Patterns and Performance of a CORBA Event Service for Large-scale Distributed Interactive Simulations

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

J. Russell Noseworthy
jrn@objectsciences.com
Object Sciences Corp.
Alexandria, VA, 22312*

This paper appeared in the *International Journal of Computer Systems Science and Engineering*, CRL Publishing, 2001.

Abstract

Advanced distributed interactive simulations have stringent quality of service (QoS) requirements for throughput, latency, and scalability, as well as requirements for a flexible communication infrastructure to reduce software lifecycle costs. The CORBA Event Service provides a flexible model for asynchronous communication among distributed and collocated objects. However, the standard CORBA Event Service specification lacks important features and QoS optimizations required by distributed interactive simulation systems.

This paper makes five contributions to the design, implementation, and performance measurement of distributed interactive simulation systems. First, it describes how the CORBA Event Service can be implemented to support key QoS features. Second, it illustrates how to extend the CORBA Event Service so that it is better suited for distributed interactive simulations, such as the next-generation Run Time Infrastructure (RTI-NG) implementation for the Defense Modeling and Simulation Organization's (DMSO) High Level Architecture (HLA). Third, it describes how to develop efficient event dispatching and scheduling mechanisms that can sustain high throughput. Fourth, it describes how to use multicast protocols to reduce network traffic transparently and to improve system scalability. Finally, it illustrates how an Event Service framework can be strategized to support configurations that facilitate high throughput, predictable bounded latency, or some combination of each.

Keywords: Scalable CORBA event systems, object-oriented communication frameworks, DMSO HLA RTI.

1 Introduction

Overview of distributed interactive simulations. Interactive simulations are useful tools for training personnel to operate equipment or to experience situations that are too expensive, impractical, or dangerous to execute in the real world. The advent of high-speed LANs and WANs has enabled the development of *distributed* interactive simulations, where participants are dispersed geographically. For example, military units stationed around the world can participate in joint training exercises, with human-in-the-loop airplane and tank simulators. Massively multiplayer online gaming is another form of distributed interactive simulation. In both examples, heterogeneous LAN-based computer systems can be interconnected by high-speed WANs, as depicted in Figure 1.

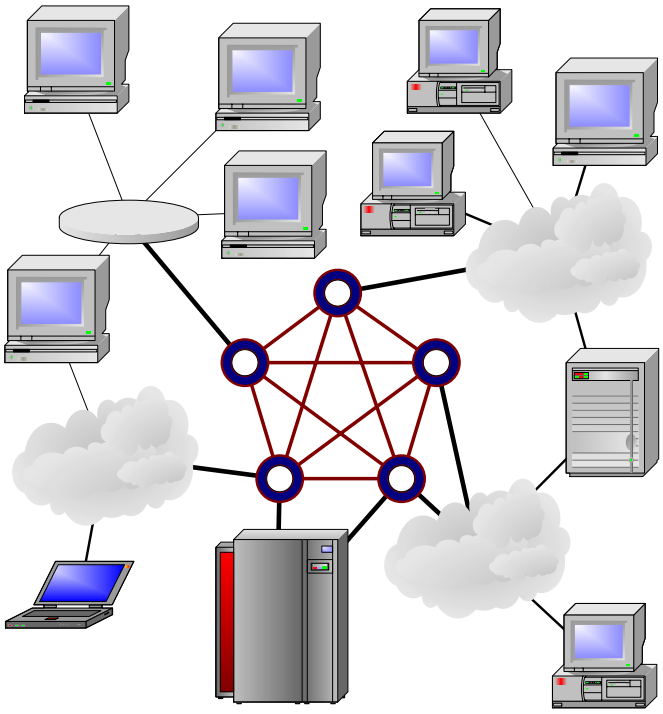

Figure 1: **Distributed Interactive Simulation Architecture**

The quality of service (QoS) requirements on the software that supports distributed interactive simulations can be quite demanding. This software must combine

1. Aspects of distributed real-time computing, such as low-

*This work was supported in part by DMSO, SAIC, and Siemens.

latency and high-throughput, with
2. The need for highly scalable multi-sender/multi-receiver communication over a wide-range of autonomous and interconnected networks.

Meeting these challenges requires an efficient and scalable communication infrastructure, which is the focus of this paper.

Distributed interactive simulation systems, such as DIS [1], have historically been based on Publish/Subscribe patterns [2]. Participants in a simulation declare the data that they supply and consume. Typically, each participant in these *event-driven* systems supplies and consumes only a subset of the possible events in the system. These systems can vary dynamically, however, *e.g.*, publishers and subscribers can join and leave at arbitrary times. Likewise, the set of events published or subscribed to can also vary during the lifetime of a simulation.

It is common for large-scale simulations, such as synthetic theater of war training (STOW) activities, to be composed of hundreds or thousands of publishers and subscribers that generate enormous quantities of events in real-time. Simulation communication infrastructures must therefore scale up to handle large event volumes, while simultaneously conserving network resources by minimizing the number of duplicated events sent to separate subscribers. In addition, the system must avoid wasteful computation. For instance, it should avoid sending events to subscribers who are not interested, as well as quickly rejecting those events if they are received. Moreover, communication infrastructures must be flexible to cope with simulation styles that require different optimization points, such as reduced latency, improved throughput, low network utilization, and reliable or best-effort delivery.

Towards a middleware-based solution. Given sufficient time and effort, it is possible to achieve the specific requirements of distributed interactive simulation applications by developing these systems entirely from scratch. In practice, however, this is unrealistic because:

- The economic context in which these systems are developed places increasingly stringent constraints on time and effort expended on developing software.
- The increasing scarcity of qualified software professionals exacerbates the risk of failing to complete mission-critical projects.

It is rarely realistic to develop complex simulation systems from scratch, therefore, unless the scope of software development required for each project can be constrained substantially.

For these reasons, it is necessary that distributed interactive simulation systems be assembled largely from reusable *middleware* components. Middleware is software that resides between applications and the underlying operating systems, protocol stacks, and hardware in complex distributed systems to enable or simplify how these components are connected [3]. When middleware is commonly available for acquisition or purchase, it becomes commercial-off-the-shelf (COTS).

Employing COTS middleware shields software developers from low-level, tedious, and error-prone details, such as socket-level programming [4]. Moreover, it provides a consistent set of higher level abstractions [5, 6] for developing more flexible and adaptive systems. In addition, it amortizes software lifecycle costs by leveraging previous design and development expertise and reifying key design patterns [7] into reusable frameworks and components.

COTS middleware has achieved substantial success in certain domains, such as avionics mission computing [8] and business applications. There is a belief in some parts of the distributed interactive simulation community, however, that the efficiency, scalability, and predictability of COTS middleware, such as CORBA [9], is not suitable for advanced large-scale simulation applications. Thus, if it can be demonstrated that the overhead of COTS middleware implementations can be removed, the resulting benefits make it a compelling choice as the communication infrastructure for large-scale distributed interactive simulation systems.

Our previous research on middleware has examined many dimensions of high-performance and real-time CORBA ORB endsystem design, including static [10] and dynamic [5] scheduling, event processing [8], I/O subsystem [11] and pluggable protocol [12] integration, synchronous [13] and asynchronous [14] ORB Core architectures, systematic benchmarking of multiple ORBs [15], patterns for ORB extensibility [7], and ORB performance [16]. This paper extends our previous work [8] on real-time extensions to the *CORBA Event Service* [17] as follows:

- We describe the patterns that guided the design and optimization of a flexible and scalable Event Service framework that allows developers to select implementation strategies that are most appropriate for their application domain.
- We show how the CORBA Event Service can be enhanced to support the QoS requirements of large-scale distributed interactive simulations by using UDP/IP multicast to federate multiple event channels and conserve network resources.
- We describe how an implementation of the TAO Event Service has been used in the U.S. Defense Modeling and Simulation Organization (DMSO)'s next-generation High-Level Architecture (HLA) [18] Run-time Infrastructure (RTI-NG) reference implementation to support large-scale distributed interactive simulations.
- We evaluate the performance and scalability of TAO's Real-time Event Service implementation empirically.

The remainder of this paper is organized as follows: Section 2 presents an overview of CORBA and TAO's Real-

time Event Service; Section 3 describes how TAO's Real-time Event Service has been used to implement the standard DMSO HLA RTI; Section 4 describes the patterns and optimizations we applied to TAO's Real-time Event Service to support scalable RTI-based distributed interactive simulation applications; Section 5 shows the results of benchmarks conducted on our implementation under different workloads; Section 6 compares our work with related research; and Section 7 presents concluding remarks.

2 Overview of CORBA and TAO's Real-time Event Service

CORBA is a distributed object computing middleware specification [19] 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 and
- Automating 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 [20].

Figure 2 illustrates the primary components in the OMG CORBA architecture. The ACE ORB (TAO) [10] is our freely

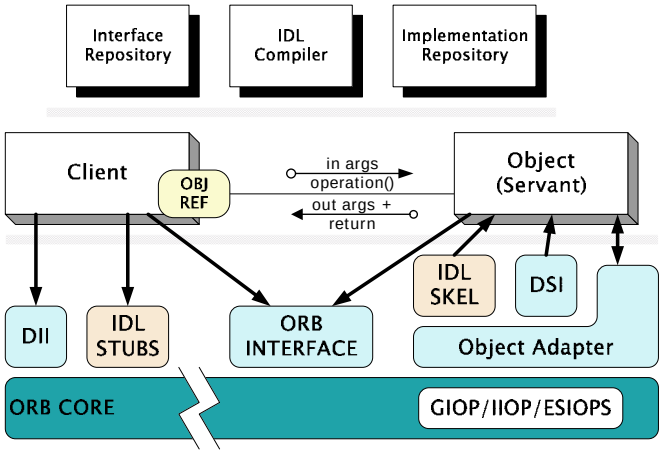

Figure 2: **OMG CORBA Architecture**

available, open-source implementation of CORBA.

Many distributed applications exchange asynchronous requests using *event-based* execution models [21, 22, 23, 24], often called Publisher/Subscriber architectures. To support these common use-cases, the OMG defined a CORBA Event Service component, which is shown in Figure 3.

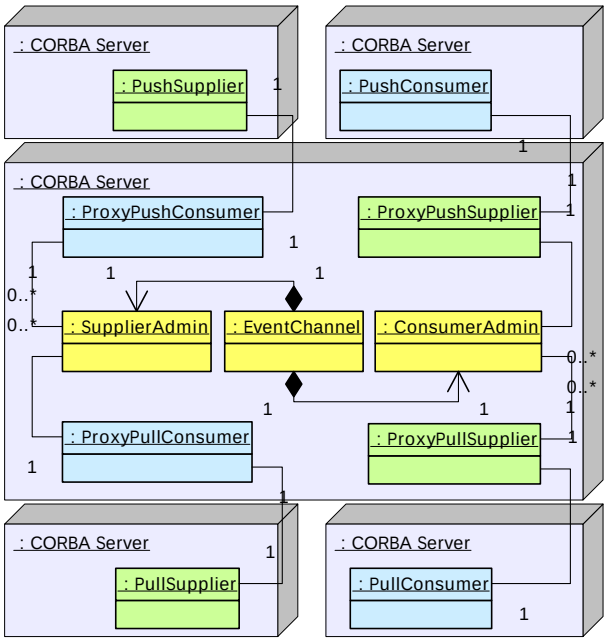

Figure 3: **CORBA Event Service Architecture**

The CORBA Event Service defines three roles:

1. *Suppliers*, which produce event data, *i.e.*they play the publisher role
2. *Consumers*, which receive and process event data, *i.e.*they play the subscriber role, and
3. *Event channels*, which are mediators [25] through which multiple consumers and suppliers communicate asynchronously.

Events are generally transferred via standard CORBA two-way operations from suppliers to an event channel, which in turn forwards the events to consumers. Some Event Service implementations transfer events using one-way operations, but this can cause flow control and reliability problems due to the semantics of CORBA one-way operations [14].

There are four general models of component collaboration in the OMG Event Service architecture. Figure 4 shows the collaborations between suppliers, consumers, and event channels in each of the models outlined below:

A. The canonical push model: In this model, event suppliers initiate the transfer of event data to consumers. As shown in Figure 4(A), suppliers are the active initiators and consumers are the passive targets of the requests. Event channels play the role of *notifier*, as defined by the Observer pattern [25], where observers of an object are notified whenever the object changes its state. Active suppliers therefore use event channels to push data to passive consumers that have registered with event channels.

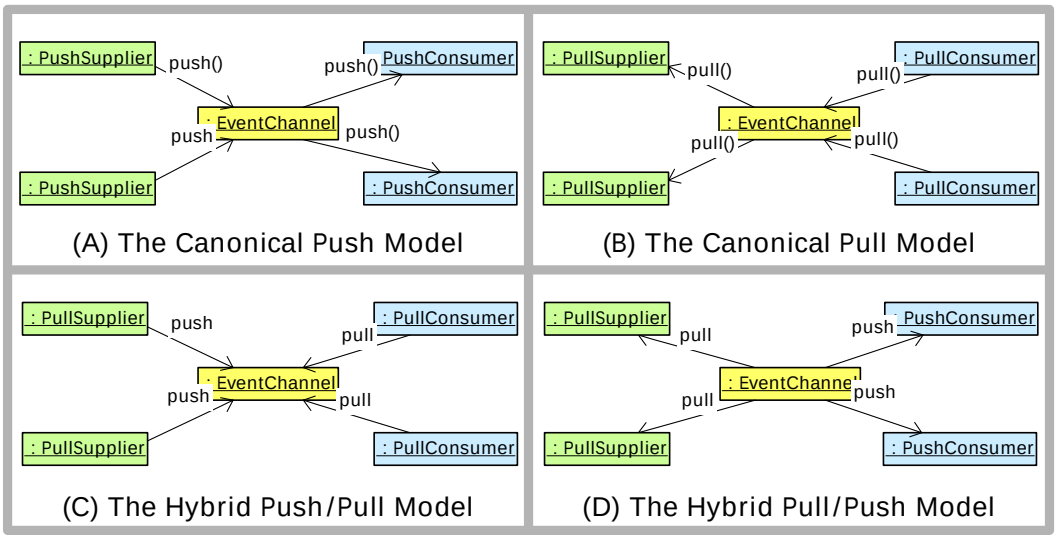

Figure 4: **Delivery Models in the CORBA Event Service**

B. The canonical pull model: In this model, consumers request events from suppliers. As shown in Figure 4(B), consumers are the active initiators and suppliers are the passive targets of the pull requests. Event channels play the role of *procurer* since they obtain events on behalf of consumers. Active consumers therefore can pull data explicitly from passive suppliers via an event channel.

C. The hybrid push/pull model: In this model, consumers request events that are queued at a channel by suppliers. As shown in Figure 4(C), both suppliers and consumers are the active initiators of the requests. Event channels play the role of a *queue*. Active consumers therefore can pull data that is explicitly deposited by active suppliers via an event channel.

D. The hybrid pull/push model: In this model, a channel pulls events from suppliers and pushes them to consumers. As shown in Figure 4(D) suppliers are passive targets of pull requests and consumers are the passive targets of pushes. Event channels play the role of *intelligent agent*. Active event channels therefore can pull data from passive suppliers and push that data to passive consumers.

2.1 Overcoming Limitations with the CORBA Event Service

Although the CORBA Event Service specification provides a standard way to decouple event suppliers and event consumers and support asynchronous communication, it lacks several important features required by large-scale distributed interactive simulations. Chief among these missing features include:

- Centralized event filtering and event correlations[1]
- Efficient and predictable event dispatching and
- Efficient use of network and computational resources.

To resolve these limitations we have developed a *Real-time Event Service* [8] as part of the TAO project [10] at Washington University, St. Louis and the University of California, Irvine. TAO's Real-time Event Service extends the CORBA Event Service specification to satisfy the quality of service (QoS) needs of real-time applications in many domains, such as avionics, telecommunications, process control, and distributed interactive simulations (which is the focus of this paper). The following discussion summarizes the features missing in the CORBA Event Service and outlines how TAO's Real-time Event Service supports them.

Support for centralized event filtering and event correlation. In large-scale distributed interactive simulations, 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, the Event Service should therefore 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 distributed system resources.

It is possible to implement filtering using standard CORBA event channels. For instance, channels can be chained together to create an event filtering graph that consumers use to receive a subset of the total events in the system. However, filter graphs defined using standard CORBA event channels increase the number of hops a message must travel between

[1] Correlation allows an event channel to wait for a *conjunction* of events before sending it to consumer(s).

suppliers and consumers. This increased traversal overhead may be unacceptable for applications with low latency requirements. Likewise, it hampers system scalability because additional processing is required to dispatch each event.

To alleviate these scalability problems, TAO's Real-time Event Service provides filtering and correlation mechanisms that allow consumers to specify logical *OR* and *AND* event dependencies. When the designated conditions are met, the event channel will dispatch all events that satisfy each consumer's dependencies.

Efficient and predictable event dispatching. To improve scalability and take advantage of advanced hardware it may be desirable to have multiple threads within an event channel forwarding events to their consumers. TAO's Real-time Event Service can be configured with an application-specified strategy to assign the number and priority of threads that will dispatch events. The standard distribution also includes several application-specified strategies to cover the most common cases. Since an event can be assigned to a thread with the appropriate OS priority, the same dispatching component can be used to achieve greater predictability and enforce scheduling decisions at run-time.

Efficient use of network and computational resources. Näive implementations of an Event Service will send one message for each remote consumer interested in the event. This design can suboptimally utilize network resources since the same data is transmitted multiple times, often to the same target host. The strategies by which TAO's Real-time Event Service can be configured to minimize network traffic include:

- Using multicast protocols to avoid duplicate network traffic and
- Building federations of Event Services that share filtering information to minimize or eliminate the transmission of unwanted events to a remote entity.

2.2 TAO's Real-time Event Service Architecture

TAO's Real-time Event Service is implemented using the Mediator pattern [25]. The heart of the Real-time Event Service service is the event channel shown in Figure 5. The features of TAO's event channel are defined by an `Event_Channel` IDL interface and implemented by a C++ class of the same name. This class also plays a mediator role by serving as a "location broker" so the rest of an event channel's components can find each other as they are being composed at initialization-time.

When a `ProxyPushConsumer` receives an event from an application the following activities occur:

1. It iterates over the set of `ProxyPushSupplier`s that represent the potential consumers interested in that event. Section 4.2 describes how this set is determined.

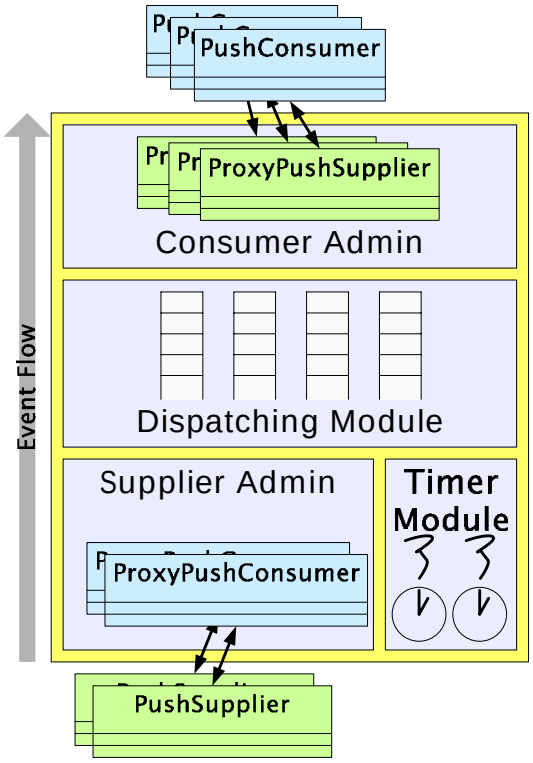

Figure 5: **TAO's Real-time Event Service Architecture**

2. Each `ProxyPushSupplier` then checks to see if the event is relevant for its consumer. This check is performed by the *filter hierarchy* described in Section 4.1.
3. If a consumer is interested in the event, a *dispatching strategy* selects the thread that will dispatch the event to the consumer. Section 4.6 discusses various tradeoffs to consider when selecting the dispatching thread strategy.
4. For real-time applications that require periodic event processing, the Event Service can contain an optional `Timer Module`. Section 4.14 outlines several strategies for generating timer events. Each strategy possesses different predictability and performance characteristics and different resource requirements.

3 Applying TAO's Real-time Event Service to DMSO's HLA RTI-NG

The *High Level Architecture* (HLA) [18] is a standard for distributed simulation middleware promulgated by the the U.S. Defense Modeling and Simulation Office (DMSO). Every implementation of this standard is referred to as a *Run-time Infrastructure* (RTI). Early RTI's demonstrated the viability of the HLA specification, so DMSO commissioned the development of a *next-generation* RTI (RTI-NG).

The architecture of the HLA RTI-NG is centered around TAO's Real-time Event Service. Fundamentally, HLA specifies publish/subscribe distributed middleware, so TAO is an excellent foundation. Some elements of the HLA are not well-suited to the publish/subscribe model, such as elements involving the control of event data dissemination. These elements use normal CORBA remote operation invocations, as provided by TAO's real-time implementation of CORBA. This discussion of the RTI-NG will only consider the publish/subscribe elements, however.

In HLA parlance, a group of participants cooperating in a distributed simulation is a *federation*.[2] A *federate* is the term describing a participant of a federation. A given process may contain one or more federates, each one belongs to a federation. Although every federate belongs to only a single federation, the federates in a multi-federate process need not all belong to the same federation, as shown in Figure 6. Moreover,

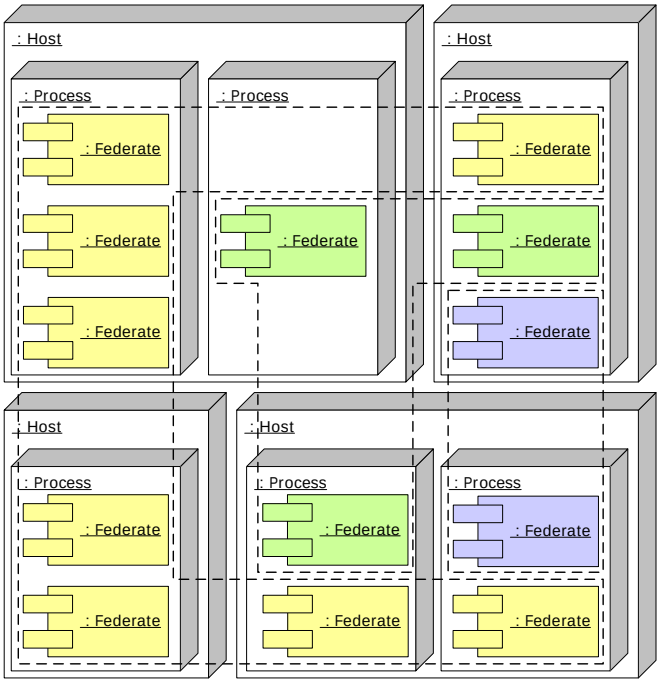

Figure 6: **Examples of Configurations for Federations, Federates, and Processes**

a single process may have more than one federate, possibly joined to different federations. Federates not in the same federation cannot communicate using the HLA standard.[3]

Since there is no need to support communication between federates in different federations, every RTI-NG process contains a separate instance of the event data dissemination mechanisms for each of the federations with which the process communicates. This event data dissemination mechanism uses three TAO real-time event channels, as well as several gateways described in Section 4.9 and a single multicast gateway described in Section 4.11.

The HLA specifies the following four combinations of event transport and ordering:

1. Reliable/receive-ordered
2. Reliable/time-stamp-ordered
3. Best-effort/receive-ordered and
4. Best-effort/time-stamp-ordered.

The multiple event channels and gateways support these four combinations of transport and ordering efficiently. Two of the event channels and the gateways support the reliable transport (both receive- and time-stamp-ordered). The third event channel and the multicast gateway support best-effort transport.

For the reliable transport, one event channel is used to handle outbound (reliable) events. The other event channel and the gateways are used to handle inbound events. As discussed in Section 4.10, event channel subscriptions and publications can change frequently. Using separate event channels for inbound and outbound communication allows publications (subscriptions) to change concurrently with the receipt (transmission) of events.

Figure 7 illustrates a configuration with two processes that contain federates in the same federation, process A and process B. The outbound event channel in A is connected to a gateway in process B that supplies events to process B's inbound event channel. The RTI can select between reliable and unreliable communication by selecting either the Internet Inter-ORB Protocol (IIOP)-based Event Service or by using a multicast-based Event Service.

To support the reliable and time-stamp-ordered event delivery required by the HLA, the RTI-NG uses a distributed snapshot algorithm that accounts for every *reliable event* that is sent and received [26]. To accomplish this, the RTI-NG extended TAO's `ProxyPushSupplier` to count every reliable event that is transmitted to a remote gateway. Likewise, TAO's event channel gateway was extended to count every reliable event that is received. This information is then provided to the distributed snapshot algorithm.

For simplicity, best-effort events are transmitted via UDP/IP multicast directly, *i.e.*, without the use of the TAO's event channel. However, best-effort events are received using TAO's mulicast event channel gateway. Each multicast gateway passes the events it receives to an inbound multicast event channel. Once again, this design minimizes the impact of publication and subscription changes on best-effort and reliable events.

[2] Strictly speaking, *federation execution* is the name given to the group of participants while they are actually running, to draw distinction between a running simulation and a simulation that is not running (*e.g.*, being planned).

[3] It is possible to bridge communication between HLA federates, although such bridging is not standardized by the HLA specification.

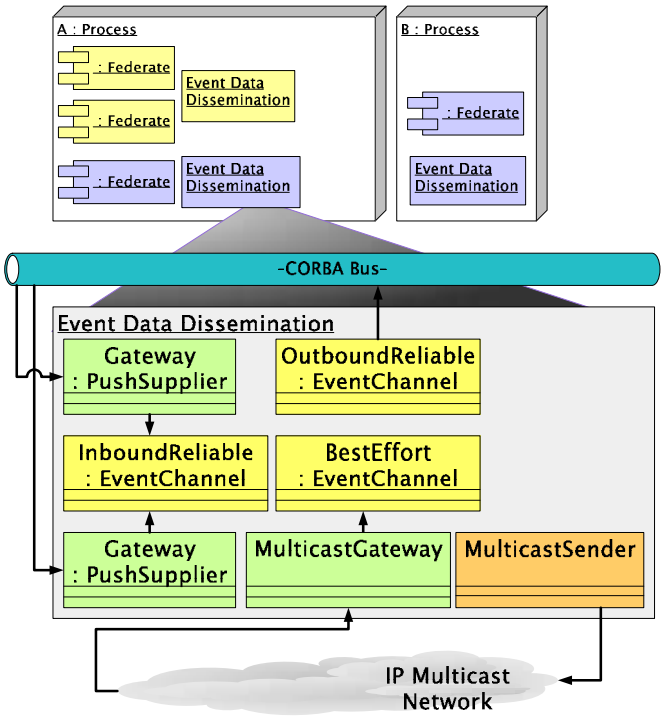

Figure 7: **Use of TAO's Event Service Gateways in the RTI**

In addition to four combinations of event transport and ordering, the HLA specifies that programmers developing a federation have the ability to logically *segment* the events exchanged amongst federates in order to minimize unwanted network traffic. Ideally, the implementation of programmer-defined segmentation would be perfect and each subscriber would receive only the events it wants. In practice, however, the implementation of the user-defined segmentation is rarely perfect, so unwanted events must be filtered by the receiver. Note that each supplier supplies only one type of event.

The RTI-NG has various static mappings between these programmer-defined segmentations and TAO suppliers. We describe the various mappings on the sender- and receiver-side below:

- On the sender, the RTI-NG maps a given programmer-defined segmentation onto a set of TAO suppliers. In the case of reliable transport, these suppliers push events onto the reliable outbound event channel. The outbound reliable event channel uses the consumer filtering scalability improvements described in Section 4.2. In the case of best-effort transport, there is only one specialized supplier. This supplier collaborates with an address server described in Section 4.11 to transmit events to the multicast group dictated by the statically determined mapping.
- On the receiver, the RTI-NG instantiates one consumer for every federate in the process. The inbound event channels (both reliable and multicast) deliver their events to these consumers. The inbound event channels therefore serve to dispatch an event to a potentially large number of co-located consumers.

Events exchanged by the RTI-NG do not carry CORBA Any's, as specified in the OMG Event Service Specification. Instead, they carry CORBA octet sequences. This is an important optimization that eliminates the overhead incurred when sending and receiving CORBA Any's.

4 Applying Patterns to Resolve Common Design Challenges

Section 2.2 outlined the core components of the CORBA Event Service that are defined by IDL interfaces. This section describes the following design challenges that we identified prior to and during the development of TAO's Real-time Event Service:

1. Implementing an Extensible and Efficient Filtering Framework
2. Improving Consumer Filtering Scalability
3. Reducing Memory Footprint
4. Supporting Re-entrant Calls while Dispatching Event
5. Reducing Synchronization Overhead
6. Selecting the Thread to Dispatch an Event
7. Configuring Event Channel Strategies Consistently
8. Supporting Rapid Testing and Run-time Changes in the Configuration
9. Exploiting Locality in Supplier/Consumer Pairs
10. Updating the Gateway Subscriptions and Publications
11. Further Improving Network Utilization
12. Exploiting Hardware- and Kernel-level Filtering
13. Breaking Event Cycles in Event Channel Federations
14. Providing Predictable and Efficient Periodic Events

These challenges and the solutions we applied to address them are discussed below. We focus on our systematic application of key patterns, such as Builder, Command, Composite, and Strategy from the GoF book [25] and Strategized Locking and Component Configurator from the POSA2 book [27], to tackle the design and implementation challenges posed when customizing TAO's Real-time Event Service for the HLA RTI-NG described in Section 3. Since these patterns are applicable to many related distributed real-time systems, we document these patterns were applied and composed in TAO to achieve our performance and scalability goals.

4.1 Implementing an Extensible and Efficient Filtering Framework

Context. TAO's real-time Event Service provides several filtering primitives, *e.g.*, a consumer can only accept events of a given type or from some particular source [8]. Not all applications require all filtering mechanisms provided by the Real-time Event Service, however. For example, many distributed interactive simulations do not require correlation. Likewise, some Event Service applications do not require filtering. Moreover, consumers often compose several filtering criteria into their subscription list, *e.g.*, they request to receive *any* event from a given list or a single notification when *all* the events in a list are received.

Problem. An event channel should support the addition of new filtering primitives flexibly and efficiently. Examples of such primitives include

- Receiving a single notification when a set of events are received in a particular order or
- Accepting any event whose type matches a designated bitmask.

Solution → Use the Composite pattern [25]. This pattern allows clients to treat individual objects and compositions of objects uniformly. Usually, the composition forms a tree structure, which in our case is called a *filter composition tree*. New filtering primitives can be implemented as leaves in the composition tree. These primitives provide applications with substantial expressive power, *e.g.*, they can can create complex filter hierarchies using disjunction and conjunction composites, as shown in Figure 8.

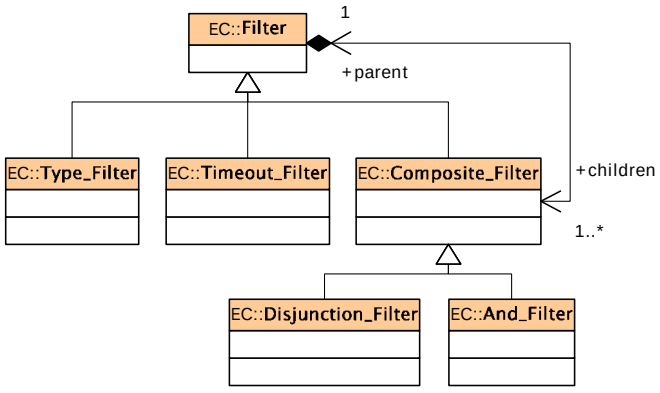

Figure 8: **Composite Event Service Filters Used to Build Composition Trees**

To control the creation of the concrete filters, we use the Builder Pattern [25], which separates the construction of a complex object from its representation. Currently applications describe the filter hierarchy using a sequence of IDL structures. However, the patterns described above can use a description based on a filtering language, such as the CORBA Extended Trader Constraint Language [28]. One important consequence of using the Builder pattern is that changing from one description format to another does not affect the overall architecture of TAO's Event Service framework.

4.2 Improving Consumer Filtering Scalability

Context. In some distributed interactive simulation applications, only a small percentage of consumers are interested in a particular event.

Problem. As the number of consumers grows, an event channel implementation that queries the complete list of consumers to check if they are interested in a particular event will scale poorly, since the time required to dispatch an event will increase linearly with the number of consumers tested.

Solution → Pre-compute the set of consumers for each supplier. Because each supplier attaches to the Event Channel via a unique `ProxyPushConsumer` each one of this objects can keep a separate list of consumers, that only includes consumers interested in events generated by the supplier. The set can be computed *a-priori* using (1) the list of event types generated by the supplier and (2) the pre-computer filter attached to each consumer. In TAO's Event Service these sets are computed by the `Supplier_Filter` class shown in Figure 9.

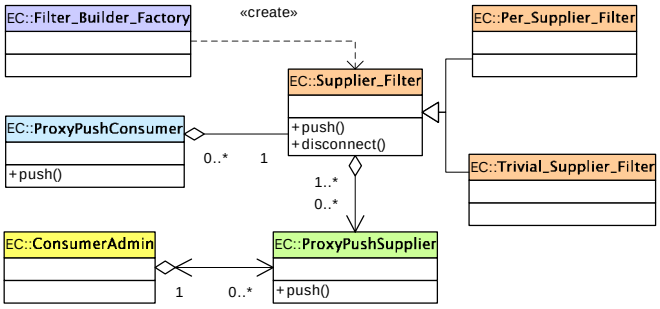

Figure 9: **TAO's Real-time Event Service Supplier Filters**

4.3 Reducing Memory Footprint

Context. In other distributed interactive simulation applications, a large percentage of consumers may be interested in the events generated by each supplier. In such cases, it is counterproductive to use the pre-computation optimization described in Section 4.2. Instead, it may be more efficient to use a single global consumer set, which reduces the memory footprint and minimizes the time required to update the consumer set.

Problem. Using a per-supplier consumer set can result in excessive memory allocations. However, the per-supplier consumer set offers the best performance for applications with medium range consumer set sizes.

Solution → Use the Strategy Pattern [25]. In this pattern, a family of algorithms is represented by classes that share a common base class. Clients access these algorithms via the base class, which enables them to select different algorithms without requiring changes to themselves. We use this pattern in TAO's Real-time Event Service framework to encapsulate the exact algorithm that controls the number of consumer sets, as well as how these sets are updated. Note that the variations mentioned thus far are not exhaustive, *e.g.*, we can store a separate consumer set for each event *type*. The framework implemented in TAO's Real-time Event Service supports this use case, as well.

4.4 Supporting Re-entrant Calls while Dispatching Events

Context. To dispatch an event to multiple consumers, an event channel must iterate over its set of `ProxyPushSupplier` objects. Some distributed interactive simulation cannot use multi-threaded configurations because the RTI is used as part or calls legacy code. In such cases reactive dispatching strategies described in Section 4.6 must be used. Therefore, the same thread that iterates over a consumer set executes the *upcall*,[4] as shown in Figure 10. Consumers can then push new events, add or remove consumers and suppliers, and call back into the event channel and its internal components.

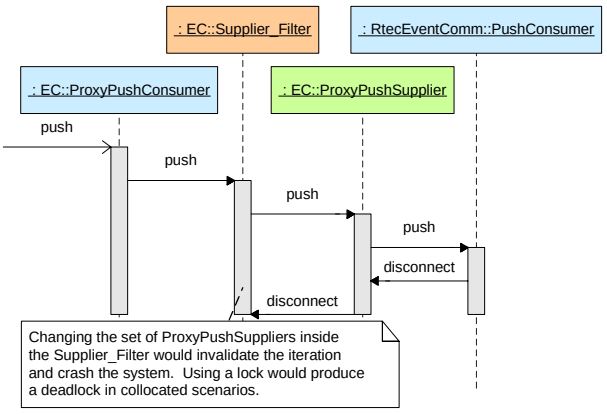

Figure 10: **An Example of a Reentrant Call During the Dispatch Sequence**

[4] An *upcall* is the invocation of an application-provided function by the middleware.

Problem. An event channel should support re-entrant calls during event dispatching, regardless of which concurrency model is being used. However, many iterator implementations become invalidated when their data structure is modified [29]. The `ProxyPushSupplier` set therefore cannot be changed when a thread is iterating over it. Simply locking the set is inappropriate because the application will either deadlock if the upcall changes the set or will invalidate iterators if recursive locks are used. Another inappropriate alternative is to copy the `ProxyPushSupplier` set *before* starting the iteration. Although this strategy works for small sets, it performs poorly for large-scale distributed interactive simulation applications.

Solution → Apply lazy evaluation to delay certain operations. TAO's event channel tracks the number of threads iterating over each set of `ProxyPushSupplier` objects. Before performing changes that would invalidate other iterators, it checks to ensure no concurrent iterations are occurring. If iterations are in progress, the operation is stored as a command object [25]. When no threads are iterating on the set, all delayed command operations can be executed sequentially.

To avoid starving a delayed operation indefinitely, limits can be placed on the number of iterations started after a pending modification occurs. After this limit is reached, all new threads must wait on a lock until the modification completes.

Although the lazy evaluation solution described above is functionally correct, it increases synchronization overhead along the critical path of the event filtering and dispatching algorithms. TAO's real-time event channels can therefore be configured to decouple (1) threads that iterate over the `ProxyPushSupplier` sets from (2) threads that perform consumer upcalls. These configurations do not suffer from the concurrency problems described earlier. Moreover, this design has other benefits since it:

- Yields more predictable behavior in hard real-time systems
- Allows event channels to re-order the events, *e.g.*, in order to perform dynamic scheduling [5] and
- Isolates event suppliers from the execution time of consumer upcalls.

Although this design may increase context switch overhead, many applications can tolerate overhead if event channels already use separate threads to perform upcalls.

4.5 Reducing Synchronization Overhead

Context. Excessive synchronization overhead can be a significant bottleneck when it occurs in the critical path of a concurrent system.

Problem. The synchronization protocols described in Section 4.4 incur more synchronization overhead than a simple critical section. This will produce a reduction in performance

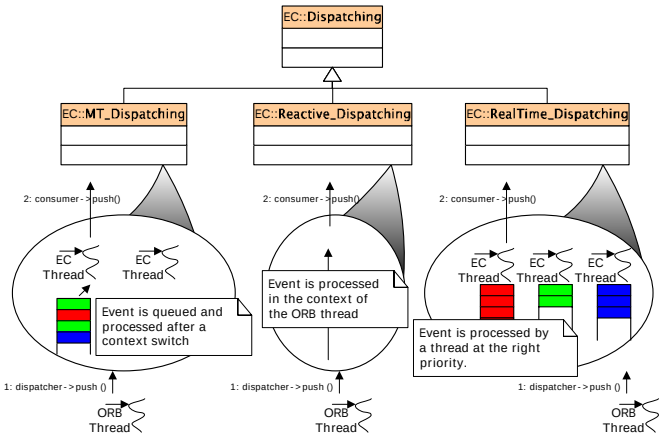

Figure 11: **Dispatching Strategies Supported in TAO's Real-time Event Channel**

for Event Service configurations that can work with a simpler synchronization strategy.

Solution → Use the Strategy Pattern [25]. TAO's event channel uses this pattern to strategize the dispatching algorithm and minimize overhead in applications that do not require complex concurrency and re-entrancy support. For complex use-cases, TAO's event channel uses a special lock that updates the state in the set to indicate that a thread is performing an iteration. When this lock is released, any operations delayed while the lock was held are executed.

4.6 Selecting the Thread to Dispatch an Event

Context. After an event channel has determined that a particular event should be dispatched to a consumer, it must decide which thread will perform the dispatching. As shown in Figure 11, there are several alternatives. Using the same thread that received the event is efficient, *e.g.*, it reduces context switch, synchronization, and data copying overhead [16]. However, this design may expose an event channel to misbehaving consumers. Moreover, to avoid priority inversions in real-time systems, events must be dispatched by a thread at the appropriate priority. Likewise, highly-scalable systems may want to use a pool of threads to dispatch events, thereby leveraging advanced hardware and overlapping I/O and computation.

Problem. No single dispatching algorithm is appropriate across all applications. For example, single-threaded applications can only use reactive dispatching. However, this will result in poor scalability on multi-processor systems. Likewise, a real-time system would require multiple dispatching queues or priority based queues to avoid unbounded priority inversions.

Solution → Use the Strategy Pattern [25]. This pattern can be applied to encapsulate the algorithm used to choose the dispatching thread. The selected dispatching strategy is responsible for performing any data copies that may be necessary to pass the event to a separate thread. The current implementation of TAO's event channel exploits several optimizations, such as reference counting, in the TAO ORB to reduce those data copies. In applications with stringent real-time requirements, the dispatching strategy collaborates with TAO's Scheduling Service [10, 5] to determine the appropriate queue and thread to process the event. When the same thread is used for reception *and* dispatching, the strategy collaborates with the `ProxyPushSupplier` to minimize locking overhead, as described in Section 4.5.

4.7 Configuring Event Channel Strategies Consistently

Context. As discussed in the previous sections, TAO's real-time event channel provides many strategies that can be configured flexibly by application developers. Often, the choice of one strategy affects other strategies. For example, if an event channel's dispatching strategy always uses a separate thread to process the event there is no risk of having re-entrant calls from the consumers modifying the `ProxyPushSupplier` sets. A simpler strategy to manipulate those sets can therefore be used.

Problem. Due to TAO's Real-time Event Service configurability, selecting a suitable combination of strategies can impose an undue burden on the developer and yield inefficient or semantically incompatible strategy configurations. Ideally, developers should be able to choose from a set of configurations whose strategies have been pre-approved to achieve certain goals, such as minimizing latency, avoiding priority inversion, or improving system scalability.

Solution → Use the Abstract Factory Pattern [25] to control the creation of all the objects in an event channel. In this pattern, a single interface creates families of related dependent objects. We use an abstract factory to provide a single point to select all event channel strategies and to avoid semantically incompatible configurations. Concrete implementations of this abstract factory ensure that strategies and components are compatible semantically and collaborate correctly.

4.8 Supporting Rapid Testing and Run-time Changes in the Configuration

Context. Some applications may be used in multiple environments, with different event channel strategies configured for each environment. During application development and

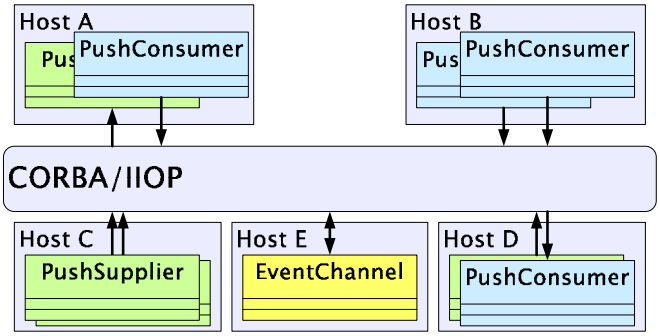

Figure 12: **A Centralized Configuration of the TAO Real-time Event Service**

testing, it may be necessary to evaluate multiple configurations to ensure that the application works in all of them or to identify the most efficient/scalable configurations.

Problem. If an event channel is configured statically, it is hard to evaluate various combinations without time consuming recompiling/relinking.

Solution → Use the Component Configurator Pattern [27]. This pattern allows applications to dynamically and/or statically configure service implementations into an application process. We use this pattern to load abstract factories dynamically and use them to create various event channel configurations. Our implementation includes a default abstract factory that employs the scripting features of the ACE Service Configurator framework [30], which is a platform-independent C++ implementation of the Component Configurator pattern. By using this default, developers or end-users can modify event channel configurations during their initialization by simply changing entries in a configuration file.

4.9 Exploiting Locality in Supplier/Consumer Pairs

Context. TAO's event channels can be accessed transparently across distribution boundaries since they are based on CORBA. Many applications want to be shielded from distribution aspects, while simultaneously achieving high performance.

Problem. There are use-cases where distribution transparency may not yield the most *effective* configuration. For example, Figure 12 illustrates a scenario where most or all consumers for common events reside in the same process, host, or network with the supplier. Thus, sending an event to a remote event channel—only to have it sent back to the same process immediately—wastes network resources and increases latency unnecessarily. Likewise, there may be multiple remote consumers expecting the same event. Ideally, bandwidth should

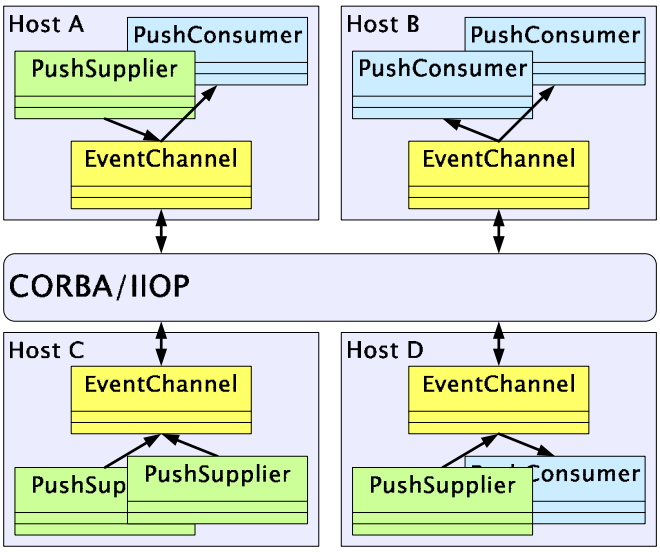

Figure 13: **A Federated Event Channel Configuration**

be conserved in this case by sending a single message across the network to all those remote consumers.

Solution → Federate Event Channels. Figure 13 illustrates a federated group of event channels. Suppliers and consumers connect only to their local event channel, while event channel instances talk to each other via the CORBA bus. This design reduces average latency for all the consumers in the system because consumers and suppliers exhibit locality-of-reference, *i.e.*, most consumers for any event are in the same domain 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 14, 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 the events that are interesting for the participants in the second event channel. When a gateway receives an event it forward the event to the second event channel, where it has connected to using its supplier role.

The application developer 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. However, collocating the gateway with its source event channel can avoid event cycles (see Section 4.13) more efficiently than the previous configuration.

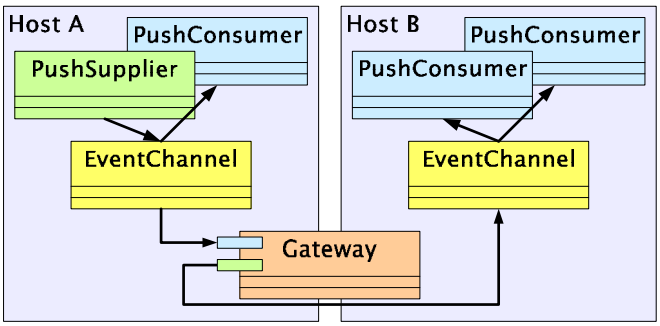

Figure 14: **Using a Gateway to Connect two Event Channels**

4.10 Updating the Gateway Subscriptions and Publications

Context. In a dynamic environment, subscriptions can change continually.

Problem. To use network resources efficiently, the event channel gateways described in Section 4.9 must avoid subscribing to all events generated by each remote event channel.

Solution → Use the Observer Pattern [25]. In this pattern, all dependents (observers) of an object are notified whenever the object changes its state. When this pattern is applied to TAO's Real-time Event Service framework, the changes in the subscription and publication lists are propagated to all interested consumers and suppliers. Gateways can use this information to receive only the events that will be of interest to consumers in the sink event channel.

In some applications the subscriptions do not change dynamically, or the application may not register any observers to an event channel. In this case, the overhead, however small, required to propagate changes in the subscriptions and publications list should be eliminated completely. TAO's Real-time Event Service uses the strategy pattern to achieve this goal.

4.11 Further Improving Network Utilization

Context. In distributed interactive simulations, it is common that an event will be dispatched to multiple hosts in the same network.

Problem. Network bandwidth is often a scarce resource for large-scale simulations, particularly when they are run over a wide-area network (WAN). As the number of nodes increase, therefore, sending the same event multiple times across a network may not scale.

Solution → Use a multicast or broadcast protocol: TAO's event channel can be configured to use UDP to multicast events. As with the gateways described in Section 4.10, a special consumer can subscribe to all the events generated by local suppliers, as shown in Figure 15. This consumer uses

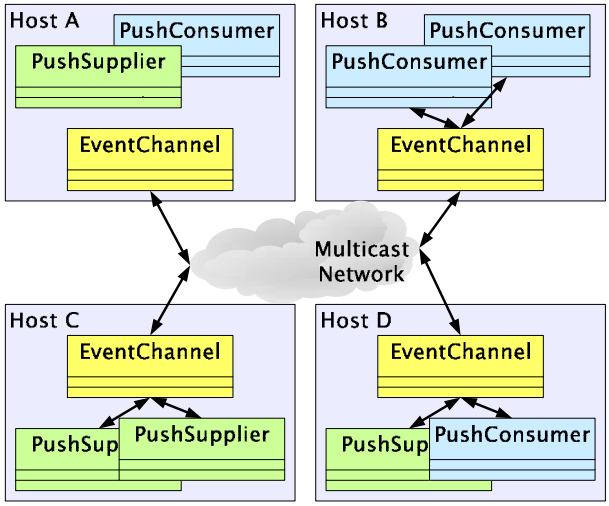

Figure 15: **Using Multicast in Federated Event Channels**

multicast to send events to selected channels in the network. On each receiver, a designated supplier re-publishes all events that are of interest for local consumers. This supplier receives remote multicast traffic, converts it into an event, and forwards the event to its local consumers via an event channel. For both consumers and suppliers, the observer interface described in Section 4.10 is used to modify the subscriptions and publications of multicast gateways dynamically.

4.12 Exploiting Hardware- and Kernel-level Filtering

Context. If different types of events can be partitioned onto different multicast groups, consumer hosts only receive a subset of the multicast traffic. In large-scale distributed interactive simulations it may be necessary to disseminate events over several multicast groups. This design avoids unnecessary interrupts and processing by network interfaces and OS kernels when receiving multicast packets containing unwanted information.

Problem. An event channel must select the multicast group used for each type of event in a globally consistent way. However, the mapping between events and multicast groups may be different for each application. Applications can use different mechanisms to achieve that goal. For instance, some use pre-established mappings between their event types and the multicast groups, whereas others use a centralized service to maintain this mapping. Moreover, applications that require highly scalable fault tolerance may choose to distribute the mapping service across a network. An event channel must be

able to satisfy all these scenarios, without imposing an inefficient one-size-fits-all implementation strategy.

Solution → Use a user-supplier callback object. Application developers can implement an *address server*, which is a CORBA object that event channel gateways query to re-direct events to the appropriate multicast group. Gateways on the receiver-side consult this service to decide which multicast groups they need to subscribe to, based upon the current set of event-type subscriptions in their sink event channel. Advanced operating systems and network adapters can use this information to process only the multicast traffic that is relevant to them.

To avoid single points of failure and to improve scalability, application developers can replicate address servers across a network. If developers use a static mapping between events and multicast groups, there is no need to communicate state between address services. Conversely, if mappings change dynamically, applications must implement mechanisms to propagate these changes to all address servers. One solution is to use an Event Service itself to propagate this information.

4.13 Breaking Event Cycles in Event Channel Federations

Context. In a complex distributed interactive simulation, the same event could be important for both local and remote consumers. For instance, a local supplier can generate *vehicle position events*. If both a local and remote consumer are interested in these events, the gateways could continuously send the event between two federated event channels.

Problem. Consumers for a particular event can be present in multiple channels in the federation. In this case, gateways will propagate events between the peers of the federation indefinitely due to *cycles* in the event flow graph. One approach would be to add addressing information to each event and enhance the routing logic in each event channel. This design would complicate the gateway architecture for simpler use-cases, however, and would require additional communication among the peers.

Solution → Use a *time-to-live* (TTL) counter. This counter is stored in each event and decremented each time an event passes through a gateway. If the TTL counter becomes zero the event is deallocated and not forwarded. Usually event channel federations are fully connected, *i.e.*, all event channels have a gateway to each of their peers. Setting the TTL counter to 1 therefore eliminates all cycles because no event traverses more than one gateway link. In more complex distributed configurations, however, the TTL can be set to a higher number, though events may loop before being discarded. To further improve performance, the TAO event channel has been optimized to reduce data copying, *e.g.*, only the event header requires a copy to change the TTL counter, the payload, that usually contains most of the data, is not touched.

4.14 Providing Predictable and Efficient Periodic Events

Context. Real-time applications require an event channel to generate events at particular times in the future. For instance, applications can use these events to detect missed dead-lines in non-critical processing or to support hardware that requires watchdog timers to identify faulty equipment. In addition, some applications require periodic events to initiate periodic tasks and to detect that periodic tasks complete before their deadline.

Applications with hard real-time requirements may assign different priorities to their timer events. To avoid priority inversions, therefore, events should be generated and dispatched by threads at the appropriate priorities. Soft real-time applications or best-effort applications often impose no such strict requirements on timer priorities and can therefore be better served by simpler strategies that conserve memory and CPU resources. Other applications require no timers at all and obviously single-threaded applications cannot use this technique to generate periodic events.

Problem. Implement predictable periodic events for hard real-time applications without undue overhead for applications with less stringent predictability requirements.

Solution → Use the Strategy Pattern [25] to select the mechanisms used to generate timeout events dynamically. In TAO's event channel, the `ConsumerFilterBuilder` creates special filter objects that adapt the `Timer Module` used to generate timeouts with consumers that expect the IDL structures used to represent events.

5 Empirical Performance Evaluation

This section describes the methodology and results of a series of experiments that evaluate empirically the solutions we applied to resolve the design challenges presented in Section 4. The tests are designed to assess the performance of TAO's Real-time Event Service with respect to some critical metrics in distributed interactive simulations, such as latency, scalability, and overhead relative to the underlying communication mechanism.

5.1 Hardware and Software Overview

Our experiments were performed using two identically configured systems. Each was equipped with an 866Mhz Pentium III, with a 256Kb cache and 512Mb of RAM. The nodes had a

Fast Ethernet (100 Mbps) network adapter and were connected via an Ethernet hub. Aside from our tests, the network had no significant traffic nor was any other processing taking place on these hosts.

We ran the tests described below using Timesys Linux/RT v.2.2.14, which adds a resource kernel (RK) [31] to the core Linux kernel. Linux/RT enhances the real-time capabilities of Linux by providing fixed-priority scheduling with priority-inheritance and higher-resolution timers. Linux/RT is also binary compatible with Linux, *i.e.*, it is possible to have Linux and Linux/RT on the same hardware. Booting with the Linux/RT kernel starts the Linux/RT OS. The rest of the OS, *e.g.*, file-systems, C-libraries, compiler, and command-line tools, behaves just like regular Linux.

The tests are also based on the upcoming TAO 1.2 release, compiled using gcc-2.95.2, with the highest level of optimization possible (`-O3`). To improve predictability and performance, the tests were statically linked and native C++ exception handling was disabled. The list of compile-time options included: `Wpointer-arith`, `O3`, `fno-implicit-templates`, `D_POSIX_THREADS`, `D_POSIX_THREAD_SAFE_FUNCTIONS`, and `D_REENTRANT`.

5.2 Performance Test Descriptions and Results

We conducted the following tests:

1. *Measuring event service latency* — This test evaluates the the baseline performance of TAO's Real-time Event Service.
2. *Measuring event service overhead* — This test measures the amount of overhead incurred by TAO's Real-time Event Service.
3. *Measuring consumer latency scalability* — This test evaluates the scalability of TAO's Real-time Event Service with respect to the number of consumers.

Our goal in these experiments is to show that the federate architecture proposed in Section 4.9 does not impose undue overhead when compared to a more traditional implementation of the service. The experiments also show empirically how TAO's Real-time Event Service architecture scales linearly as the number of consumers increases. Moreover, our federated architecture proposed in Section 4.9 is an order of magnitude better than the more traditional centralized approach.

5.2.1 Measuring Event Service Latency

Overview. An important metric for any event service is *event latency*, *i.e.*, the time elapsed from when a supplier sends an event until the last consumer interested in the event receives it. RTI-NG application developers often have stringent requirements on this metric and cannot use an event service that incurs high latencies. Event latency varies according to the number of consumer and suppliers, the types of filters that are configured, and the configuration of the system and network. We therefore created a test to evaluate the *minimum event latency* of the TAO Real-time Event Service.

Experimental setup description. Measuring the latency in the event service is hard since events are delivered via a uni-directional flow of communication from suppliers to consumers. As shown below, event delivery time and jitter is comparable to the network propagation delay, because a distributed clock precision is bounded by the jitter (see [32]) measuring the latency in the configurations show on Figures 12 and 13 is impossible.

Fortunately, the latency for the centralized configuration can be measured directly using a consumer located in the same host as the supplier and measuring the roundtrip delay as shown in Figure 16.

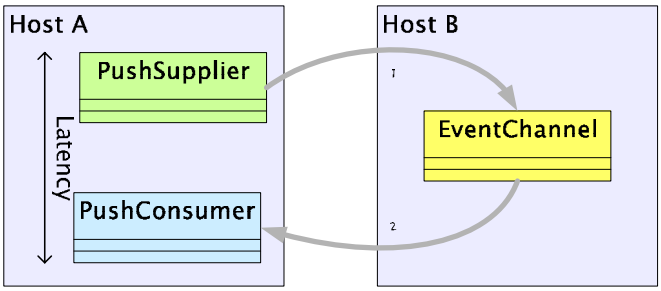

Figure 16: **Experimental Setup to Measure Event Service Latency**

Our first experiment sends events from a supplier to a remote event channel. The event channel then delivers the event to a consumer located in the same host as the supplier. Since messages are timestamped by the supplier, the consumer can measure the roundtrip delay simply by comparing the timestamp with the current time. For all our measures we use the high-resolution timer (under 2 nanosecond resolution) available on Pentiums. This timer is implemented via a special register that counts the number of clock ticks since the CPU was reset.

A different testbed is required to estimate the latency over a federated configuration of the event channel. As shown in Figure 17, our benchmark uses two event channels located in separate nodes. On the first node a supplier generates events of type A and a consumer subscribes for events of type B. Meanwhile, in the opposite node the configuration is reversed, *i.e.*, the consumer subscribes for events of type A and the supplier generates events of type B. One of the suppliers (*e.g.*, on the first node) generates and timestamps an event. Since the only consumer interested in the event is on the remote node the event service must send this event to its peer. Upon ar-

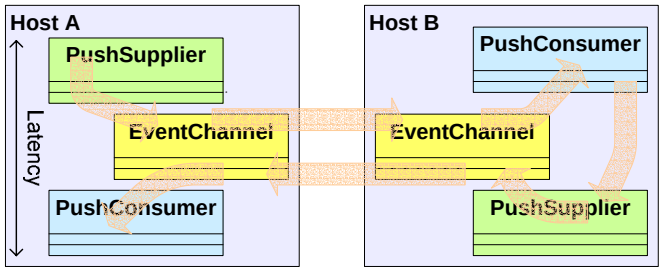

Figure 17: **Measuring Latency in a Simple Event Service Configuration**

rival to the remote consumer, the event type is modified (from A to B), and pushed through the local event channel. At this point the event is sent back to the remote consumer and the total roundtrip delay is calculated. Notice that with this configuration the message is sent twice through the event channel, first the the $A \rightarrow B$ direction and next in the opposite direction. In each case the event goes through the complete set of operations required to deliver it in the federated architecture. Therefore, the roundtrip delay measured through this experiment is comparable to two times the event latency in the federated architecture.

Despite this fact we present the roundtrip delay in the tables and figures below, we believe that this allows us to compare the results of all the experiments directly, and makes comparisons in the predictability and scalability of each configuration more i

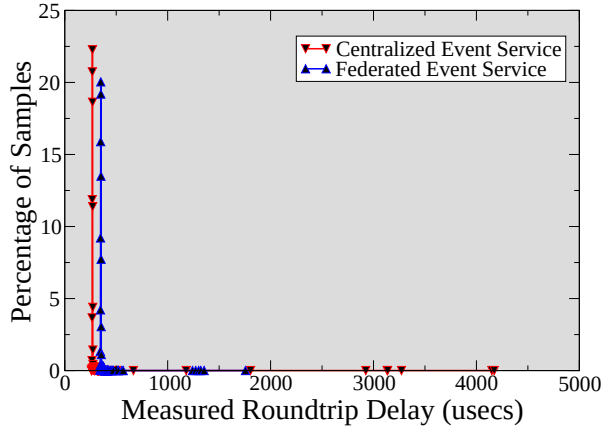

Figure 18: **Histogram for the Event Service Roundtrip Delay Results**

Minimum event latency results. The experiments take 50,000 samples and computes the average, minimum, maximum and standard deviation (or jitter) roundtrip delay, which are shown below in μsecs:

Configuration	Avg	Min	Max	Jitter
Centralized	271	260	4,173	36.89
Federated	351	343	1,756	12.89

As shown in the table, the estimated latency for the federated event channel ($175.5\mu secs = 351/2\mu secs$) is significantly lower than the measured latency for the centralized configuration (271 μsecs). The tests save all the samples, which we show in Figure 18. This figure also shows that a large number of samples are close to the average, and only a few samples are over one standard deviation from the average.

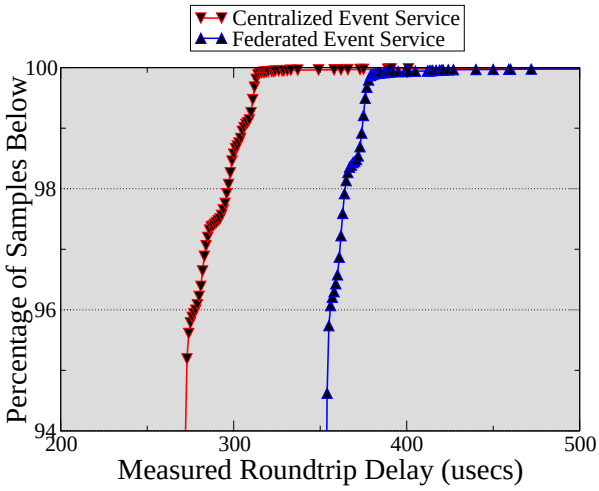

Figure 19: **Accumulated Histogram for the Event Service Roundtrip Delay Results**

Figure 19 illustrates the results of using the *cumulative histogram*.[5] This figure shows how over 99% of the samples in the centralized configuration are under 307 μsec, which is less than one standard deviation from the average. More than 99.9% of the samples are under 343 μsecs, which is less than 2 standard deviations over the average. Likewise, over 97% of the samples in the federated configuration are under one standard deviation from the average (363 μsecs) and over 99% of the samples are under two standard deviations from the average.

Results synopsis. Minimum event latency is an important measure of event service performance. Our experiment measures the minimum latency of the centralized event service and shows that for simple configurations it adds little or no jitter over the underlying network and ORB. Although the federated Event Service roundtrip delay is higher than the centralized configuration, the event *latency* is considerably lower.

[5]The cumulative function is obtained from a histogram by plotting the result of adding all the values smaller or equal than a given value, i.e. if $f(x)$ is the histogram function then $F(x) = \int_0^x f(u)du$ is the cumulative histogram. This representation facilitates the quantitative analysis of a system predictability.

Although the Federated Event Service adds some measurable overhead over the centralized configuration it scales better, as we discuss in Section 5.2.3.

5.2.2 Measuring Event Service Overhead

Overview. Another metric to be considered in an evaluation of an event service is the amount of *overhead* introduced over the underlying communication mechanisms. In this context we will deem any extra communication delay over the underlying transport mechanism *overhead*, we must therefore consider the roundtrip delay of sending a message over our CORBA implementation. For completeness, we also consider the roundtrip delay over TCP/IP.

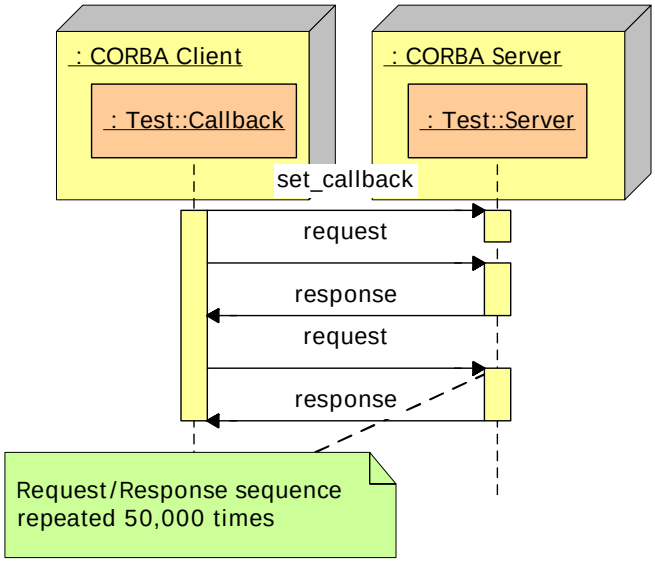

Figure 20: **Measuring Latency Over the ORB**

Experimental setup description. To measure TAO's Event Service overhead we implemented two benchmarks that measure the roundtrip delay over the following infrastructures:

- *TCP/IP* — This test consists of a client and server application. The client sends a small 64 byte message to the server over a TCP/IP socket. When the server receives the message it responds by sending a reply message back across the same socket. The client and server repeat this procedure 50,000 times and the client records the delay for each request/response pair.
- *The ORB using a callback object* — This test also consists of a client and a server, which is shown in Figure 20. To approximate the behavior of the event service, the client (1) creates a callback object and connects it to the remote server and then (2) sends a CORBA one-way request to the server. The server responds to the client via the callback interface configured earlier. As in the previous benchmark, the client in the ORB callback test records the roundtrip delay for each message.

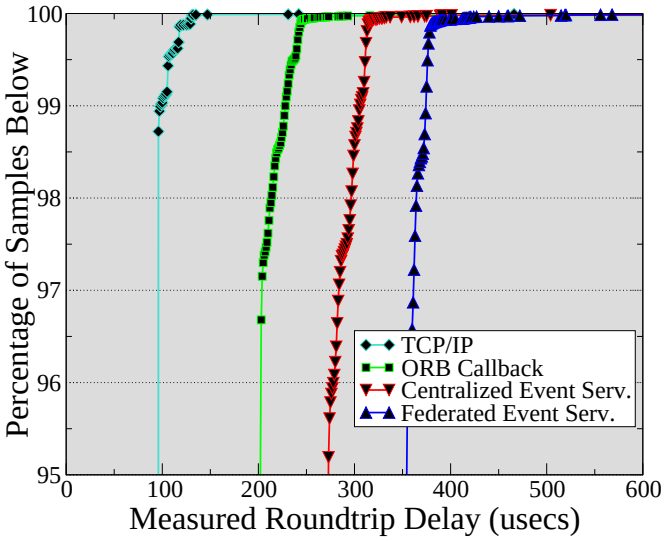

Figure 21: **Compared Roundtrip Delay Results for TCP/IP, the ORB, and the Event Service**

Event Service overhead results. The results of the TAO event service overhead experiments are summarized in the following table:

Transport	Avg	Min	Max	Jitter
TCP/IP	96	95	10,096	45.32
ORB Callback	201	191	3,198	31.49
Centralized EC	271	260	4,173	36.89
Federated EC	351	343	1,756	12.89

These results show that a significant portion of the event latency is due to the underlying ORB communication. In fact, for the centralized event service, over 60% of the propagation delay can be attributed to the ORB communication overhead. Moreover, as shown in Figure 21 all the experiments have a similar distribution, thus the Event Service does not introduce any additional jitter over the underlying ORB and TCP/IP transport mechanisms.

5.2.3 Measuring Consumer Latency Scalability

Overview. This experiment characterizes the event delivery latency as the number of consumers increases. Real-time applications need to determine the maximum latency over all the consumers connected to the event service. In a quality event service implementation, the maximum latency will increase linearly with the number of consumers. The average latency per consumer is the metric used to compare different event service architectures.

Experimental setup description. To perform these measurements, we extend the benchmarks described in Section 5.2.1. In particular, as shown in Figures 22 and 23 we create and connect multiple consumers for both tests. The test

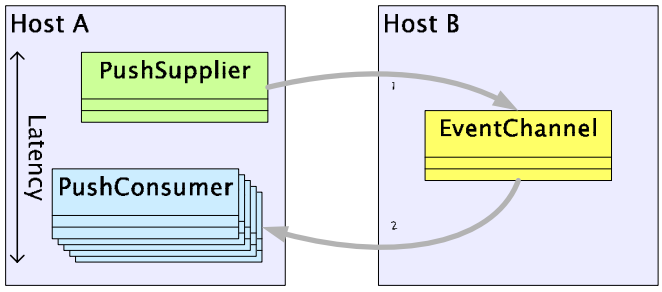

Figure 22: **Experimental Setup to Measure Scalability wrt to the Number of Consumers**

proceeds to timestamp and send messages as before. On each iteration, however, we record the worst case latency over the set of consumers. The number of consumers is increased from 5 to 100, in increments of 5. For a fixed number of consumers, the test performs 50,000 iterations and records the minimum, maximum, average and standard deviation.

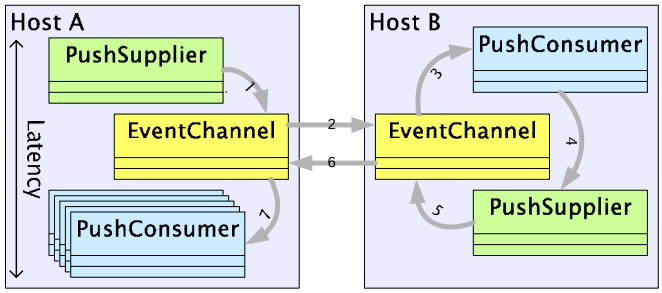

Figure 23: **Experimental Setup to Measure Scalability in the Federated Architecture**

Consumer latency scalability results. Figure 24 shows the results of the experiment described above. This figure illustrates how both architectures scale linearly with the number of consumers. As expected, however, the federated architecture scales better as a result of the reduced network overhead.

To estimate the cost-per-consumer, we apply linear regression over the data described above. Linear regression is an statistical technique that determines the best linear approximation for a set of data points. The results of a linear regression are thus the slope of such best linear approximation, the displacement on the abscissa and the *correlation factor*. This last value provides a measure of the confidence on the linear approximation, the closer the correlation factor is to 1 the better the approximation.

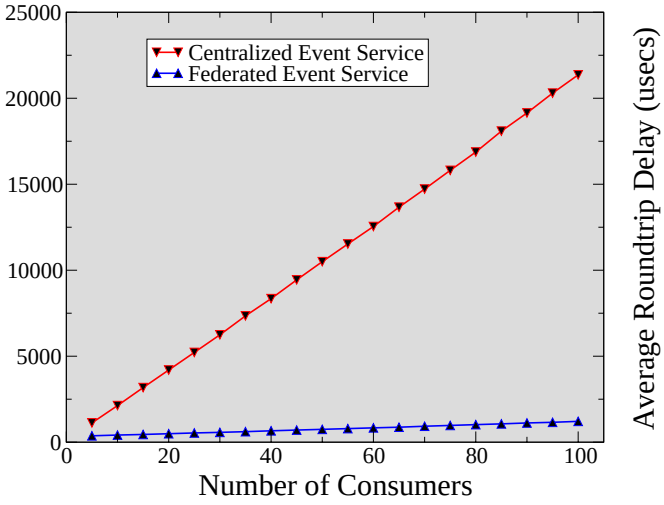

Figure 24: **Average Roundtrip for Last Consumer**

We use the slope of the linear approximation to estimate the cost-per-consumer, as shown in the following table:

Configuration	μsecs of Latency per-consumer	Correlation Factor
Centralized EC	212.9	0.999
Federated EC	8.8	0.999

Results synopsis. The cost per-consumer in the event service is an important metric to evaluate its performance. Though the centralized architecture can scale linearly with the number of consumers, our results show that a federated architecture can be at least an order of magnitude better with respect to this metric.

6 Related Work

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

The OMG has issued an specification for a Notification Service [33], 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. We believe that 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. Based on that idea, we have implemented a Notification Service for TAO [34] and used it to validate the feasibility of building reusable components for the Notification Service, CORBA Event Service and TAO's Real-time Event Service.

COBEA [22] 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, the 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.

In [21] 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.

Rajkumar, *et al.*, describe a real-time publisher/subscriber prototype developed at CMU SEI [23]. Their Publisher/Subscriber model is functionally similar to the CORBA Event Service, though it uses real-time threads to prevent priority inversion within the communication framework. An interesting aspect of the CMU model is the separation of priorities for subscription and event transfer so that these activities can be handled by different threads with different priorities. However, the model does not utilize any QoS specifications from publishers (suppliers) or subscribers (consumers). As a result, the message delivery mechanism does not assign thread priorities according to the priorities of publishers or subscribers. In contrast, the TAO Event Service utilizes QoS parameters from suppliers and consumers to guarantee the event delivery semantics determined by a real-time scheduling service.

The OMG Messaging specification [35] gives application developers control over several QoS parameters, such as one-way reliability and timeouts, and introduces type-safe asynchronous method invocation (AMI) models [14]. 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, for example, 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 risking lost messages. We have augmented TAO with the features [14] defined by the Messaging specification, which complement its Real-time Event Service implementation.

7 Concluding Remarks

Many distributed interactive simulation applications require support for asynchronous, event-based communication. The CORBA Event Service provides a flexible object-oriented model where event channels dispatch events to consumers on behalf of suppliers. TAO'S Real-time Event Service described in this paper augments this model with event channels that support

- Source and type-based filtering
- Event correlations
- Event channel federations
- Hardware and kernel-level filtering based on UDP/IP multicast and
- Large numbers of suppliers and consumers.

Our performance results in Section 5 demonstrate that using a single event channel for distribution yields poor scalability and high latency. We present an architecture to build federations of event channels that yields near optimal performance for collocated supplier/consumer pairs and does not affect remote event performance significantly. Moreover, the same architecture can be used to exploit multicast protocols, improving both the performance and scalability of the service. In general, our results illustrate that it is feasible to apply CORBA Object Services to develop high-performance, large-scale systems, complementing previous results where we show how CORBA can be used to build real-time embedded avionics systems [8].

The implementation of TAO's Real-time Event Service described in this paper is written in C++ and provided as a service in TAO [10]. TAO's Real-time Event Service is currently used as part of the HLA RTI-NG, which is the next-generation Run Time Infrastructure (RTI) implementation for the Defense Modeling and Simulation Organization's (DMSO) High Level Architecture (HLA). The source code and documentation for TAO and its Real-time Event Service implementation are freely available at URL `http://ace.cs.wustl.edu/Download.html`. Additional information about the HLA is available at URL `http://hla.dmso.mil`. The RTI-NG is available at URL `http://hlasdc.dmso.mil/RTISUP/hla_soft/hla_soft.htm`.

8 Acknowledgments

This work was funded in part by DMSO, SAIC, and Siemens. We particularly acknowledge the support and direction of those members of the RTI-NG group concerned with TAO, including Steve Bachinsky, Mike Mazurek, and Dave Meyer. In addition, we would like to thank Irfan Pyarali for his insightful comments on the concurrency and dispatching strategies in TAO's Real-time Event Service and Angelo Corsaro for his encouragement in the design of the performance benchmarks. Finally, thanks to Don Hinton for his comments that helped improve the quality of this paper.

References

[1] I. C. Society, ed., *Standard for Distributed Interactive Simulation – Communication Services and Profiles*. 345 E. 47th St, New York, NY 10017, USA: IEEE Computer Society, 1995.

[2] 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.

[3] R. E. Schantz and D. C. Schmidt, "Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications," in *Encyclopedia of Software Engineering* (J. Marciniak and G. Telecki, eds.), New York: Wiley & Sons, 2001.

[4] D. C. Schmidt, T. H. Harrison, and E. Al-Shaer, "Object-Oriented Components for High-speed Network Programming," in *Proceedings of the 1^{st} Conference on Object-Oriented Technologies and Systems*, (Monterey, CA), USENIX, June 1995.

[5] 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.

[6] J. A. Zinky, D. E. Bakken, and R. Schantz, "Architectural Support for Quality of Service for CORBA Objects," *Theory and Practice of Object Systems*, vol. 3, no. 1, pp. 1–20, 1997.

[7] D. C. Schmidt and C. Cleeland, "Applying Patterns to Develop Extensible ORB Middleware," *IEEE Communications Magazine*, vol. 37, April 1999.

[8] 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.

[9] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.3 ed., June 1999.

[10] 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.

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

[12] F. Kuhns, C. O'Ryan, D. C. Schmidt, and J. Parsons, "The Design and Performance of a Pluggable Protocols Framework for Object Request Broker Middleware," in *Proceedings of the IFIP 6^{th} International Workshop on Protocols For High-Speed Networks (PfHSN '99)*, (Salem, MA), IFIP, August 1999.

[13] 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.

[14] 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.

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

[16] I. Pyarali, C. O'Ryan, D. C. Schmidt, N. Wang, V. Kachroo, and A. Gokhale, "Applying Optimization Patterns to the Design of Real-time ORBs," in *Proceedings of the 5^{th} Conference on Object-Oriented Technologies and Systems*, (San Diego, CA), USENIX, May 1999.

[17] Object Management Group, *CORBAServices: Common Object Services Specification, Updated Edition*, 95-3-31 ed., Dec. 1998.

[18] F. Kuhl, R. Weatherly, and J. Dahmann, *Creating Computer Simulation Systems*. Upper Saddle River, New Jersey: Prentice Hall PTR, 1999.

[19] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.4 ed., Oct. 2000.

[20] M. Henning and S. Vinoski, *Advanced CORBA Programming With C++*. Reading, Massachusetts: Addison-Wesley, 1999.

[21] 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.

[22] 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.

[23] 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.

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

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

[26] Nancy Ann Lynch, *Distributed Algorithms*. Morgan Kaufmann Publishers, Inc., 1996.

[27] 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.

[28] Object Management Group, *Trading ObjectService Specification*, 1.0 ed., Mar. 1997.

[29] T. Kofler, "Robust iterators for ET++," *Structured Programming*, vol. 14, no. 2, pp. 62–85, 1993.

[30] D. C. Schmidt and S. D. Huston, *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks*. Reading, Massachusetts: Addison-Wesley, 2002.

[31] S. Oikawa and R. Rajkumar, "Portable RK: A Portable Resource Kernel for Guaranteed and Enforced Timing Behavior," in *Proceedings of the 5^{th} IEEE Real-Time Technology and Applications Symposium*, (Vancouver, British Columbia), IEEE, June 1999.

[32] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Norwell, Massachusetts: Kluwer Academic Publishers, 1997.

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

[34] 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.

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