.3.1. In particular, disseminating events to consumers collocated in the same process is one to two orders of magnitude more efficient than disseminating them remotely.

[4] In this test configuration, a "local" event is one intended for a consumer collocated within the same process and a "remote" event is one intended for a consumer located in the other process.

5.2.2 Minimal Event Spacing Test

Overview. Another important performance metric is the *minimum event spacing*, which is the maximum rate an event channel can deliver messages before its overhead incurs measurable 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.

Minimal event spacing test results. Figure 15 shows that the

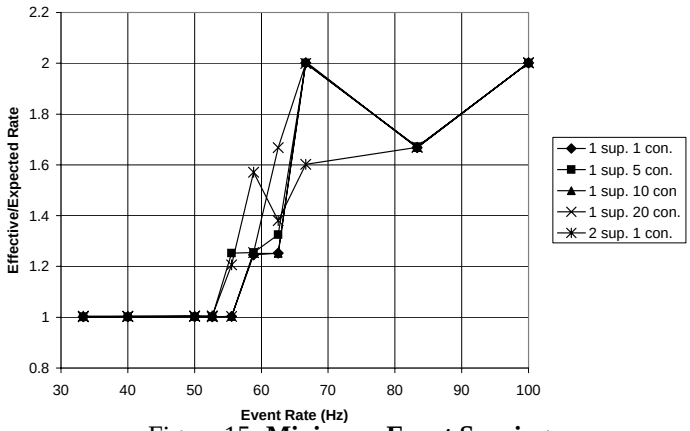

Figure 15: **Minimum Event Spacing**

event channel can deliver over 50 messages per second (*i.e.*, a 50 Hz rate) before it experiences any measurable overhead.

Analysis of results. These results indicate the "class" of DRE systems that can be supported by TAO's Real-time Event Service. In particular, DRE systems that run at rates less than 50 Hz (which includes the important class of real-time avionics mission computing systems (24; 25)) should incur inconsequential amounts of latency due to the overhead of the event channel. However, DRE systems that run at higher rates (which includes flight control software and automative breaking systems) may incur too much overhead to operate within their schedulable bound. We believe that TAO's Real-time Event Service performance can be improved, and are currently experimenting with additional patterns and optimizations to reduce its overhead systematically.

6 Related Work

Event-driven middleware for distributed real-time and embedded applications is an emerging field of study. An increasing number of research efforts are focusing on end-to-end QoS properties, such as timeliness, by integrating QoS management policies and mechanisms into publisher/subscriber middleware. This section describes the growing body of work related to CORBA-based Event Services, which we compare and contrast to our work with TAO.

OMG standard specifications. The OMG has specified a Notification Service (36), which is a superset of the CORBA Event Service that adds interfaces for event filtering, configurable event delivery semantics (*e.g.*, at least once or at most once), security, event channel federations, and event delivery QoS. The patterns and techniques used in the implementation of TAO's Real-time Event Service can be used to improve the performance and predictability of Notification Service implementations. To explore that idea, we have implemented a Notification Service for TAO (37) and used it to validate the feasibility of building a reusable framework that factors out common code for TAO's Notification Service, its standard CORBA Event Service implementation, and its Real-time Event Service.

The OMG Messaging specification (26) gives application developers control over several QoS parameters, such as one-way reliability and timeouts, and introduces type-safe asynchronous method invocation (AMI) models. The CORBA AMI specification solves many problems with the original CORBA invocation model, but it does not address anonymous or single-point-to-multiple-point communication. The Messaging specification can complement implementations of the CORBA Event Service, *e.g.*, it defines several levels of reliability for one-way calls. This feature could be used in Event Service implementations to improve decoupling of the clients, without the risk of losing messages. We have augmented TAO with the AMI features (16) defined by the Messaging specification, which complement its Real-time Event Service implementation.

COBEA. COBEA (8) is a CORBA-based event architecture service that generates parameterized events, which are published by a trading service. For scalability, clients must register their interest with the service, at which point an access control check is performed. Subsequently, whenever a matching event occurs, the client is notified. As with TAO's Real-time Event Service the authors propose a number of extensions to support event filtering and correlation. However, COBEA does not takes advantage of the broadcast capabilities of modern networks to reduce traffic, nor does COBEA use multicast to offload processing from the CPU to the network cards.

Fault-tolerant Notification Service In (9) the authors study the fault tolerance capabilities provided by the CORBA Notification Service and propose a configuration that can achieve the highest event delivery guarantees. The authors then examine the performance of such configuration of the Notification Service under different loads. TAO's Real-time Event Service has been designed to satisfy the requirement of high-performance real-time systems and of highly-scalable distributed interactive simulations. In these environments, reliability is commonly obtained via other means, such as hardware redundancy. Nevertheless, we believe that extending TAO's Real-time Event Service to provide higher degrees of reliability is possible, and we are pursuing this topic in our future work on FaultTolerant CORBA (27) and DOORS (38).

7 Concluding Remarks

Minimizing coupling between components is an important means to fulfill key quality requirements in software-intensive applications. Many applications use publisher/subscriber architectures to deliver events from suppliers to consumers without introducing excessive coupling between event sources and sinks. Though the concepts, programming abstractions, and benefits of publisher/subscriber architectures are well understood, there has been relatively little research on how to design architectures that are efficient and predictable enough to meet the quality of service (QoS) requirements of distributed real-time and embedded (DRE) applications. In particular, the QoS performance tradeoffs between different publisher/subscriber configurations are not well understood.

Many DRE applications require support for anonymous, asynchronous, predictable, and scalable event-based communication. The CORBA Event Service defines a standard publisher/subscriber architecture where event channels dispatch events to consumers on behalf of suppliers. The TAO Real-time Event Service described in this paper augments the CORBA Event Service by providing low latency and low jitter dispatching, support for periodic real-time processing, source-based and type-based filtering and event correlations, and efficient use of network and computational resources.

The empirical results presented in Section 5 illustrate that systematically applying key patterns and optimizations make it feasible to apply Real-time CORBA middleware to important classes of DRE applications. The flexibility and QoS offered by TAO's Real-time Event Service have made it the foundation for many research and production DRE applications. Our future work is focusing on the patterns and optimization techniques necessary to support even more demanding DRE applications that run at higher rates and that must simultaneously handle multiple QoS properties, such as dependability, scalability, predictability,and security.

Acknowledgments

We would like to thank the SAIC RTI-NG group, particularly Steve Bachinsky and Russ Noseworthy, and Boeing, particularly Bryan Doerr, for their support and direction.

References

[1] L. Lundberg, D. Hggander, and W. Diestelkamp, "Conflicts and Trade-offs between Software Performance and Maintainability," in *Performance Engineering. State of the Art and Current Trends*, Lecture Notes in Computer Science, Springer Verlag, 2000.

[2] J. K. Cross and D. C. Schmidt, "Applying the Quality Connector Pattern to Optimize Distributed Real-time and Embedded Middleware," in *Patterns and Skeletons for Distributed and Parallel Computing* (F. Rabhi and S. Gorlatch, eds.), Springer Verlag, 2002.

[3] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns*. Boston: Addison-Wesley, 2002.

[4] L. Bass, P. Clements, R. Kazman, and K. Bass, *Software Architecture in Practice*. Boston: Addison-Wesley, 1998.

[5] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture – A System of Patterns*. New York: Wiley and Sons, 1996.

[6] 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.

[7] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf, "Design and Evaluation of a Wide-Area Event Notification Service," *ACM Transactions on Computer Systems*, vol. 19, pp. 332–383, Aug. 2001.

[8] 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.

[9] Srinivasan Ramani and Balabrishnan Dasarathy and Kishor S. Trivedi, "Reliable Messaging Using the CORBA Notification Service," in *Proceedings of the 3^{rd} International Symposium on Distributed Objects and Applications (DOA 2001)*, Sept. 2001.

[10] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.6 ed., Dec. 2001.

[11] 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.

[12] A. Gokhale and D. C. Schmidt, "Measuring and Optimizing CORBA Latency and Scalability Over High-speed Networks," *Transactions on Computing*, vol. 47, no. 4, 1998.

[13] F. Kuhns, D. C. Schmidt, C. O'Ryan, and D. Levine, "Supporting High-performance I/O in QoS-enabled ORB Middleware," *Cluster Computing: the Journal on Networks, Software, and Applications*, vol. 3, no. 3, 2000.

[14] C. O'Ryan, F. Kuhns, D. C. Schmidt, O. Othman, and J. Parsons, "The Design and Performance of a Pluggable Protocols Framework for Real-time Distributed Object Computing Middleware," in *Proceedings of the Middleware 2000 Conference*, ACM/IFIP, Apr. 2000.

[15] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, and A. Gokhale, "Software Architectures for Reducing Priority Inversion and Non-determinism in Real-time Object Request Brokers," *Journal of Real-time Systems, special issue on Real-time Computing in the Age of the Web and the Internet*, vol. 21, no. 2, 2001.

[16] A. B. Arulanthu, C. O'Ryan, D. C. Schmidt, M. Kircher, and J. Parsons, "The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging," in *Proceedings of the Middleware 2000 Conference*, ACM/IFIP, Apr. 2000.

[17] C. O'Ryan, D. C. Schmidt, F. Kuhns, M. Spivak, J. Parsons, I. Pyarali, and D. L. Levine, "Evaluating Policies and Mechanisms to Support Distributed Real-Time Applications with CORBA," *Concurrency and Computing: Practice and Experience*, vol. 13, no. 2, pp. 507–541, 2001.

[18] O. Othman, C. O'Ryan, and D. C. Schmidt, "An Efficient Adaptive Load Balancing Service for CORBA," *IEEE Distributed Systems Online*, vol. 2, Mar. 2001.

[19] N. Wang, D. C. Schmidt, M. Kircher, and K. Parameswaran, "Towards a Reflective Middleware Framework for QoS-enabled CORBA Component Model Applications," *IEEE Distributed Systems Online*, vol. 2, July 2001.

[20] A. Gokhale and D. C. Schmidt, "Optimizing a CORBA IIOP Protocol Engine for Minimal Footprint Multimedia Systems," *Journal on Selected Areas in Communications special issue on Service Enabling Platforms for Networked Multimedia Systems*, vol. 17, Sept. 1999.

[21] C. O'Ryan, D. C. Schmidt, and J. R. Noseworthy, "Patterns and Performance of a CORBA Event Service for Large-scale Distributed Interactive Simulations," *International Journal of Computer Systems Science and Engineering*, vol. 17, Mar. 2002.

[22] 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, March 2001.

[23] D. A. Karr, C. Rodrigues, Y. Krishnamurthy, I. Pyarali, and D. C. Schmidt, "Application of the QuO Quality-of-Service Framework to a Distributed Video Application," in *Proceedings of the 3rd International Symposium on Distributed Objects and Applications*, (Rome, Italy), OMG, September 2001.

[24] D. C. Sharp, "Reducing Avionics Software Cost Through Component Based Product Line Development," in *Proceedings of the 10th Annual Software Technology Conference*, Apr. 1998.

[25] B. S. Doerr and D. C. Sharp, "Freeing Product Line Architectures from Execution Dependencies," in *Proceedings of the 11th Annual Software Technology Conference*, Apr. 1999.

[26] Object Management Group, *CORBA Messaging Specification*. Object Management Group, OMG Document orbos/98-05-05 ed., May 1998.

[27] Object Management Group, *Fault Tolerant CORBA Specification*, OMG Document orbos/99-12-08 ed., December 1999.

[28] Object Management Group, *CORBAServices: Common Object Services Specification, Updated Edition*. Object Management Group, Dec. 1998.

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

[30] R. Johnson, "Frameworks = Patterns + Components," *Communications of the ACM*, vol. 40, Oct. 1997.

[31] 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), pp. 184–199, ACM, October 1997.

[32] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects, Volume 2*. New York: Wiley & Sons, 2000.

[33] Jim Coplien, *Advanced C++: Programming Styles and Idioms*. Addison-Wesley, 1992.

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

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

[36] Object Management Group, *Notification Service Specification*. Object Management Group, OMG Document telecom/99-07-01 ed., July 1999.

[37] P. Gore, R. K. Cytron, D. C. Schmidt, and C. O'Ryan, "Designing and Optimizing a Scalable CORBA Notification Service," in *Proceedings of the Workshop on Optimization of Middleware and Distributed Systems*, (Snowbird, Utah), pp. 196–204, ACM SIGPLAN, June 2001.

[38] B. Natarajan, A. Gokhale, D. C. Schmidt, and S. Yajnik, "DOORS: Towards High-performance Fault-Tolerant CORBA," in *Proceedings of the 2^{nd} International Symposium on Distributed Objects and Applications (DOA 2000)*, (Antwerp, Belgium), OMG, Sept. 2000.