Applying a Scalable CORBA Events Service to Large-scale Distributed Interactive Simulations

Carlos O'Ryan and David L. Levine
{coryan,levine}@cs.wustl.edu
Department of Computer Science, Washington University
St. Louis, MO 63130, USA*

Douglas C. Schmidt
schmidt@uci.edu
University of California, Irvine
Irvine, CA, 92697

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

This paper appeared in the proceedings of the 5^{th} Workshop on Object-oriented Real-time Dependable Systems, Montery, CA, November, 1999.

Abstract

Next-generation 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 Events Service provides a flexible model for asynchronous communication among distributed and collocated objects. However, the standard CORBA Events 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 Events Service can be implemented to support key QoS features. Second, it illustrates how to extend the CORBA Events Service so that it is better suited for distributed interactive simulations. 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 improve system scalability. Finally, it illustrates how an Events 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.

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

1 Introduction

Overview of distributed interactive simulations: Interactive simulations are useful tools for training personnel to operate equipment or 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 geographically disperse. For example, military units stationed around the globe can participate in joint training exercises, with human-in-the-loop airplane and tank simulators. Internet 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.

The QoS requirements on the software that support distributed interactive simulations are quite demanding. They combine aspects of distributed real-time computing, with the need for low-latency, high-throughput, multi-sender/multi-receiver communication over wide-range of autonomous and interconnected networks. Meeting these challenges requires new software infrastructures, such as those described in this paper.

Historically, distributed interactive simulation systems, such as DIS [1], were based on publisher/subscriber patterns [2]. Participants in the simulation declare the simulation data that they supply and consume. To exchange this data, distributed interactive simulation systems require an efficient and scalable communication infrastructure.

Typically, each participant in these *event-driven* systems consume and supply only a subset of the possible events in the system. By nature, however, these systems can vary dynamically, *e.g.*, consumers and suppliers can join and leave at

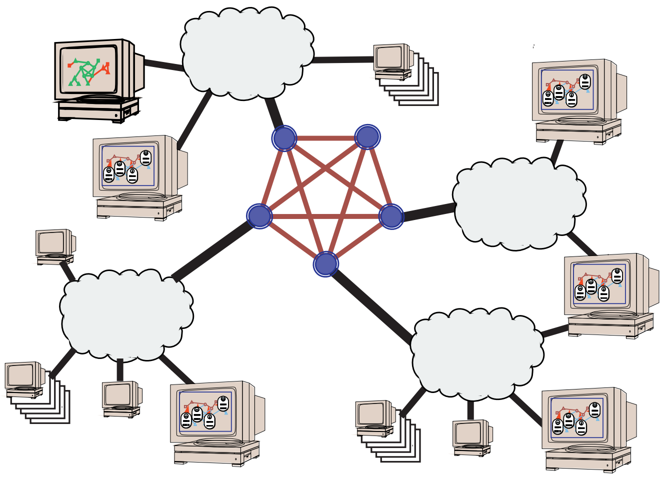

Figure 1: **Architecture of a Distributed Interactive Simulation Application**

arbitrary times. Likewise, the set of events published or subscribed to can also vary during the lifetime of the simulation.

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

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 systems from scratch. In practice, however, the environment in which these systems are developed places increasingly stringent constraints on time and effort for software development. Moreover, the increasing scarcity of qualified software professionals exacerbates the risk of companies failing to complete mission-critical projects, unless the scope of software development required for each project can be substantially constrained.

For these reasons, it is necessary that distributed interactive simulation systems be built largely from re-usable middleware. Middleware is software that resides between applications and the underlying operating systems, protocol stacks, and hardware in complex real-time 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 adaptive systems. In addition, it amortizes software lifecycle costs by leveraging previous design and development expertise and reifying key design patterns [7] in 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 widespread belief in the distributed interactive simulation community, however, that the efficiency, scalability, and predictability of COTS middleware, such as CORBA [9], is not suitable for next-generation large-scale simulation applications. Thus, if it can be demonstrated that the overhead of COTS middleware implementations can be optimized away, the resulting benefits make it a very compelling choice for large-scale simulation systems.

Our previous research 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 Events Service* to show how to support the QoS requirements of large-scale distributed interactive simulations using IP multicast to federate multiple event channels and conserve network resources. In addition, we describe the design of a flexible Events Service framework that allows developers to select implementation strategies most suitable to their application domain.

The remainder of this paper is organized as follows: Section 2 outlines the CORBA reference model, the CORBA Events Service, and the TAO real-time (RT) Events Service; Section 3 discusses the optimizations and extensions we added to the standard CORBA Events Service to support large-scale distributed interactive simulation applications; Section 4 compares our work with related research; and Section 5 presents concluding remarks.

2 Technical Background

This section outlines the CORBA reference model, the TAO [10] Real-time CORBA [17] ORB, the CORBA Events

Service, and the real-time (RT) Events Service integrated with TAO.

2.1 Synopsis of CORBA

CORBA is a distributed object computing middleware specification [9] being standardized by the Object Management Group (OMG). CORBA is designed to support the development of flexible and reusable service components and distributed applications by (1) separating interfaces from (potentially remote) object implementations and (2) automating many common network programming tasks, such as object registration, location, and activation; request demultiplexing; framing and error-handling; parameter marshaling and demarshaling; and operation dispatching. Figure 2 illustrates the primary components in the OMG Reference Model architecture.

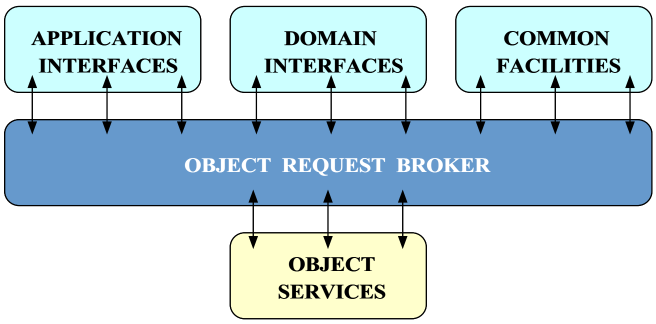

Figure 2: **OMG Reference Model Architecture**

At the heart of the OMG Reference Model is the *Object Request Broker* (ORB). CORBA ORBs allow clients to invoke operations on target object implementations without concern for where the object resides, what language the object is written in, the OS/hardware platform, or the type of communication protocols and networks used to interconnect distributed objects [18]. Because CORBA is a standard, there are now a wide range of companies and organizations that provide interoperable ORB implementations targeted for various domains.

2.2 Synopsis of the TAO Real-time ORB

TAO is a freely-available, open-source, standards-compliant [9], real-time CORBA [17] ORB that provides end-to-end quality of service guarantees to applications by *vertically* (*i.e.*, network interface $\leftrightarrow$ application layer) and *horizontally* (*i.e.*, end-to-end) integrating CORBA middleware with OS I/O subsystems, communication protocols, and network interfaces. TAO is implemented using the ACE framework [19], which contains a rich set of high-performance and real-time reusable software components. These components automate common communication tasks such as connection establishment, event demultiplexing and event handler dispatching, message routing, dynamic configuration of services, and flexible concurrency control for network services. ACE and TAO have been ported to many real-time OS platforms including VxWorks, LynxOS, CHORUS/ClassiX, and most POSIX 1003.1c implementations. In addition, ACE and TAO run on general-purpose operating systems, such as Windows NT and Solaris 2.x, that provide real-time threads, though they lack certain features required for hard real-time systems [20, 21, 22].

2.3 Synopsis of the CORBA Events Service

Many distributed applications exchange asynchronous requests using *event-based* execution models [23, 24, 25]. To support these common use-cases, the OMG defined a CORBA Events Service component in the CORBA Object Services (COS) layer, as shown in Figure 2. The COS specification [26] presents architectural models and interfaces that factor out common object services, such as persistence [27], security [28], transactions [29], fault tolerance [30], and concurrency [31].

The CORBA Events Service defines *supplier* and *consumer* participants that are designed to alleviate some of the restrictions with standard CORBA invocation models. As shown in Figure 3 suppliers generate events and consumers process

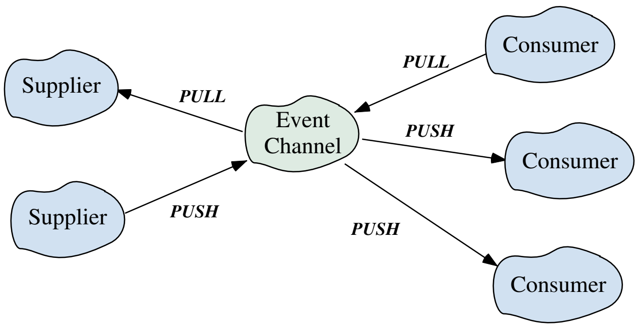

Figure 3: **Participants in the COS Events Service Architecture**

events received from suppliers. This figure also illustrates the *event channel*, which is a mediator [32] that propagates events to consumers on behalf of suppliers. By using an event channel, events can be delivered from suppliers to consumers without requiring these participants to know about each other explicitly. In addition, event channels can simplify application software by implementing group communication and serving as a replicator [33], broadcaster, or multicaster that forward events from one or more suppliers to multiple consumers.

In theory, the CORBA Events Service addresses many needs of event-based applications. In practice, however, the standard CORBA Events Service specification lacks other important features, such as *efficient event filtering*, *group communication protocols*, *minimal data copying*, and *scalable dispatching strategies*, that are required by next-generation distributed interactive simulation systems, which motivates the need for the TAO RT Events Service described next.

2.4 Synopsis of the TAO RT Events Service

To alleviate the limitations with the standard CORBA Events Service, therefore, we have developed a *Real-time (RT) Events Service* as part of the TAO project [10] at Washington University. Figure 4 illustrates the key architectural components in TAO and their relationship to the real-time Events Service.

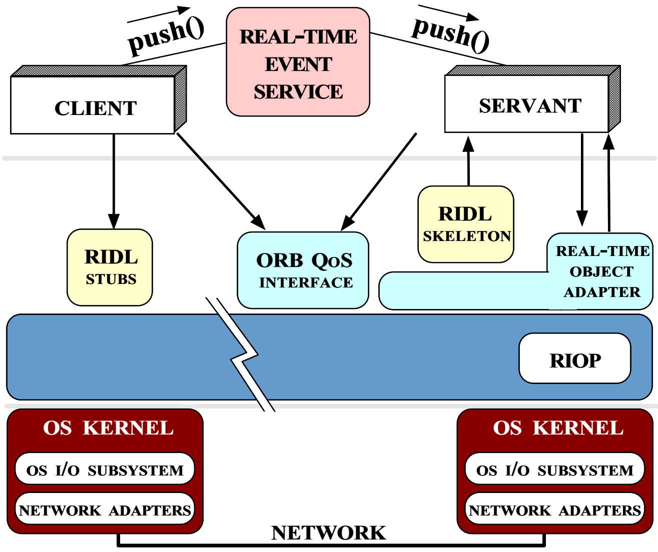

Figure 4: **Architecture of the TAO ORB Endsystem**

TAO's original RT Events Service [8] augmented the CORBA Events Service model by providing source-based and type-based filtering, event correlations, and real-time dispatching. The RT Events Service described in this paper is a second-generation implementation designed to satisfy the QoS requirements of large-scale systems, such as distributed interactive simulations. To meet these new requirements, TAO's new RT Events Service can be configured to form a *federated mesh* of event channels. The event channel peers in this federation can exchange events using the standard CORBA IIOP interoperability protocol (which is based on TCP/IP unicast), UDP broadcast, or UDP multicast messages. TAO's implementation also supports multiple dispatching, filtering, and update strategies that can be selected during application initialization using the Service Configurator [34] pattern. This choice can be scripted in a configuration file to enable rapid prototyping, evaluation, and adaptation during the initialization process.

In addition, TAO's RT Events Service can be integrated with TAO's RT Scheduling Service [10, 5] to support applications, such as avionics mission computing [8], with stringent end-to-end real-time requirements. This Scheduling Service can analyze and assess the schedulability of the overall system [10]. TAO's RT Scheduling Service is designed as a framework [5] that can be configured to use multiple real-time scheduling policies, such as rate monotonic (RMS) [35] and maximum urgency first (MUF) [36]. Once the feasibility of the system's schedule has been determined, TAO's RT Events Service and Scheduling Service collaborate to support various strategies for priority-based event dispatching and preemption.

3 Overview of TAO's Real-time Events Service

3.1 Overcoming Limitations with the CORBA Events Service

The standard COS Events Service Specification lacks several important features required by large-scale distributed interactive simulations. Chief among these missing features include centralized event filtering, efficient and predictable event dispatching, periodic event processing, and event correlations.[1] To resolve these limitations, we have developed a Real-time Events Service (RT Events Service) as part of the TAO project [10]. TAO's RT Events Service extends the COS Events Service specification to satisfy the quality of service (QoS) needs of real-time applications in domains like distributed interactive simulations, avionics, telecommunications, and process control.

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

3.1.1 Support for Centralized Event Filtering

In a large-scale distributed interactive simulation, not all consumers are interested in all events generated by all suppliers. Although it is possible to let each application perform its own filtering, this solution wastes network and computing resources. Ideally, therefore, the Events Service should send an event to a particular consumer only if the consumer has explicitly subscribed for it. Care must be taken, however, to ensure that the subscription process used to support filtering does not itself cause undue burden on distributed system resources.

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

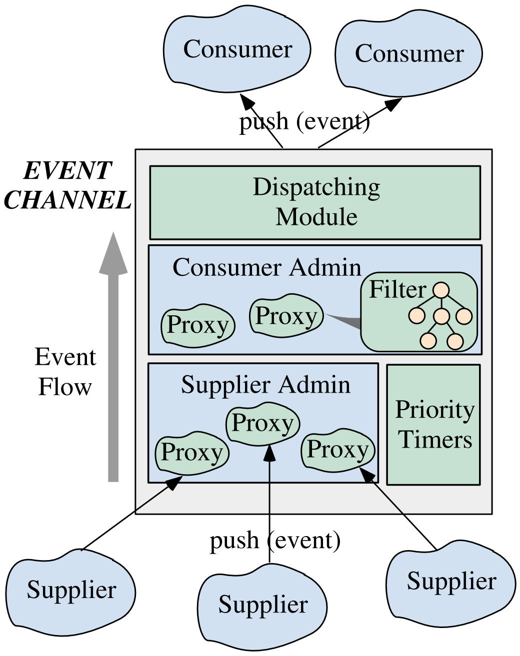

Figure 5: **RT Events Service Architecture**

It is possible to implement filtering using standard COS event channels [26]. For instance, channels can be chained 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 COS event channels increase the number of hops a message must travel between 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, therefore, TAO's RT Events 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 consumers' dependencies.

3.2 TAO's RT Events Service Architecture

TAO's RT Events Service is implemented using the Mediator Pattern [32]. The heart of the RT Events 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 the event channel components can find each other.

When a `ProxyPushConsumer` receives an event from the application it iterates over the set of `ProxyPushSupplier` that represent the potential consumers interested in that event. Section 3.3.2 describes how that set is computed. Each `ProxyPushSupplier` checks to see if the event is relevant for its consumer. This check is performed by the *filter hierarchy* described in section 3.3.1. If the event is of interest to a consumer a `Dispatching Strategy` selects the thread that should dispatch the event to the consumer. Section 3.3.6 discusses various tradeoffs to consider when selecting the dispatching thread strategy.

For real-time applications that require periodic event processing, the Events Service can contain an optional `Timer Module`. Section 3.3.14 outlines several strategies for generating timer events. Each strategy possesses different predictability and performance characteristics and different resource requirements.

3.3 Design and Implementation Challenges

Section 3.2 outlined the core components of the CORBA Events Service that are defined by IDL interfaces. Below, we describe how we have systematically applied key the design patterns, such as Builder, Command, Composite, and Strategy from the GoF book [32], Strategized Locking [37], and Service Configurator [34], to tackle the design and implementation challenges posed by TAO's RT Events Service. Because these patterns are applicable to related systems, we document how we applied and composed these patterns in our design to achieve our performance and scalability goals.

3.3.1 Implementing an Extensible and Efficient Filtering Framework

Context: TAO's real-time Events 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 RT Events Service, however. For example, many distributed interactive simulations do not require correlation and some Events Service applications do not require filtering either. 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: The event channel should support the addition of new filtering primitives flexibly and efficiently. For instance, an event channel should allow new filtering primitives, *e.g.*, 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 [32]: 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 a filter composition tree. New filtering primitives can be implemented as leaves in the composition tree. The primitives can create complex filter hierarchies using disjunction and conjunction composites, thereby providing applications with substantial expressive power.

To control the creation of the concrete filters we use the Builder Pattern [32], which separates the construction of a complex object from its representation. In our case, we build the filter hierarchy from the subscription IDL structures, though we are implementing the complete Trader Constraint Language [38] filtering language. It is interesting to note that this change will not affect the overall architecture of the system, however, because the Builder pattern isolate us from the concrete form that the subscription takes.

Because filters are built on a per-consumer basis, no lookup operations are required to find (1) a consumer's correlation queues or (2) the list of event types to which a consumer subscribes. At run-time, an event channel's filtering engine simply traverses the tree of filters starting from the top. *Disjunction* nodes in the filter pass the event to the children until one of them accepts it, whereas *conjunction* nodes pass it down until *all* of them accept the event.

3.3.2 Improving Scalability with the Number of Consumers

Context: In many applications only a small fraction of the consumers are interested in a particular event. If the implementation was to query each `ProxyPushSupplier` to check if an event is interesting it will scale very poorly with the number of events.

Problem: Reduce the time required to dispatch an event by reducing the set of consumers tested.

Solution → pre-compute the set of consumers for each `ProxyPushConsumer` object: we can use the publications declared by the Supplier to find what consumers could be interested in the events generated by that supplier, we use the `Filter` hierarchy in each `ProxyPushSupplier` to estimate if the corresponding consumer is willing to receive at least one of the events published by the supplier. If the consumer is not interested in any of the events we can leave it out of the set and improve the overall performance of the system.

3.3.3 Reducing Memory Footprint

Context: Not all applications have a small fraction of the consumers interested in the events generated by each supplier. Most of the suppliers may actually generate events that are interesting to most consumers. In such a case it is actually counter productive to use the optimization described above, and it is more efficient to use a single global collection of consumers, reducing the memory footprint and reducing the time required to update the collection of consumers.

Problem: Give the application developer control over the algorithm used to build the consumer sets.

Solution → use the Strategy Pattern [32] where a family of algorithms is represented by classes sharing a common ancestor, the clients use the ancestor class and thus can select different algorithms without requiring any changes. In our case we use this pattern to encapsulate the exact algorithm used to control the number of collections are how are they updated. Notice that the variations mentioned so far are in no way exhaustive, for example, we can keep separate collections of consumer for each event *type*. The framework implemented in TAO's RT Events Service has been designed to support that use case too.

3.3.4 Supporting Re-entrant Calls while Dispatching Events

Context: To dispatch an event to multiple consumers, an event channel must iterate over its collection of `ProxyPushSupplier` objects. In some concurrency models, such as the single-threaded or reactive dispatching strategies described in Section 3.3.6, the same thread that iterates over a collection executes the upcall to consumers. Consumers are then allowed to push new events, add or remove consumers and suppliers, and in general, call back into the event channel and its internal components.

Problem: The event channel should support re-entrant calls during event dispatching, regardless of concurrency model. However, many iterator implementations become invalidated when their data structure is modified [39], thus the `ProxyPushSupplier` set cannot be changed when a thread is iterating over it. Simply locking the collection is inappropriate because the application will either dead-lock if the upcall changes the collection or will invalidate iterators if we use recursive locks. Another inappropriate alternative is to copy the `ProxyPushSupplier` collection *before* starting the iteration. Although this works with small collections, it performs poorly for large-scale distributed interactive simulation applications.

Solution → apply lazy evaluation to delay certain operations: TAO's event channel keeps track of how many threads are iterating over each collection of `ProxyPushSupplier` objects. Before performing changes that would invalidate other iterators, it checks to ensure there are no concurrent iterations in progress. If there are, the operation is stored as a

Command object [32]. When there are no threads iterating on the collection, all delayed command operations are executed sequentially.

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

3.3.5 Reducing Synchronization Overhead

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

Problem: 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. In particular, applications may choose to decouple (1) threads that iterate over the `ProxyPushSupplier` collections from (2) threads that perform consumer upcalls. This decoupling (1) yields more predictable behavior in hard real-time systems, (2) allows the application to re-order the events based to perform dynamic scheduling, and (3) isolates event suppliers from the execution time of consumer upcalls.[2]

Solution → use the Strategy Pattern [32] TAO's event channel uses the 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 object that updates the state in the collection to indicate that a thread is performing an iteration. When this lock is released, any delayed operations are executed. An alternative strategy provides a more efficient lock that simply acquires and releases a mutex.

3.3.6 Selecting the Thread to Dispatch an Event

Context: Once the 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 6, there are several alternatives. Using the same thread that received the event is efficient, *e.g.*, it reduces context switching, synchronization, and data copying overhead [16], but potentially exposes the 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. In turn, highly-scalable systems may want to use a pool of threads to dispatch the events, thereby taking advantage of advanced hardware and overlapping I/O and computation.

[2] Although this design may increase context switching overhead, many applications can tolerate this if developers already use separate threads to perform upcalls.

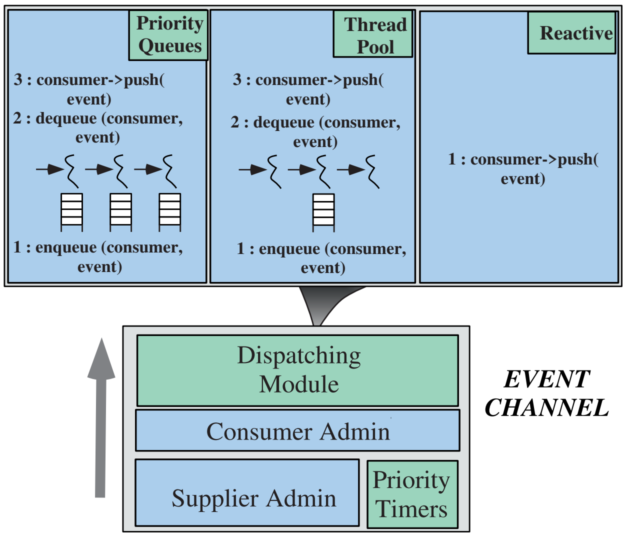

Figure 6: **Dispatching Strategies Supported in TAO's Event Channel**

Problem: An event channel must provide a flexible infrastructure to determine which thread dispatches an event to a particular consumer.

Solution → use the Strategy Pattern [32]: 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] 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 3.3.5.

3.3.7 Configuring Event Channel Strategies Consistently

Context: To adapt to various use-cases, TAO's event channel provides myriad strategies that can be configured by application developers. Often, the choice of one strategy affects other strategies. For example, if the 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. Thus, a simpler strategy to manipulate those collections can be used.

Problem: Selecting a suitable combination of strategies can impose an undue burden on the developer and yield inefficient or semantically incompatible strategy configurations. Ideally, developer should be able to select 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 [32]: to control the creation of all the objects in the event channel. In this pattern a single interface creates families of related on dependent objects. We use it to provide a single point to select all the event channel strategies, and avoid incompatible choices. Concrete implementations of this Abstract Factory ensure that strategies and components are semantically compatible and collaborate correctly.

3.3.8 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 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 the event channel is statically configured, it is hard evaluate various combinations without time consuming recompiling/relinking.

Solution → use the Service Configurator Pattern [34]: This pattern allows applications to dynamically or statically configure service implementations. We use ACE's implementation of this pattern to dynamically load Abstract Factorys that create various event channel configurations. Our implementation includes a default Abstract Factory that uses the scripting features of the ACE Service Configurator framework. By using this default, developers or end-user can modify event channel configuration at initialization time by simply changing entries in a configuration file.

3.3.9 Exploiting Locality in Supplier-Consumer pairs

Context: Because it is based on CORBA, TAO's event channels can be accessed transparently across distribution boundaries. Many applications want to be shielded from distribution aspects, while simultaneously achieving the optimal performance.

Problem: There are use-cases where distribution transparency may not yield the most *effective* configuration. For example, Figure 7 illustrates a scenario where most or all consumers for common events may 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 right back to the

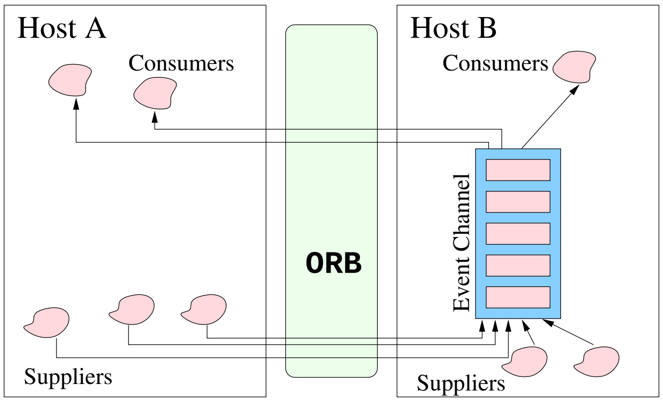

Figure 7: **A Centralized Configuration of the TAO RT Events Service**

same process, is a waste of network resources and unnecessarily increases latency. Likewise, there may be multiple remote consumers expecting the same event. Ideally, bandwidth should be conserved in this case by sending a single message across the network to all those remote consumers.

Solution → federate event channels: Figure 8 illustrates the use of event channel *gateways* to federate event channels. Each gateway is a CORBA object that connects to the local

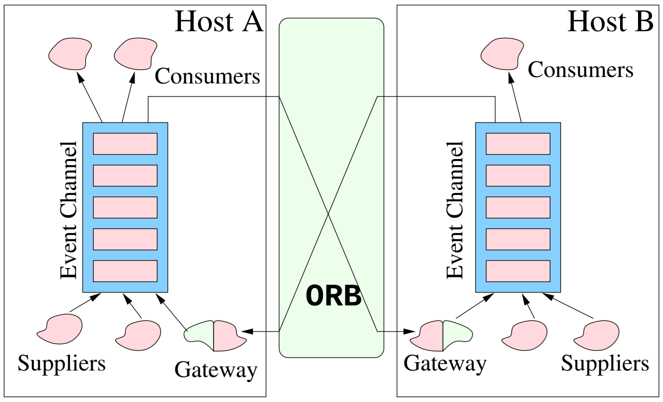

Figure 8: **A Federated Event Channel Configuration**

event channel as suppliers and connects to the remote event channel as a consumer. To reduce network traffic, the gateway must subscribe to events that are of interest to at least one local consumer. Suppliers and consumers connect directly to their local event channel, this results in reduced average latency for all the consumers in the system, because consumers and suppliers exhibit locality of reference, *i.e.* most consumers for any one event are in the same domain than the supplier that generated the event. Moreover, if multiple remote consumers are interested in the same event only one message is sent to the remote gateway, thereby minimizing network utilization.

A straightforward and portable way to implement a gateway is to use IIOP to receive a single event from a remote event channel and propagate it, through the local event channel, to multiple consumers. This design conserves network resources and increases latency only for an uncommon case, *i.e.*, where local consumers receive events directly through the local event channel.

3.3.10 Updating the Gateway Subscriptions and Publications

Context: In a dynamic environment, subscriptions change constantly.

Problem: To use network resources efficiently, the event channel gateways described in Section 3.3.9 must avoid subscribing to all events in a remote events channel. Otherwise, the locality of reference benefits of event channel federation are lost.

Solution → use the Observer Pattern [32]: In this pattern, whenever one object changes state, all its dependents are notified. In our case, we propagate the changes in the subscription and publication lists to any and all the interested parties. The implementation of *observables*, *i.e.*, event channels, can be strategized. Thus, if applications know *a priori* that there will be no observers at run-time, they can configure the event channel to disable this feature, thereby eliminating the overhead required to update the (empty) observer list.

3.3.11 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 WANs. Thus, as the number of nodes increase, sending the same event multiple times across a network is not scalable.

Solution → use a multicast or broadcast protocol: TAO's event channel can be configured to use UDP to multicast events. As with the gateways describe in Section 3.3.10, a special consumer can subscribe to all the events generated by local suppliers, as shown in Figure 9. This consumer uses 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 3.3.10 is used to modify the subscriptions and publications of multicast gateways dynamically.

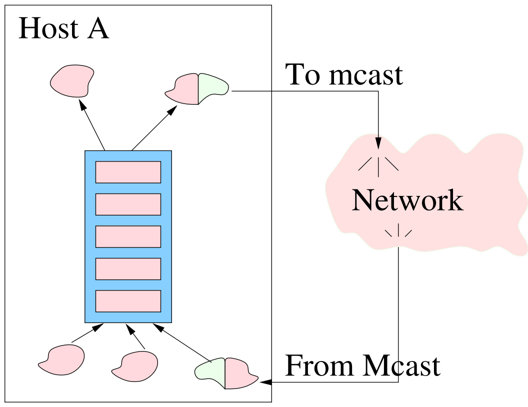

Figure 9: **Using Multicast in Federated Event Channels**

3.3.12 Exploiting Hardware- and Kernel-level Filtering

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

Problem: The event channel must select the multicast group used for each type of event in a globally consistent way, but the mapping between events and multicast groups may be different in each application. Applications 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 the mapping. Moreover, applications that need highly scalable fault tolerance may choose to distribute the mapping service across the network. An event channel must be able to satisfy all those scenarios, without imposing any one 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. On the receiving side, the gateways consult this service to decide which multicast groups to subscribe to, based upon the current set of subscriptions in the local event channel. Advanced operating systems and network adapters can use this information to process only the relevant multicast traffic.

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

One solution is to use the Events Service to propagate this information.

3.3.13 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 *tank position events*; if both a local and remote consumer are interested in the event 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. However, this design would complicate the gateway architecture for simpler use-cases and requires 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 deallocated and not forwarded. Usually event channel federations are fully connected, *i.e.*, all event channels have a gateway to each of their peers. Thus, setting the TTL counter to 1 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 code has been optimized to reduce data copying, only the event header requires a copy to change the TTL counter, the payload, that usually contains most of the data, is not touched.

3.3.14 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 the periodic tasks complete before their dead time.

Hard real-time applications may assign different priorities to their timer events. To avoid priority inversions, therefore, events should be generated and dispatched by thread at the appropriate priorities. Soft real-time applications or best-effort applications often impose no such strict requirements on timer priorities and thus are 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 the periodic events.

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

Solution → use the Strategy Pattern [32] to dynamically select the mechanisms used to generate timeout events. 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.

4 Related Work

Conventional approaches to quality of service (QoS) enforcement have typically adopted existing solutions from the domain of real-time scheduling [35], fair queuing in network routers [40], or OS support for continuous media applications [41]. In addition, there have been efforts to implement new concurrency mechanisms for real-time processing, such as the real-time threads of Mach [42] and real-time CPU scheduling priorities of Solaris [21].

However, QoS research at the network and OS layers has not necessarily addressed key requirements and usage characteristics of distributed object computing middleware [10]. For instance, research on QoS for network infrastructure has focused largely on policies for allocating bandwidth on a per-connection basis [43]. Likewise, research on real-time operating systems has focused largely on avoiding priority inversions and non-determinism in synchronization and scheduling mechanisms [44]. In contrast, the programming model for developers of OO middleware focuses on invoking remote operations on distributed objects. Determining how to map the results from the network and OS layers to OO middleware is the main focus of the TAO research project.

There are several commercial CORBA-compliant Events Service implementations available from multiple vendors, such as IONA and Inprise. IONA also produces OrbixTalk, which is a messaging service based on IP multicast. Because the CORBA Events 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 new Notification Service [45]. This Notification Service is a superset of the COS Events 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. However, the Notification Specification does not address the implementation issues related to the Notification Service. We believe that the patterns and techniques used in the implementation of TAO's RT Events Service can be used to improve the performance and predictability of Notification Service implementations. Based on that idea we have recently begun the implementation of a Notification Service for TAO, were we will research the feasibility of building reusable components for the Notification Service, CORBA Events Service and TAO's RT Events Service.

Although there has been research on formalisms for real-time objects [46], there is relatively little published research on the design and performance of real-time OO systems. Our approach is based on the OMG CORBA distributed object computing standard. In this paper, we focus on the design and performance of various strategies for implementing QoS in real-time ORBs [10].

The QuO project at BBN [6] has defined a model for communicating changes in QoS parameters between applications, middleware, and the underlying endsystems and network. We have integrated QuO and TAO's RT Events Service to demonstrate dynamic QoS support [47]. Other research on the CORBA Events Service [25, 24] describe techniques for optimizing events service performance for filtering and message delivery. As with QuO, the focus of this work is not on assuring CPU availability for events with hard real-time deadlines.

Rajkumar, *et al.*, describe a real-time publisher/subscriber prototype developed at CMU SEI [23]. Their Publisher/Subscriber model is functionally similar to the COS Events 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 Events Service utilizes QoS parameters from suppliers and consumers to guarantee the event delivery semantics determined by a real-time scheduling service.

The OMG recently adopted the Messaging specification [48], which 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 Events Service, for example, it defines several levels of reliability for `oneway` calls, this feature could be used in Events Service implementations to improve decoupling of the clients, without risking lost messages. We are rapidly adding to TAO the features defined by the Messaging specification, thus they will complement TAO's RT Events Service implementation.

5 Concluding Remarks

Many distributed interactive simulation applications require support for asynchronous, event-based communication. The COS Events Service provides a flexible OO model where event channels dispatch events to consumers on behalf of suppliers. TAO'S real-time (RT) Events Service described in this paper augments this model with event channels that support (1) source and type-based filtering, (2) event correlations, (3) event channel federations, (4) hardware and kernel-level filtering based on IP multicast, and (5) large numbers of suppliers and consumers.

The implementation of TAO's Real-time Events Service described in this paper is written in C++ and provided as a TAO [10] service. TAO's RT Events Service is currently used as part of the HLA RTI-NG. This 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 RT Events Service implementation are freely available at URL `www.cs.wustl.edu/ schmidt/TAO.html`. Additional information about the HLA is available at `hla.dmso.mil`. The RTI-NG is available at URL `hlasdc.dmso.mil/RTISUP/hla_soft/hla_soft.htm`.

6 Acknowledgments

This work was funded in part by SAIC and Boeing. 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 the TAO Events Service.

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*. Wiley and Sons, 1996.

[3] A. Gokhale and D. C. Schmidt, "Techniques for Optimizing CORBA Middleware for Distributed Embedded Systems," in *Proceedings of INFOCOM '99*, Mar. 1999.

[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," *The International Journal of Time-Critical Computing Systems, special issue on Real-Time Middleware*, 2000, to appear.

[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, 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), 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*, To appear 1999.

[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 *Submitted to the Middleware 2000 Conference*, 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, *Realtime CORBA Joint Revised Submission*, OMG Document orbos/99-02-12 ed., March 1999.

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

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

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

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

[22] D. L. Levine, D. C. Schmidt, and S. Flores-Gaitan, "An Empirical Evaluation of OS Support for Real-time CORBA Object Request Brokers," in *Proceedings of the Multimedia Computing and Networking 2000 (MMCN00) conference*, (San Jose, CA), ACM, Jan. 2000.

[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] 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.

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

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

[27] Object Management Group, *Persistent State Service 2.0 Specification*, OMG Document orbos/99-07-07 ed., July 1999.

[28] Object Management Group, *Security Service Specification*, OMG Document ptc/98-12-03 ed., December 1998.

[29] Object Management Group, *Transaction Services Specification*, OMG Document formal/97-12-17 ed., December 1997.

[30] Object Management Group, *Fault Tolerance CORBA Using Entity Redundancy RFP*, OMG Document orbos/98-04-01 ed., April 1998.

[31] Object Management Group, *Concurrency Services Specification*, OMG Document formal/97-12-14 ed., December 1997.

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

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

[34] P. Jain and D. C. Schmidt, "Dynamically Configuring Communication Services with the Service Configurator Pattern," *C++ Report*, vol. 9, June 1997.

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

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

[37] D. C. Schmidt, "Strategized Locking, Thread-safe Decorator, and Scoped Locking: Patterns and Idioms for Simplifying Multi-threaded C++ Components," *C++ Report*, vol. 11, Sept. 1999.

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

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

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

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

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

[43] Z. D. Dittia, G. M. Parulkar, and J. R. Cox, Jr., "The APIC Approach to High Performance Network Interface Design: Protected DMA and Other Techniques," in *Proceedings of INFOCOM '97*, (Kobe, Japan), pp. 179–187, IEEE, April 1997.

[44] R. Rajkumar, L. Sha, and J. P. Lehoczky, "Real-Time Synchronization Protocols for Multiprocessors," in *Proceedings of the Real-Time Systems Symposium*, (Huntsville, Alabama), December 1988.

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

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

[47] J. P. Loyall, A. K. Atlas, R. Schantz, C. D. Gill, D. L. Levine, C. O'Ryan, and D. C. Schmidt, "Flexible and Adaptive Control of Real-Time Distributed Object Middleware," *Submitted to The International Journal of Time-Critical Computing Systems*, 2000.

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