The Journal of Systems and Software 66 (2003) 213–223

The Journal of Systems and Software

www.elsevier.com/locate/jss

# Patterns and performance of distributed real-time and embedded publisher/subscriber architectures ☆

Douglas C. Schmidt *, Carlos O'Ryan

*Department of Electrical and Computer Engineering, University of California at Irvine, 305 Engineering Tower, ZOT 2625, Irvine, CA 92697-2625, USA*

Received 16 January 2002; received in revised form 9 April 2002; accepted 5 June 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; Publisher/subscriber architectural patterns

## 1. Introduction

Due to constraints on weight, power consumption, memory footprint, and performance, the development techniques for distributed real-time and embedded (DRE) application software have historically lagged behind those used for mainstream desktop and enterprise software. In addition, programming DRE applications is hard because quality of service (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, custom-programming makes it hard to gracefully support changing application requirements and environments. Too often, developers find themselves reengineering existing software interfaces and implementations in response to planned and unplanned changes. *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 (Buschmann et al., 1996):

- *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

☆ This work was supported in part by BBN, Boeing, DMSO, SAIC, and Siemens.

* Corresponding author. Tel.: +1-949-824-1901.

*E-mail addresses:* schmidt@cs.wustl.edu (D.C. Schmidt), coryan@uci.edu (C. O'Ryan).

*URL:* http://www.ece.uci.edu/schmidt.

0164-1212/03/$ - see front matter © 2002 Elsevier Science Inc. All rights reserved.
doi:10.1016/S0164-1212(02)00078-X

subscribers. Event channels can perform event filtering and routing, QoS enforcement, and fault management.

Publisher/subscriber architectures can be used to address the following challenges:

- *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.

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 (Object Management Group, 2001) and the TAO real-time ORB (Schmidt et al., 1998). TAO is an open source [1] implementation of real-time CORBA that supports efficient, predictable, and flexible DRE applications.

Our previous work (O'Ryan et al., 2002) 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 (Gill et al., 2001), mission-critical distributed audio/video processing (Karr et al., 2001), and large-scale distributed interactive simulation (O'Ryan et al., 2002). TAO and its real-time event service are used in many production DRE applications.

The remainder of this paper is organized as follows: Section 2 describes how publisher/subscriber architectures can be mapped to standard 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 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; and Section 6 presents concluding remarks.

## 2. The CORBA event service

This section describes the CORBA event service and how TAO's real-time event service overcomes limitations with the CORBA event service.

### 2.1. Overview of the CORBA event service

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 (Object Management Group, 1998) 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. Fig. 1 illustrates the relationships between components in the 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,

[1] The source code and documentation for TAO and its real-time event service are freely available at http://deuce.doc.wustl.edu/Download.html.

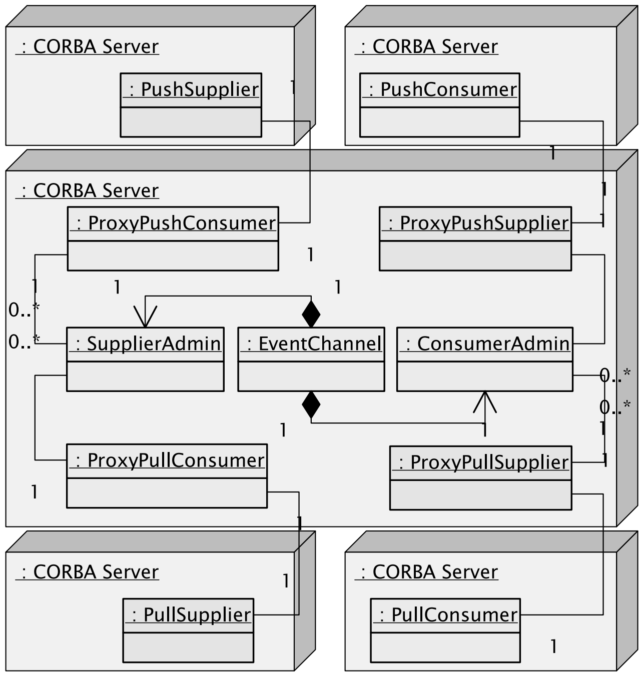

Fig. 1. CORBA event service architecture.

- *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 (Gamma et al., 1995) through which multiple consumers and suppliers communicate asynchronously, i.e., they play the same role as the event channels in the publisher/subscriber architecture.

## 3. The TAO real-time event service

### 3.1. Overcoming CORBA event service Limitations with TAO

Although the CORBA event service described in Section 2.1 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. Fig. 2 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 (Johnson, 1997) that contains a series of processing modules linked together in accordance with the pipes and filters pattern (Buschmann et al., 1996), which provides a structure for systems that process a stream of event data.

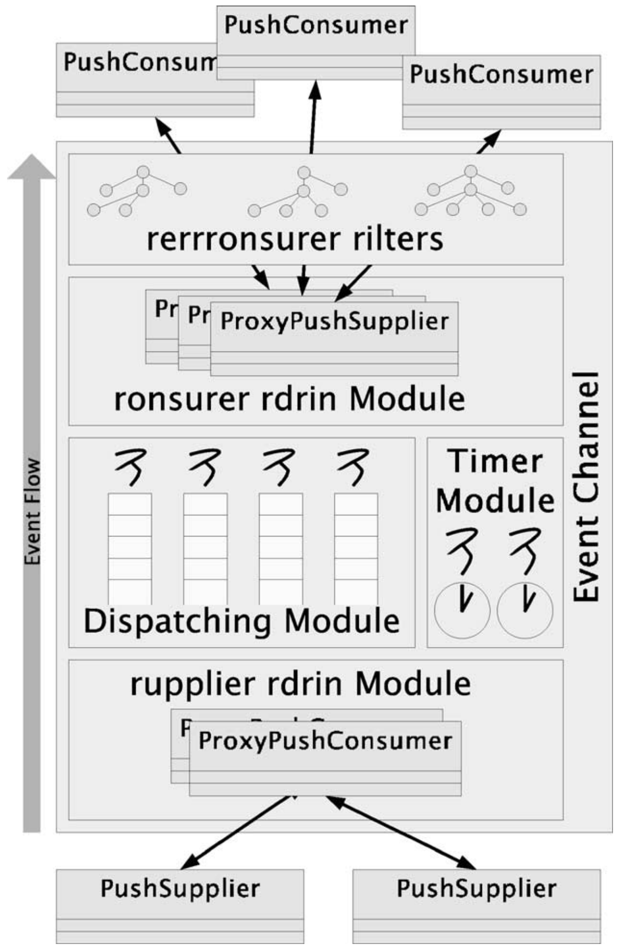

Fig. 2. TAO real-time event service architecture.

### 3.2. Alternative event service configurations

Although TAO's real-time event service architecture 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. Its flexible architecture helps developers of DRE applications meet their requirements

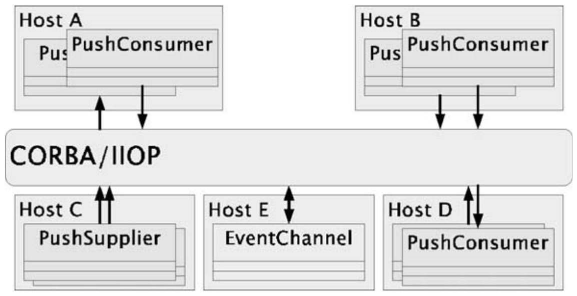

Fig. 3. A centralized configuration for the event channel.

without incurring time and space overhead for capabilities they do not use.

*3.2.1. Centralized vs. federated event channels*

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 Fig. 3. A centralized real-time event channel architecture incurs 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 (Wang et al., 2001), so it is desirable to exploit this optimization whenever possible.

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 Fig. 4. 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.

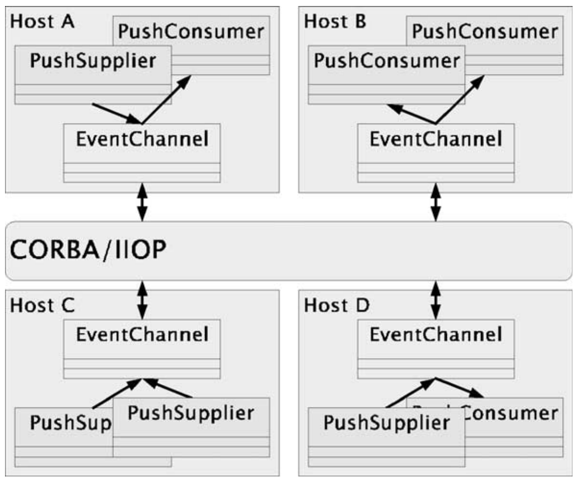

Fig. 4. Federated event channels.

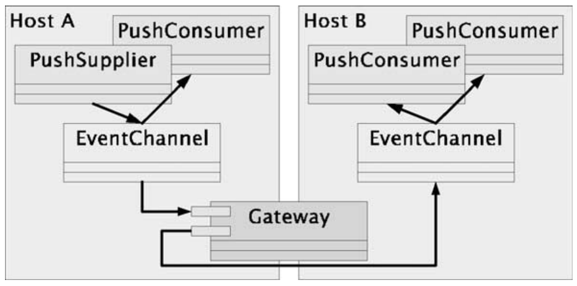

Fig. 5. Using a gateway to connect two event channels.

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 Fig. 5.

*3.2.2. 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 (Buschmann et al., 1996) and builder (Gamma et al., 1995) 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:

*3.2.2.1. Complete event channel configuration*. A complete 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. Fig. 2 illustrates a complete event channel configuration.

*3.2.2.2. 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 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 Fig. 6(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 Fig. 6(B).

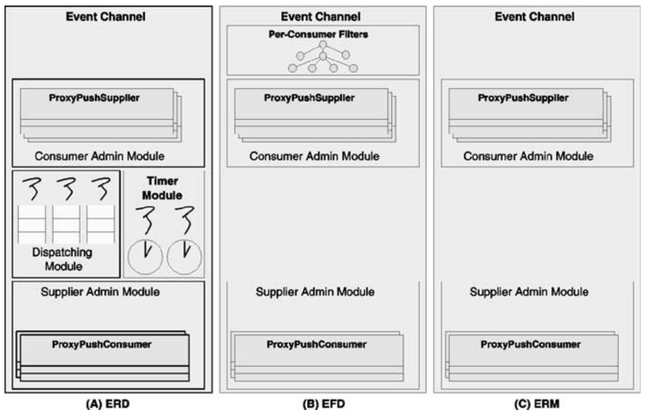

Fig. 6. Event channel subset configurations.

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 Fig. 6(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.

## 4. Designing and optimizing TAOs 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 (Lundberg et al., 2000). To reify this point, this section describes the patterns and idioms we applied to address the design and performance challenges encountered when developing TAO's real-time event service. Section 5 presents empirical results that quantify the benefits of these patterns and optimizations.

### 4.1. Challenge 1: ensuring timeliness in event processing

#### 4.1.1. 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.

#### 4.1.2. 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.

#### 4.1.3. Solution: the command object pattern

This pattern encapsulates an event as an object, thereby allowing parameterization of different events (Gamma et al., 1995). 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."

#### 4.1.4. 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 Fig. 7 instead of performing these operations synchronously,

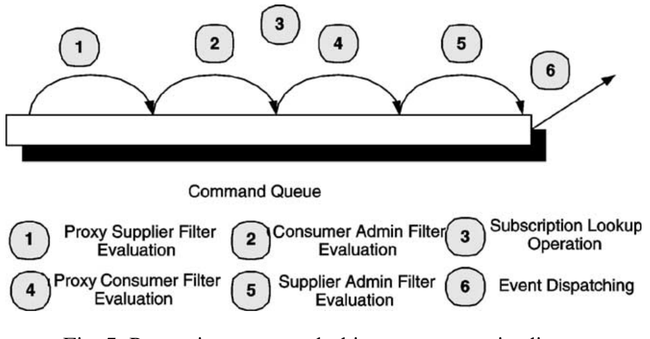
1
2
3
4
5
6
Command Queue
1 Proxy Supplier Filter Evaluation
2 Consumer Admin Filter Evaluation
3 Subscription Lookup Operation
4 Proxy Consumer Filter Evaluation
5 Supplier Admin Filter Evaluation
6 Event Dispatching

Fig. 7. Processing command objects to ensure timeliness.

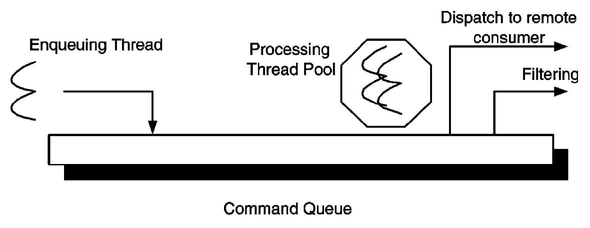

Fig. 8. Asynchronous event processing using the active object pattern.

their 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 Fig. 8.

## 4.2. Challenge 2: optimizations for footprint reduction

### 4.2.1. 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.

### 4.2.2. Problem

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

### 4.2.3. Solution: the pipes and filters pattern and the builder pattern

The pipes and filters pattern (Buschmann et al., 1996) provides a structure for systems that process a stream of data. The builder pattern (Gamma et al., 1995) separates the construction of a complex object from its representation so that the same construction process can create different representations.

### 4.2.4. 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. 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, multithreaded dispatching, or even omitted altogether to create one of the subset event channel configurations described in Section 3.2.2.

## 4.3. Challenge 3: customizing event channels for particular deployment environments

### 4.3.1. 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.

### 4.3.2. Problem

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

### 4.3.3. 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 (Schmidt et al., 2000). 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()`.

### 4.3.4. 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. Fig. 9 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.

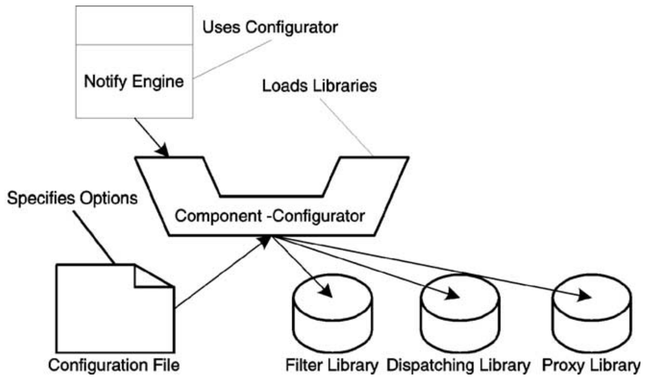

Fig. 9. Apply the component configurator pattern in TAO's real-time event channel.

## 5. Empirical results

Our previous work (Harrison et al., 1997) 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 3.2.1. We then measure several aspects of event channel latency using the ERD configuration described in Section 3.2.2. The benchmarks reported in this section are based on performance requirements gleaned from our extensive work (Schmidt et al., 1998; Gokhale and Schmidt, 1998, 1999; Gill et al., 2001) on real-time avionics mission computing systems (Sharp, 1998; Doerr and Sharp, 1999). Since our focus in this paper is on real-time properties, rather than the scalability properties described in (O'Ryan et al., 2002), 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) (Khanna et al., 1992).

[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 (Sharp, 1998; Doerr and Sharp, 1999).

### 5.1. CPU utilization measurements

#### 5.1.1. Overview

For non-real-time event channels, such as the EFD configuration described in Section 3.2.2, 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 complete 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 (Gopalakrishnan and Parulkar, 1996). 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 Service 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.

#### 5.1.2. 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 Fig. 10 we plot the 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.

Fig. 10 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.

#### 5.1.3. 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

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

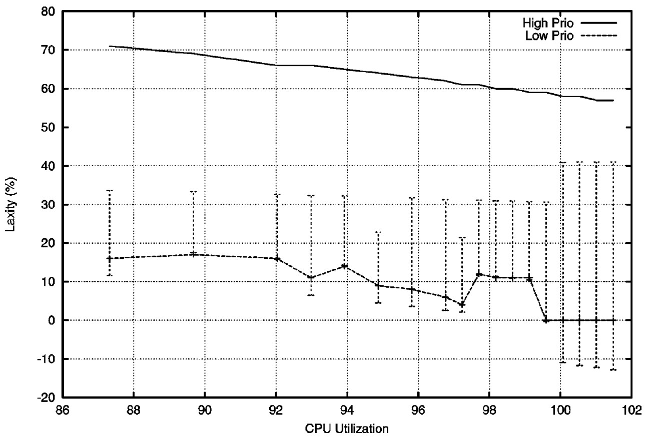

Fig. 10. CPU utilization for a single event channel.

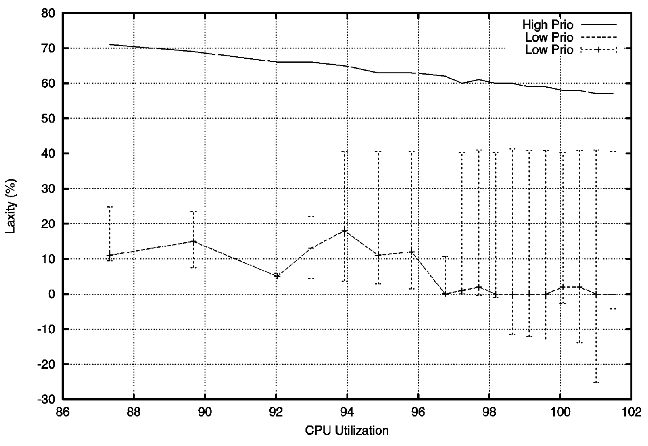

Fig. 11. CPU utilization for a federated event channel.

the processing time for high-priority events was increased until deadlines were missed. Fig. 11 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 (∼3%) loss of utilization yields a substantial performance improvement for the common case, where events are exchanged between local (collocated) suppliers and consumers.

### 5.1.4. 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.
- 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

*5.2.1.1. 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 → consumer latency using the HOP 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.

*5.2.1.2. 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 ∼50 μs. In each case, as the number of suppliers and/or consumers increased, the latency also increased, as shown in Table 1.

*5.2.1.3. 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.

*5.2.1.4. 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

[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.

Table 1
Event latency for collocated event processing

| Supplier | Consumer | Latency (μs) | | | | | |
|---|---|---|---|---|---|---|---|
| | | First consumer | | | Last consumer | | |
| | | 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 | 1210 | 284 |
| 10 | 10 | 167 | 222 | 176 | 369 | 2545 | 576 |
| 10 | 20 | 211 | 310 | 290 | 741 | 4895 | 1137 |

Table 2
Event latency for local and remote event processing

| Supplier | Consumer | Latency (μs) | | | | | |
|---|---|---|---|---|---|---|---|
| | | Local event | | | Remote event | | |
| | | Min | Max | Avg | Min | Max | Avg |
| 1 | 1 | 57 | 71 | 63 | 772 | 1078 | 820 |
| 1 | 5 | 108 | 207 | 155 | 765 | 1995 | 1283 |
| 1 | 10 | 170 | 598 | 289 | 795 | 5853 | 3544 |
| 1 | 20 | 303 | 819 | 534 | 756 | 6047 | 3084 |
| 2 | 1 | 50 | 95 | 57 | 1226 | 2329 | 1297 |
| 2 | 5 | 101 | 214 | 152 | 1274 | 4577 | 2330 |
| 2 | 10 | 167 | 421 | 274 | 1226 | 6676 | 3448 |
| 2 | 20 | 280 | 821 | 519 | 1225 | 20,727 | 6038 |
| 10 | 1 | 49 | 218 | 60 | 1406 | 3969 | 3102 |
| 10 | 5 | 100 | 1170 | 172 | 1477 | 12,773 | 6282 |
| 10 | 10 | 158 | 2379 | 310 | 1258 | 19,153 | 7904 |
| 10 | 20 | 209 | 4900 | 596 | 1266 | 65,449 | 24,345 |

event channel only supports unreliable multicast currently.

- Table 2 illustrates the benefits of the federated event channel configuration described in Section 3.2.1. In particular, disseminating events to consumers collocated in the same process is one to two orders of magnitude more efficient than disseminating them remotely.

### 5.2.2. *Minimal event spacing test*

*5.2.2.1. 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.

*5.2.2.2. Minimal event spacing test results*. Fig. 12 shows that the event channel can deliver over 50 messages per second (i.e., a 50 Hz rate) before it experiences any measurable overhead.

*5.2.2.3. 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 (Sharp, 1998; Doerr and Sharp, 1999)) 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

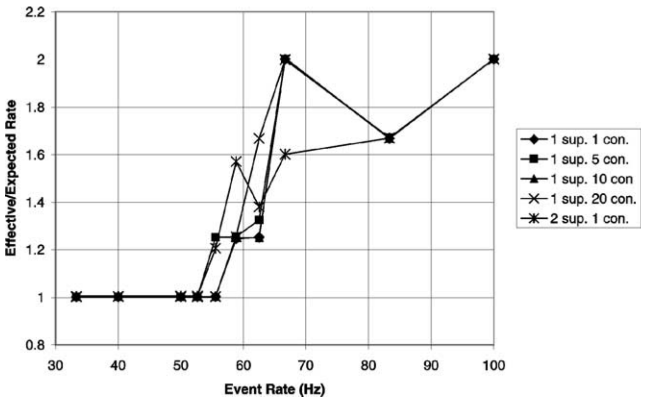

Fig. 12. Minimum event spacing.

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. Concluding remarks

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.

## References

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., 1996. Pattern-oriented software architecture—a system of patterns. Wiley and Sons, New York.

Doerr, B.S., Sharp, D.C., 1999. Freeing product line architectures from execution dependencies. In: Proceedings of the 11th Annual Software Technology Conference.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design patterns: elements of reusable object-oriented software. Addison-Wesley, Reading, Massachusetts.

Gill, C.D., Levine, D.L., Schmidt, D.C., 2001. 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 20 (2).

Gokhale, A., Schmidt, D.C., 1998. Measuring and optimizing CORBA latency and scalability over high-speed networks. Transactions on Computing 47 (4).

Gokhale, A., Schmidt, D.C., 1999. Optimizing a CORBA IIOP protocol engine for minimal footprint multimedia systems. Journal on Selected Areas in Communications special issue on Service Enabling Platforms for Networked Multimedia Systems 17 (9).

Gopalakrishnan, R., Parulkar, G., 1996. Bringing real-time scheduling theory and practice closer for multimedia computing. In: SIGMETRICS Conference, Philadelphia, PA. ACM.

Harrison, T.H., Levine, D.L., Schmidt, D.C., 1997. The design and performance of a real-time CORBA event service. In: Proceedings of OOPSLA'97, Atlanta, GA. ACM, pp. 184–199.

Johnson, R., 1997. Frameworks = patterns + components. Communications of the ACM 40 (10).

Karr, D.A., Rodrigues, C., Krishnamurthy, Y., Pyarali, I., Schmidt, D.C., 2001. 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.

Khanna, S. et al., 1992. Realtime scheduling in SunOS 5.0. In: Proceedings of the USENIX Winter Conference, USENIX Association, pp. 375–390.

Lundberg, L., Hggander, D., Diestelkamp, W., 2000. Conflicts and trade-offs between software performance and maintainability. In: Performance engineering. State of the art and current trends. In: Lecture notes in computer science. Springer-Verlag.

Object Management Group, 1998. CORBAServices: common object services specification, updated edition. Object Management Group, December.

Object Management Group, 2001. The common object request broker: architecture and specification. Object Management Group, 2.6 edition, December.

O'Ryan, C., Schmidt, D.C., Russell Noseworthy, J., 2002. Patterns and performance of a CORBA event service for large-scale distributed interactive simulations. International Journal of Computer Systems Science and Engineering 17 (2).

Schmidt, D.C., Levine, D.L., Mungee, S., 1998. The design and performance of real-time object request brokers. Computer Communications 21 (4), 294–324.

Schmidt, D.C., Stal, M., Rohnert, H., Buschmann, F., 2000. In: Pattern-oriented software architecture: patterns for concurrent and networked objects, vol. 2. Wiley & Sons.

Sharp, D.C., 1998. Reducing avionics software cost through component based product line development. In: Proceedings of the 10th Annual Software Technology Conference.

Wang, N., Schmidt, D.C., Kircher, M., Parameswaran, K., 2001. Towards a reflective middleware framework for QoS-enabled CORBA component model applications. IEEE Distributed Systems Online 2 (5).

**Dr. Schmidt** is an Associate Professor in the Electrical and Computer Engineering department at the University of California, Irvine. He is currently also serving as a program manager the DARPA, where he is leading the national research effort on distributed real-time and embedded (DRE) middleware. His research focuses on patterns, imple-

mentation, and experimental analysis of object-oriented techniques that facilitate the development of high-performance DRE middleware on parallel processing platforms running over high-speed networks and embedded system interconnects. Dr. Schmidt is an internationally recognized and widely cited expert on DRE patterns, middleware frameworks, and real-time CORBA and has published widely in top IEEE, ACM, IFIP, and USENIX technical journals, conferences, and books. Dr. Schmidt received BA and MA degrees in Sociology from the College of William and Mary in Williamsburg, Virginia, and an MS and a PhD in Computer Science from the University of California, Irvine in 1984, 1986, 1990, and 1994, respectively.

**Carlos O'Ryan** completed his PhD at the University of California, Irvine (UCI) Department of Electrical and Computer Engineering. He also works full-time as chief architect and software developer at Automated Trading Desk in Charleston, South Carolina. Previously, he worked as a research assistant for the Distributed Object Computing (DOC) Lab at UCI, focusing the TAO real-time event service, ORB Core, IDL compiler, pluggable protocol framework, Asynchronous Method Invocation, CDR engine, and ORB memory management strategies. Carlos obtained his MS degree in Computer Science from Washington University in St. Louis in 1999 and his BA in Mathematics from Pontificia Universidad Catslica de Chile in 1992.