The Design and Performance of a Real-time Notification Service

Pradeep Gore and Irfan Pyarali	Christopher D. Gill	Douglas C. Schmidt
OOMWorks LLC	Washington University	Vanderbilt University
Metuchen, NJ	St. Louis, MO	Nashville, TN
{pradeep,irfan}@oomworks.com	cdgill@cse.wustl.edu	d.schmidt@vanderbilt.edu

Abstract

Many distributed real-time and embedded (DRE) applications require a scalable event-driven communication model that decouples suppliers from consumers and simultaneously supports advanced quality of service (QoS) properties and event filtering mechanisms. The CORBA Notification Service provides publisher/subscriber capabilities designed to support scalable event-driven communication by routing events efficiently between suppliers and consumers, enforcing QoS properties (such as reliability, priority, ordering, and timeliness), and filtering events at multiple points in a distributed system. The standard CORBA Notification Service is insufficient, however, to enforce predictable communication needed by DRE applications and does not leverage Real-time CORBA capabilities, such as end-to-end priority preservation, priority models, and scheduling.

This paper makes three contributions to the study of scalable real-time notification services for DRE applications. First, we describe the requirements of the OMG Request for Proposals (RFP) on Real-time Notification, which seeks solutions to the problem of enforcing real-time properties by enhancing the standard CORBA Notification Service. Second, we explain how we have addressed key design challenges faced when implementing a Real-time Notification Service for TAO, which is our CORBA-compliant real-time Object Request Broker (ORB). We discuss the optimizations used to improve the scalability of TAO's Real-time Notification Service, which integrates Real-time CORBA features (such as thread pools, thread lanes, and priority models) to provide real-time event communication by dedicating thread resources with minimal locking overhead in the critical path of event propagation. Finally, we analyze the results of empirical benchmarks of the performance and predictability of TAO's Real-time Notification Service. These results show that the static real-time assurances provided by Real-time CORBA are maintained within the more flexible context of TAO's Real-Time Notification Service.

Keywords: Distributed real-time embedded (DRE) systems, Quality of Service (QoS), CORBA, Event /Notification Services.

1. Introduction

Many distributed real-time and embedded (DRE) applications (such as real-time avionics mission computing systems, distributed interactive simulations, and computer-assisted stock trading) require an event-based communication model. Client/server communication via distribution middleware (such as CORBA [5], Java RMI and COM+) typically support a synchronous method invocation (SMI) model, where a client invokes a two-way operation on a target object implemented by a server and then blocks waiting for the response. This model has limitations, however, stemming from its tight coupling between client and server lifetimes, synchronous communication, and point-to-point communication.

The deficiencies of the SMI model can be resolved by publisher/subscriber services that support the decoupling of event suppliers and consumers, asynchronous communication, and transparent group communication. For example, the CORBA Event Service [6] introduces a standard object model consisting of an event channel that acts as a broker between anonymous event suppliers and consumers connected to the event channel via proxy objects. Likewise, the CORBA Notification Service [1] is an extension to the CORBA Event Service that provides (1) greater scalability via event filters, (2) simplified administration via standard event channel factory support, shared subscription information, and the ability to navigate the event channel object hierarchy, (3) resource management via QoS properties for reliability, event priority, and internal ordering/discarding of events, and (4) improved usability via support for three different types of events (CORBA anys, structured events, and batch events) and a constraint language (ETCL) for specifying filters.

Although the event-based publisher/subscriber CORBA middleware has been implemented and studied widely [7][8][9][10], the original CORBA Event and Notification Services have drawbacks that limit their applicability to applications with stringent QoS requirements, such as real-time deadline assurance [11][12]. For example, the existing CORBA publisher/subscriber services do not address the requirements of real-time event communications that requires timeliness and predictability when delivering events from suppliers to event consumers via event channels. Likewise, they do not use the priority and scheduling capabilities defined in the Real-time CORBA 1.0 [2] and 2.0 [3], respectively. Though Real-time CORBA provides end-to-end QoS support for point-to-point

communication via prioritized operation calls [13], it does not address end-to-end QoS guarantees for anonymous event-based communication. To overcome these limitations, therefore, the OMG has issued a Request for Proposals (RFP) [4] to provide enhancements that will support a Real-time Notification Service, which must satisfy the following requirements:

Limit the complexity of filters. As Section 2 describes, filters specify constraint expression strings that are evaluated against an event in an event channel. Events that do not match the constraints specified in the filter are discarded. A filtering expression can be arbitrarily complex. Submissions are required to provide mechanisms and describe the interfaces by which the filtering expressions could be minimized by limiting the number of different constraints in a filter, specifying the length of a message that might be successfully evaluated by a filter, and/or limiting the number of different event types that a supplier can produce. This requirement in the RFP enables a DRE application to bound the execution time spent in evaluating an event against a filter. Moreover, since the filtering process could be unbounded, submissions must define a subset of the filtering process that can bound the filtering process.

Subset functionality for real-time behavior. Submissions are required to document which portions of the Real-time Notification Service can be used for predictable resource management and timeliness predictability. The standard Notification Service has several features such as invoking administrative and factory methods that can affect real-time behavior of event delivery. By documenting these features, developers can learn how to avoid unnecessary latency and jitter in the critical path of timely event propagation.

Describe the schedulable entities. Submissions must describe the schedulable entities (such as event messages) that would participate in the operation of the Real-Time Notification Service. Moreover, both the Real-time CORBA 1.0 CLIENT_PROPAGATED and SERVER_DECLARED priority models must be supported.

Priority aware end-to-end event propagation. Real-time CORBA 1.0 describes the priority levels at which participant objects are executed. Submissions are required to support these priority levels so that when the events are propagated across hosts, the event priority is considered in the processing.

Provide means to set real-time QoS parameters. DRE applications may have QoS parameters (*e.g.*, priorities) that need to be communicated end-to-end. Submissions are required to provide mechanisms for this, along with a means to set real-time QoS parameters at the channel, connection, proxy, and message levels.

Interface for resource management. Real-time CORBA 1.0 provides interfaces to manage resources, such as the number of threads in a thread-pool. Similarly, submissions are therefore required to provide an interface that can help manage real-time notification resources, such as event channels.

Interface to support interactions with a Scheduling Service. Some real-time QoS requirements on a Notification Service might need to be checked via a global Scheduling Service. Submissions are required to define an appropriate interface to support interactions with the Real-time CORBA 1.0 Scheduling Service.

Since the Real-time Notification Service is still undergoing adoption via the OMG standardization process, no commercial products or research prototypes are available for it yet.. To facilitate the study of real-time publisher/subscriber services, therefore, we have implemented a prototype of the latest revised submission [2] that enhances on our prior work with The ACE ORB (TAO) [15] and its Real-time Event Service [11][16] and Notification Service [10]. Our new prototype implements the real-time extensions to the existing CORBA Notification Service designated in the revised submission to support predictable end-to-end event communications. This paper describes how we designed and optimized the performance of our Real-time Notification Service by (1) identifying sources of unbounded priority inversions, (2) using non-multiplexed resources to alleviate sources of unbounded priority inversions, and (3) ensuring predictable behavior for resources that must be shared. The results of our efforts have been integrated with the TAO open-source software release and are available from `http://deuce.doc.wustl.edu/Download.html`.

The remainder of this paper is organized as follows: Section 2 summarizes the key components in the OMG CORBA Notification Service, which provides the baseline for our work; Section 3 describes the design of TAO's Real-time Notification Service, focusing on how we resolved the key challenges faced in meeting the Real-time Notification RFP requirements outlined above; Section 4 analyzes the results of empirical benchmarks that illustrate the end-to-end predictability and scalability of our solution; Section 5 compares our work on TAO's Real-time Notification Service with related R&D activities; and Section 6 presents concluding remarks and describes future work.

2. Overview of the CORBA Notification Service

This Section describes the architecture of the core components in the standard CORBA Notification Service, which forms the basis for the Real-time Notification Service capabilities described in the remainder of this paper. The features of the Notification Service architecture are backwardly compatible to preserve interoperability with clients written for the CORBA Event Service. As shown in Figure 1, this architecture is similar to the architecture of the CORBA Event Service, though some components have a broader range of capabilities in the Notification Service. These components are as follows.

Structured events, which define a standard data structure into which event message can be stored. The Notification Service and its clients know the schema for structured events. Consumers can

install different filters that use the *filterable body* fields of the structured event definition to match with the filter constraint expressions efficiently. As shown in Figure 2, the header of a structured event consists of type information and a variable header, which can carry the QoS properties of an event. The event body consists of filterable body fields, followed by the payload data.

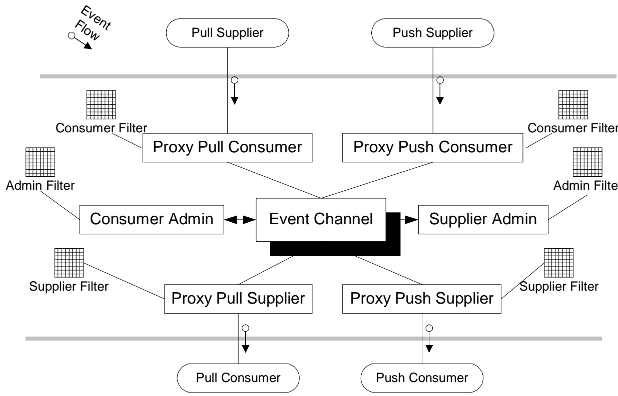


Figure 1: CORBA Notification Service Components

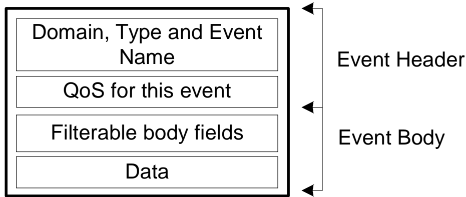


Figure 2: Header of the Structured Event

Proxy objects, which are delegates that provide complementary interfaces to clients, i.e., a consumer obtains and connects to a **ProxySupplier** and a supplier obtains and connects to a **ProxyConsumer**. Hence, a supplier sends events to its **ProxyConsumer**, whereas a consumer receives events from its **ProxySupplier**. This abstraction enables anonymous connectivity and communication between consumers and suppliers.

Admin objects, which are factories that create the proxy interfaces to which clients will connect. **ConsumerAdmins** create **ProxySuppliers** to which consumers connect. Conversely, **SupplierAdmins** create **ProxyConsumers** to which the suppliers connect.

The CORBA Notification Service treats each **Admin** object as the manager of the group of proxies it has created. **Admin** objects can themselves have QoS properties and **Filter** objects associated with them. The QoS properties associated with an **Admin** object are assigned to the **Proxy** objects that the **Admin** creates, but can be tailored subsequently on a per-proxy basis. The set of **Filter** objects associated with a given **Admin** is treated as a unit, which applies at all times to all **Proxy** objects that the **Admin** creates.

Filter and Mapping Filter objects, which can be associated with all **Admin** and **Proxy** objects and used to encapsulate a set of constraints that affect the event forwarding decisions made by **Proxy** objects. Each constraint consists of (1) a sequence of event types and (2) a string containing a boolean expression whose syntax conforms to a constraint grammar. The following is an example of a constraint expression:

```
$type_name ==‘CommunicationsAlarm’ and
$event_name == ‘lost_packet’ and
$priority < 2
```

The default constraint language defined by the Notification Service is the *Extended Trader Constraint Language (ETCL)*, which extends the TCL specified by the CORBA Trading Service [17]. The CORBA Notification Service introduces ***Mapping Filter*** objects, which are **Filter** objects that affect (1) the priority property of the events they receive and (2) the lifetime property of events they receive. **Mapping Filter** objects are associated with **ProxySupplier** objects.

EventChannels, which are factories that create **ConsumerAdmin** and **SupplierAdmin** objects. The **EventChannels** defined by the CORBA Notification Service differ slightly from the CORBA Event Service **EventChannels**, which only have one instance of each **Admin** object. QoS and administrative properties can be set on an **EventChannel** during its creation. These parameters are passed as default values to any **Admin** object created by the **EventChannel**. Consumers and suppliers can change these parameters later to tailor the properties for their specific requirements.

EventChannelFactory, which creates **EventChannels**.

Figure 3 shows the CORBA Notification Service interface hierarchy. The hierarchical object model shown in this figure is used to improve the administration of the Notification Service by applying the following design principles:

- Each object is created by another object factory,
- each parent factory assigns a unique identifier to the objects that it creates,
- each object maintains a back pointer to its parent, and
- parents maintain a list of children that can be queried.

This hierarchy allows any client of the Notification Service to discover all objects that comprise an event channel, starting with any object in the channel.

QoS properties. The CORBA Notification Service specification uses properties (represented as **<String, Any>** tuples) to define QoS properties. QoS properties can be associated with an **EventChannel**, **Admin** objects, **ProxySuppliers**, and **ProxyConsumers**, and individual event messages. The Notification Service specifies get/set methods for these properties. By setting these properties, the behavior of the Notification Service is modified as per the value of the property set. The properties defined by the Notification Service specification include:

- *Reliability*, which specifies the event reliability and connection reliability fault tolerance properties to the Notification Service. If these properties are enabled then when a Notification Service is restarted after a crash, it must reconnect to all its clients and deliver all events that have not expired yet to its consumers.
- *Priority*, which controls the order in which events are delivered to consumers. An event channel will deliver messages to consumers in priority order.
- *Expiration times*, which indicates the time range in which an event is valid. If an event is not delivered within a specified time then an event channel should discard it. The *StoptTime* specifies the absolute expiration time while the *Timeout* specifies the relative expiration time.
- *Earliest delivery time (StartTime)*, which specifies until when an event must be held in the channel before it is delivered.
- *MaximumEventsPerConsumer*, which bounds the maximum number of events the channel will queue on behalf of a given consumer to help avoid the case when the channel fills up its queues with events destined for a misbehaving consumer.
- *Order Policy*, which specifies the order in which events are queued for delivery, *e.g.*, by priority, arrival, or deadline.
- *Discard Policy*, which specifies policies for discarding events when queues are full.
- *MaximumBatchSize*, which specifies the maximum size of each batch of events to be delivered to a consumer receiving sequences of structured events.
- *PacingInterval*, which specifies the maximum period of time an event channel will collect individual events into a sequence before delivering the sequence to the consumer.

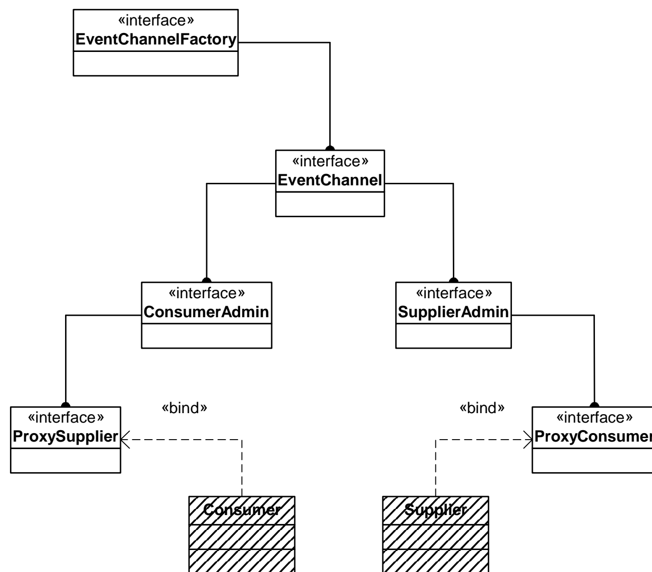


Figure 3: Notification Service Interface Hierarchy

The QoS properties supported by TAO's Notification Service are summarized in Table 1.

Table 1: TAO's Notification Service QoS Property Support

QoS Property	Support
EventReliability, ConnectionReliability	Best Effort
Priority	Supported Per Message
StartTime, StopTime	Not Supported
Timeout	Supported Per Message
MaxEventsPerConsumer	Supported
OrderPolicy, DiscardPolicy	PriorityOrder,AnyOrder, FifoOrder, LifoOrder, DeadlineOrder
MaxEventsPerConsumer	Supported
MaximumBatchSize	Supported
PacingInterval	Supported

Administrative properties. Certain properties can be set only on event channels. These administrative properties (all of which are supported by TAO's Notification Service) are:

- *MaxQueueLength*, which specifies the maximum number of events that will be queued by an event channel before the channel begins discarding events or rejecting new events upon receipt of each new event.
- *MaxConsumers*, which specifies the maximum number of consumers that can be connected to an event channel at any given time.
- *MaxSuppliers*, which specifies the maximum number of suppliers that can be connected to an event channel at any given time.
- *RejectNewEvents,* which specifies if new events entering an event channel should be rejected or if an existing event discarded as per the Discard Policy.

3. Design of TAO's Real-time Notification Service

This section describes the design of TAO's Real-time Notification Service, whose key components are shown in Figure 4. These components and their capabilities are as follows:

- **EventChannel component**, which is the CORBA interface presented to supplier and consumer participants. The **EventChannelFactory**, **EventChannel**, **ConsumerAdmin**, **SupplierAdmin**, **ProxyConsumer** and **ProxySupplier** (for CORBA any, structured, and sequence events) interfaces in the Notification Service give access to this component.
- **Subscription lookup component,** which consists of the subscription map of consumers to event types and the publication map of suppliers to event types. The subscription map is consulted to obtain the list of consumers subscribed to an event. This component also provides subscription/publication update messages to participants when the respective map is modified.
- **Filtering component**, which provides the filtering support to check if an event matches the filtering constraints in its

path of event propagation within an event channel. It provides interface support for the **Filter** and **FilterAdmin** interfaces of the Notification Service. The OMG standard ETCL filtering language is supported via a parser component.

- **Dispatching component**, which consists of a buffer queue that provides support for the *Order* and *Discarding Policy* QoS properties. Event buffering is not used by the real-time configuration of TAO's Notification Service, however, because the Real-Time CORBA 1.0 specification lacks the necessary means to control the buffer. The **Dispatching** component offers the *Pacing* and *BatchSize* properties to support event batching in sequence consumers.

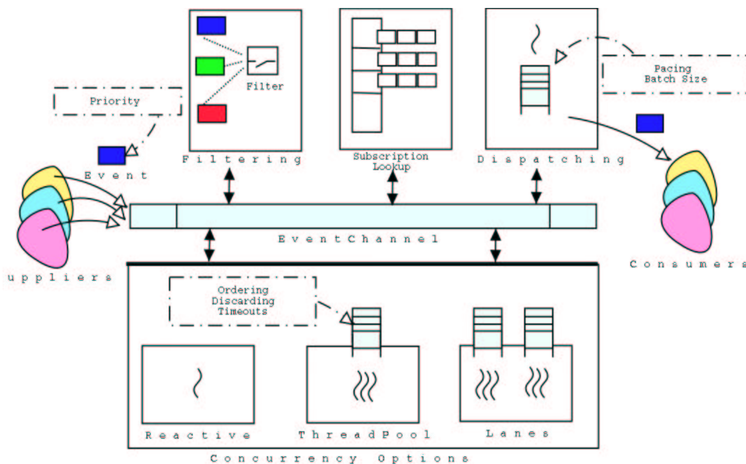


Figure 4: TAO's Real-time Notification Service

TAO's Real-time Notification Service can also be configured with one of concurrency models to process events. In the **Reactive** concurrency model a single thread is assigned to perform all actions, *i.e.*, event push invocations, filtering, subscription lookup, and dispatching of the event to subscribed consumers. In the **ThreadPool/Lane** concurrency model incoming events are processed by Real-time CORBA thread pools.

In general, the enhancements provided in TAO's Real-time Notification Service support:

- **Priority-aware, end-to-end event propagation.** By leveraging RTCORBA 1.0 features, event priorities are maintained and respected along the entire path of event propagation from event supplier to event consumers, which provides DRE applications with an end-to-end, priority aware publisher/subscriber service.
- **Administration of Concurrency options.** Extensions to the **Proxy** interfaces provide support for the configuration of concurrency within the Real-time Notification Service. **ThreadPool/Lanes** parameters can be applied at the event channel, admin, and proxy levels of the Real-time Notification Service object hierarchy.

The remainder of this section describes how we resolved the following challenges faced when meeting the Real-time Notification Service requirements described in Section 1:

- Limiting filter complexity
- Ensuring end-to-end priority preservation
- Supporting real-time thread pools
- Optimizing Event Processing
- Minimizing context switching between supplier and consumer proxies, and
- Minimizing unwanted jitter in timeliness of event delivery.

For each challenge, we describe the context in which the challenge arises, identify the specific problem that must be addressed, describe our solution for resolving the challenge, and explain how this solution was applied to TAO's Real-time Notification Service. Section 4 then presents the analysis of empirical benchmarks that illustrate the end-to-end predictability and scalability of our solutions.

3.1 Limiting Complexity of Filters

Context: Filter objects can be applied at the proxy and admin levels of the Notification Service hierarchy. The length of the constraints specified in a filter is unbounded. Moreover, filters can be changed dynamically and a filter can be a remote object. The RFP seeks to limit the complexity of such filters so that DRE applications do not incur unbounded filter evaluation overhead. The goal is to allow an application developer to set a useful bound on the time needed to evaluate a filter for an event.

Problem: Without bounds on the number of filters or the complexity of evaluating each filter, filter processing itself could consume an excessive and unpredictable amount of time, leading to deadline failures for delivery and processing of notifications.

Solution: Timeouts. Timeouts can be used to ensure that filters don't take too long to run, thereby ensuring that other deadlines aren't violated. The CORBA messaging extensions provide a `RelativeRoundtripTimeoutPolicy` that can be applied to a two-way invocation to specify how much time is allowed to deliver a request and its reply.

Applying the solution to TAO's Real-time Notification Service. TAO Real-time Notification Service provides two different mechanisms for applying timeouts to ensure filter processing times are bounded. First, a supplier can map an event deadline to a timeout QoS parameter of a structured event. TAO's Real-time Notification Service can then use the `RelativeRoundtripTimeoutPolicy` from the CORBA `Messaging` interface when making an invocation to a filter object where the timeout value equals the time left for the event to reach its destination. Second, the **Filter** interface can be extended such that the `Filter::match()` operation accepts timeouts, *i.e.*:

```
interface TimeoutFilter : Filter {
 boolean match_with_timeout (in any
filterable_data, TimeBase::UtcT timeout)
    raises  (UnsupportedFilterableData);
```

```
  boolean match_structured_with_timeout (
        in CosNotification::StructuredEvent data,
        TimeBase::UtcT timeout)
    raises (UnsupportedFilterableData);
};
```

This alternative pushes responsibility for implementing timeouts to the filter object developer.

3.2 Ensuring End-to-end Priority Preservation

Context: When an event enters a Real-time Notification Service event channel, it carries a priority. The event propagation mechanism in the event channel must ensure that the priority at which the event is processed is maintained consistently and correctly as the event traverses the path from supplier to consumer(s).

Problem: Maintaining a per-priority path from supplier to consumer(s) through the standard Notification Service preserves end-to-end priority. This priority preservation is only assured, however, across each instance of the Notification Service implementation, and is not enforced across the remote invocations between these instances. We therefore need to maintain per-priority paths *end-to-end*.

Solution: Real-time CORBA CLIENT_PROPAGATED priority model. Real-time CORBA 1.0 gives the CLIENT_PROPAGATED priority model that an application can use to convey the priority of a supplier thread to the thread processing the event in the Real-time Notification Service. Likewise, Real-time CORBA 1.0 *thread lanes* can be used to configure event paths based on priority.

Applying the solution to TAO's Real-time Notification Service: As shown in Figure 5, consumer and supplier proxies are activated in real-time POAs associated with Real-time CORBA thread pools. The priority of the supplier thread pushing events to the Real-time Notification Service will match the priority of the event sent by the supplier.

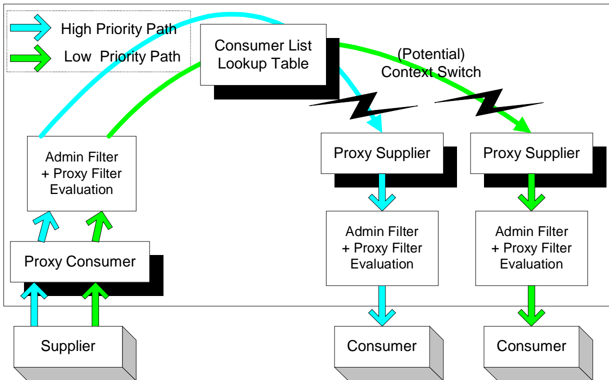


Figure 5: Prioritized Event Propagation Path

The proxy consumer in a thread of matching priority will service this event. After being filtered by the Real-time Notification Service, the event will be delivered to the consumer by the proxy supplier in a thread of matching priority. Finally, the consumer in a thread of matching priority will process the event. The proxy consumer executes the consumer admin and proxy level filters and queries a lookup table to retrieve a list of consumers subscribed to receive the event. The proxy supplier executes the supplier admin and proxy level filters and delivers the event to the consumer. A proxy supplier can be configured with its own thread pool.

3.3 Support for Real-time Thread Pools

Context: The Real-time Notification Service RFP [4] requires submissions to define schedulable entities and to support the CLIENT_PROPAGATED and SERVER_DECLARED priority models defined by Real-time CORBA 1.0 [2]. The Highlander Engineering response [18] to the RFP specified changes to the Notification Service, i.e., each factory method accepts the name of the POA in which the object should be activated. Specifying the target POA for objects allows applications to decide how operation invocations on a target object are processed, e.g., the configured thread pools and priority models. The OIS response [19] to the RFP proposes the structured event as a schedulable entity that is propagated via a distributable thread [3].

Problem: The OMG Notification Service does not specify a mechanism for specifying policies to the POA in which the proxy objects are activated. Since the Real-time CORBA 1.0 thread pool, thread pool with lanes, and priority model policies are specified on POA objects, a mechanism is needed to express these policies in the POAs for the proxy objects.

Solution: Use QoS properties to specify POA policies. Specifying new QoS properties for the Real-time Notification Service enables support for Real-time CORBA 1.0 features. These properties can be applied to POAs at multiple levels, i.e., event channel, admin, and proxy.

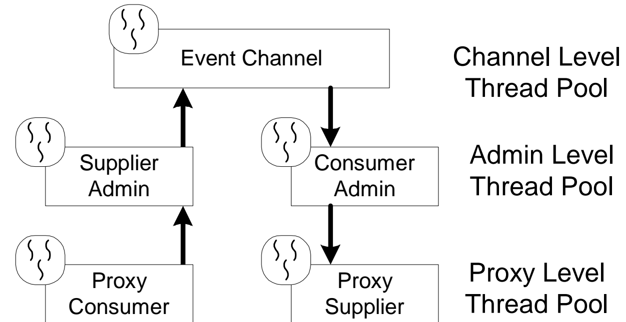


Figure 6: Real-time Notification Service Thread Pools

Figure 6 shows the thread pool policy applied to POA's that exist at these three levels. All admin and proxy objects share the event channel-level thread pool. The admin-level thread pool is only available to proxies that are created by that admin. The proxy-level thread pool is only available exclusively to the proxy with which the thread pool is associated.

Applying the solution to TAO's Real-time Notification Service: When a QoS property specifying the POA policy is set on the on a TAO Real-time Notification Service event channel,

admin or proxy a POA is created and the POA policies are applied to this new POA. This POA is used exclusively to activate the proxy objects.

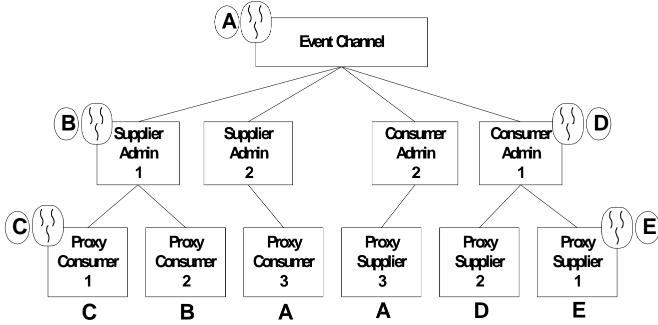


Figure 7: RT Notification Service Thread Pool Configurations

Figure 7 shows several possible thread pool configurations in the Notification Service. The consumer-side of the Real-time Notification Service is configured as follows: **ProxyConsumer** 1 is associated with **ThreadPool** C. **ThreadPool** C is used only by **ProxyConsumer** 1. **ProxyConsumer** 2 is associated with **ThreadPool** B. **ProxyConsumer** 2 does not have its own exclusive **ThreadPool**: it therefore uses the **ThreadPool** from **SupplierAdmin** 1. **ProxyConsumer** 3 is associated with **ThreadPool** A. **ProxyConsumer** 3 does not have its own exclusive **ThreadPool** and neither does **SupplierAdmin** 2. It therefore uses the **ThreadPool** from the **EventChannel**.

The supplier-side is configured similarly: **ProxySupplier** 1 is associated with **ThreadPool** E. **ThreadPool** E is exclusively used by **ProxySupplier** 1. **ProxySupplier** 2 is associated with **ThreadPool** D. **ProxySupplier** 2 does not have its own exclusive **ThreadPool**. It therefore uses the **ThreadPool** from **ConsumerAdmin** 1. **ProxySupplier** 3 is associated with **ThreadPool** A. **ProxySupplier** 3 does not have its own exclusive **ThreadPool** and either does **ConsumerAdmin** 2. It therefore uses the **ThreadPool** from the **EventChannel**.

Typically, an administrative application with knowledge of the overall deployment scenario of the Real-time Notification Service would specify QoS properties to control its thread resource usage at initialization. Similarly, the CLIENT_PROPAGATED and SERVER_DECLARED priority models can be specified by another QoS property. The Real-time CORBA 1.0 *thread pool with lanes* feature is also specified in the same manner.

3.4 Optimizing Event Processing

Context: During the processing of events within an event channel, the event type of the incoming event is matched in a *Subscription Lookup Table* to determine the set of consumers that are subscribed to receive that event. The lookup operation simply reads the table. Conversely, administrative methods write to this table to update subscription information.

Problem. Since multiple threads can access the *Subscription Lookup Table*, it is necessary to serialize access to it. A mutex lock causes a *read* operation to wait while other *read/write* operations are in progress. In a typical Real-time Notification Service deployment, however, the number of *push* operations is far greater than the subscription change operations. Using a simple mutex can therefore unnecessarily reduce concurrent access to data structures in the critical path of event propagation.

Solution: Readers/writer Lock. A readers/writer locks allow any number of threads to hold a lock for reading as long as no thread holds the lock for writing. A thread can hold the lock for writing only if no thread holds the lock for reading or writing.

Applying the solution to TAO's Real-time Notification Service. The *Event Map* data structure used to maintain subscription information in the *Subscription Lookup Table* has a strategized locking policy that uses a readers/writer lock to give preferred access to the event propagation threads.

3.5 Minimizing Context Switching Between Consumer to Supplier Proxies

Context: Event propagation may need to switch from a proxy consumer thread pool to a proxy supplier thread pool if the two proxies are in different thread pools. Such a configuration may be required to assign a dedicated thread pool to dispatch events to a consumer.

Problem: If the proxy consumer and proxy supplier are activated in separate POA's, there are no interfaces that can be used to transfer the thread of execution from a thread in the proxy consumer to the proxy supplier.

Solution: Use an internal interface to exploit the ORB's collocation mechanism. We exploit the fact that when a CORBA interface operation is invoked, the thread pool configured at the POA in which the interface is activated processes the request. As the **ProxySupplier** interface does not support any operation to accept an event, it must be extended so that the **ProxyConsumer** can forward the event to the extended interface.

Applying the solution to TAO's Real-time Notification Service: We define an internal interface, **Event_Forwarder** that extends the **Structured ProxyPushSupplier** interface. The **Event_Forwarder** interface supports the operation signature **void forward (in CosNotification::StructuredEvent event)**. The **ProxySupplier** interface implements the **Event_Forwarder** interface. When a proxy consumer needs to switch execution context to the proxy supplier, it simply calls the **Event_Forwarder::forward()** operation, passing it the event. The ORB collocation mechanism ensures that the execution context switches to the thread configured in the proxy supplier. If the proxy supplier is configured in the same thread pool as the proxy consumer, however, no context switch occurs.

3.6 Minimizing Unwanted Jitter in the Timeliness of Event Delivery

Context: The Real-time Notification Service RFP [4] requires submissions to identify the subset of the functionality of the standard Notification Service that can be used while achieving the goals of predictable resource management and timeliness. The purpose is to constrain those features of the Real-time Notification Service that can affect predictability.

Problem: The Notification Service has a number of administrative interfaces and operations that create **Proxy** and **Admin** objects, perform subscription updates, and traverse the Notification Service object hierarchy. Invoking these operations when event propagation is in progress can incur jitter in the timeliness of event delivery since those administrative operations may compete for key resources like CPU cycles.

Solution: Execute administrative operations at the lowest priority. All administrative operations are executed at the lowest available priority to ensure they run only when event propagation is not in progress. Note that OS-level thread scheduling policies will determine when the lowest priority thread is ultimately scheduled. This solution therefore emphasizes the predictability of event propagation over the predictability of executing administrative operations.

Applying the solution to TAO's Real-time Notification Service: We create a worker thread and set its priority to the lowest priority available. This worker thread runs the **ORB::run()** operation. When an administrative operation is invoked, the worker thread handles the request. When an event is pushed, however, the **push()** operation is invoked in a higher priority thread because all **Proxy** objects are activated in a Real-time POA, as described in Section 3.2.

4. Empirical Analysis of End-to-end Predictability

This section describes experiments we conducted to validate TAO's Real-time Notification Service prototype described in Section 3. We first describe the testbed environment and experimental benchmarks used for our experiments. We then present and analyze the results of experiments conducted to compare the throughput of TAO's Real-time Notification Service implementation with TAO's standard Notification Service implementation as the load is increased and the number of supplier-to-consumer paths is increased. We also assess the overhead of TAO's Real-Time Notification Service implementation by comparing its latency and jitter to that of (1) the standard Notification Service, (2) a direct Real-Time CORBA 1.0 priority connection, and (3) a standard CORBA connection that does not use Real-time CORBA priority features.

4.1 Testbed Environment and Benchmarks

All the experiments were conducted in the Emulab [20] testbed at the University of Utah (http://www.emulab.net). The results presented in the following subsections were obtained on 1 PC with all suppliers, all consumers, and the Notification Service each in separate processes. Similar results were observed by running each supplier and consumer on a separate PC. Each PC was a 800 MHz Pentium3 processor, with 256KB on-chip cache and 239MB of free RAM memory, running RedHat Linux 7.1 in the real-time scheduling class. The benchmarking programs were compiled using the GCC compiler version 2.96, with all optimizations enabled. TAO version 1.3.4 was used for all tests. We used this testbed to conduct experiments that measured load vs. throughput, the number of paths vs. throughput, and the overhead of various Notification Service implementations, including the Real-time Notification Service (RT-Notification) and the standard Notification Service (CosNotification).

4.2 Load vs. Throughput

Overview. In this experiment, an exclusive high priority *path* is set up between a supplier/consumer pair, as shown in the Figure 8. To achieve this, the consumer subscribes to a type that is supplied only by the high-priority supplier. The proxy consumer for the high-priority supplier is activated in a real-time POA with 1 high-priority thread lane. The supplier uses the CLIENT_PROPAGATED priority model to send events to the consumer. Likewise, medium-priority and low-priority paths are also established. Each supplier sends events every 10 ms, i.e., at 100 Hz. The experiment was performed using both the RT-Notification and CosNotification services by increasing the amount of CPU intensive work performed by each consumer and measuring the throughput obtained. The "load" is a positive count that is supplied with the event payload. A prime number calculation is performed proportional to the supplied load value.

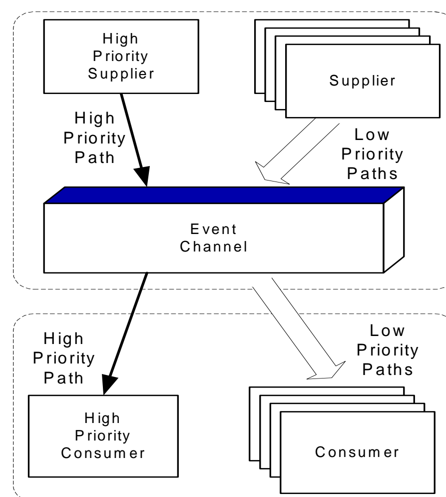


Figure 8 : Exclusive High-priority Path Setup

Results. Figure 9 shows that with CosNotification, the throughput of the high-, medium-, and low-priority paths decreased proportionally with increasing load.

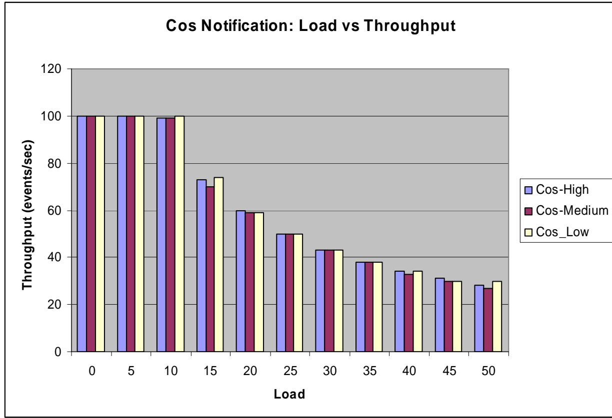


Figure 9: Cos Notification: Load vs Throughput

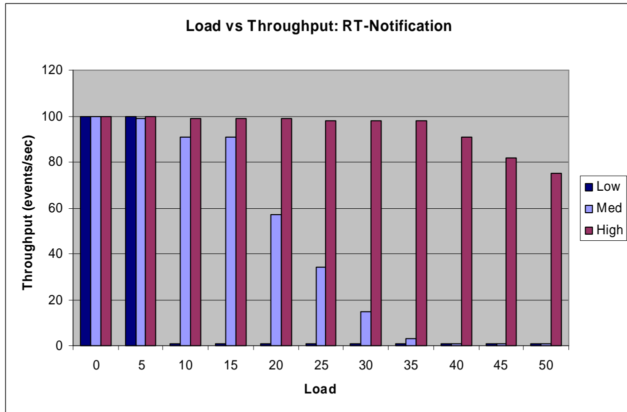


Figure 10: RT Notification: Load vs Throughput

Figure 10 shows that with RT-Notification, under increasing load the throughputs of the high-, medium-, and low-priority paths are maintained preferentially according to their specified priorities, i.e., low falls off first, followed by medium, and then high. These results indicate how RT-Notification is more effective at enforcing end-to-end priorities than CosNotification.

Latency analysis. Figures 11 and 12 show the latency graph of the experiment run at Load 25. The latency of the high-priority path using RT-Notification is about 1/8th of the latency obtained when using CosNotification. Note that in Figure 11, the latency of the low- and medium-priority paths overlap and have a bimodal distribution, which is possibly due to the real-time thread scheduling characteristics of the RedHat 7.1 OS.

Jitter analysis. Figure 13 shows the standard deviation of the latency measured for the high-priority path as load is increased. Note that the jitter obtained with CosNotification increases linearly with the load. In contrast, the jitter for RT-Notification remains low and fairly constant. These results show that the throughput of the high-priority path is maintained while there is available capacity in the system. The latency of high-priority invocations is the lowest among the competing paths.

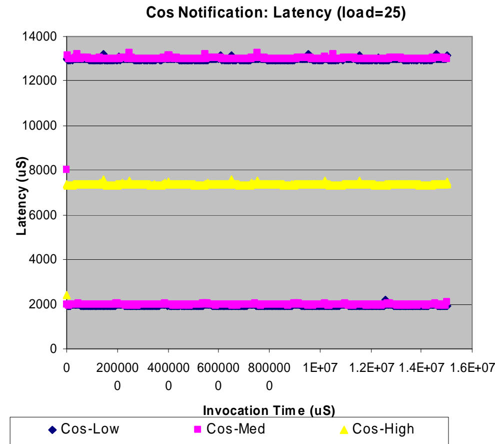


Figure 11 – CosNotification Latency

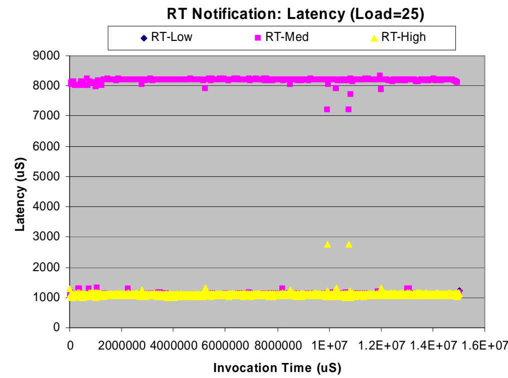


Figure 12 – RT-Notification Latency

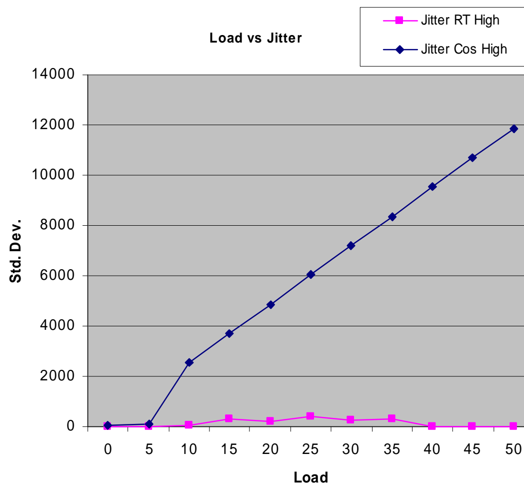


Figure 13 – Load vs Jitter

4.3 Number of Paths vs. Throughput

Overview. This benchmark measures the effects of increasing the number of low-priority paths, similar to the one described in Section 4.2. In this case, however, the load is held constant at 30 units and the number of low-priority paths is increased. All low-priority tasks are at the same priority. We ran the experiment with 1, 5, 10, and 20 low-priority paths.

Results. Figure 14 shows that the throughput of the high priority path was maintained consistently at 100 events/sec, despite the increase in the number of low-priority paths. This result shows that TAO's Real-time Notification Service protects the performance of higher priority paths, even as resource demands of lower-priority paths are increased.

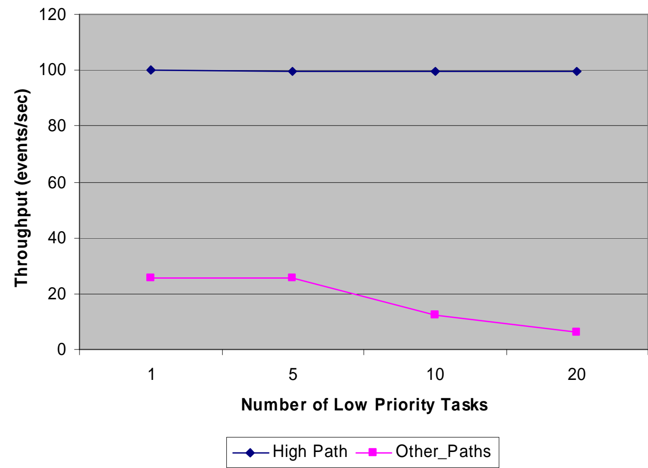


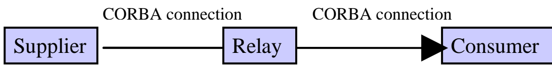
Figure 14 : Increasing the Number of Paths.

4.4 Overhead of RT- Notification

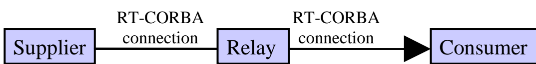
Overview. This experiment measured the maximum throughput obtained between a single supplier and consumer path for the following cases where the payload is a structured event that contains 64 bytes of payload representing the time at which the event was sent.

- 2-Hops: In this configuration a supplier sends an event directly to a consumer without an event channel.

- 2-Hops-RT: This configuration modifies the 2-Hops test to use Real-time CORBA - the consumer is activated in an real-time POA with a single lane.



- 3-Hops: In this configuration, the supplier sends events to a *relay* consumer, which forwards the event to the final consumer. Hence, the relay consumer behaves as a minimal event channel.



- 3-Hops-RT: The 3-Hops configuration is modified to use Real-time CORBA. The relay consumer and final consumer are activated in an real-time POA with a single lane.

- CosNotify: This configuration is the standard CosNotification with 1 supplier and 1 consumer.

- RT-Notify: This configuration uses the RT-Notification Service.

Results. Figure 15 shows that the RT-Notification does not add significant overhead to the CosNotification Service. This overhead that is observed is due to the additional processing performed, per request, by the ORB due to the extra Real-time CORBA information in the IIOP messages.

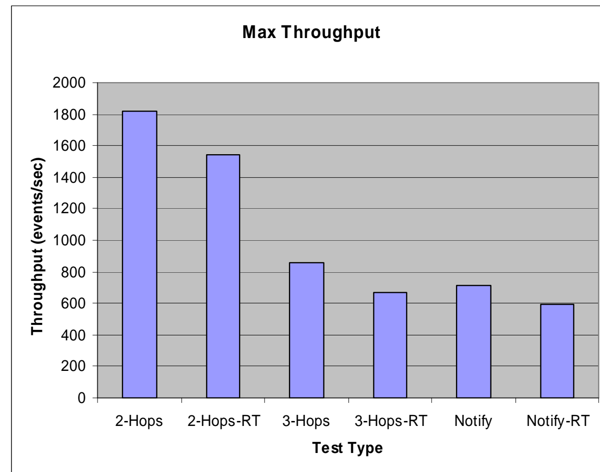


Figure 15: Max Throughput for Various Configurations

5. Related Work

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

The OMG has specified a Notification Service [1], which is a superset of the CORBA Event Service [6] that adds interfaces for

(1) event filtering, which allow applications to control which supplier events are disseminated to which consumers and (2) QoS properties, such as reliability, priority, ordering, and timeliness, which allow applications to bound resource consumption of the Notification Service. It also enables applications to specify the ordering of events in an event channel, thereby allowing events to be propagated with priorities and deadline criteria, rather than the strict FIFO ordering of the standard CORBA Event Service.

TAO's Real-time Event Service [11][16] extends the CORBA Event Service to support periodic rate-based event processing, efficient source and type based event filtering and event correlation. It uses UDP/IP multicast to federate multiple event channels and conserve network resources. The patterns and techniques used in TAO's Real-time Event Service have been applied and extended to improve the performance and predictability of TAO's Notification Service implementation, as described in Section 3.

The Highlander Engineering response [18] to the Real-Time Notification Service RFP [4] proposed removing support for **Mapping Filter** objects, which allow the Notification Service to modify the properties of the events, and instead proposed a Real-time Default Filter Constraint Language that is a subset of the Extended Trader Constraint Language. The Highlander Engineering RFP response also proposed removal of the following features in its proposal: Typed Events, Sequence Event Types, Mapping Filters, and the **validate_qos()** and **validate_event_qos()** operations. QoS properties that are removed include Reliability, Expiry Times, Earliest Delivery times, Maximum Batch Size and Pacing Interval. These features are removed to reduce storage requirements and improve predictability by removing features that could result in unbounded execution times.

The OIS response [19] to the RFP does not propose any changes to the filtering facility provided in the Notification Service. As developers are aware of the performance implications of using filters, the use of filters is left as a design issue. It identifies that the addition/removal of filtering constraints and filter execution could affect predictability. These solutions do not, however, address the need to limit the execution time of a filter. The OIS RFP does not propose removing any features of the COS Notification Service.

The OMG Messaging specification [21] gives application developers control over several QoS parameters, such as one-way reliability and timeouts, and introduces type-safe asynchronous method invocation (AMI) models. Although the CORBA AMI specification solves many problems with the original CORBA invocation model, it does not address anonymous or single-point-to-multiple-point communication, which is a central contribution of the CORBA Real-time Notification Service. The Messaging specification can complement implementations of the CORBA Real-time Notification Service, however, e.g., it defines several levels of reliability for one-way calls that can improve decoupling of the clients, without the risk of losing messages. We have augmented TAO with the AMI features [22] defined by the Messaging specification, which complement our Real-time Notification Service implementation.

COBEA [8] is a CORBA-based event architecture service that generates parameterized events, which are published by a trading service. For scalability, clients must register their interest with the service, at which point an access control check is performed. Subsequently, whenever a matching event occurs, the client is notified. As with TAO's Real-time Notification Service, the authors propose a number of extensions to support event-filtering, correlation, user defined event types, security on event access and priority based event delivery. COBEA does not support end-to-end event priority preservation., administration of concurrency, however, nor does it support subscription change notifications, and QoS parameters viz. Timeouts, Ordering and Discarding – LIFO, Deadline, Event Batch, Pacing and MaximumQueueLength.

In [24] 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 Notification 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 Notification Service to provide higher degrees of reliability is possible, and we are pursuing this topic in our future work on Fault-tolerant CORBA [23] and DOORS [24].

6. Concluding Remarks

TAO's Real-time Notification Service provides end-to-end QoS support for anonymous event communication, improved timeliness and predictability in the transmission and delivery of events to event consumers via event channels, and integration with Real-Time CORBA features (particularly in the areas of configuration of priorities and scheduling). The empirical results of Section 4 show that our protoeyp exhibits priority preservation, low jitter, and low overhead.

TAO's Real-time Notification Service currently provides real-time distributed event communication for statically scheduled applications and does not address dynamic scheduling issues. Our future work will therefore integrate the dynamic scheduling features of the Kokyu scheduling framework [25] that have already been integrated with TAO's Real-Time Event Service [14] with TAO's Real-Time Notification Service. In addition, we will enhance TAO's Real-time CORBA Notification Service so that it conforms to the OMG Real-time Notification Service specification when it is finalized.

References

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

[2] Object Management Group, *Real-time CORBA Joint Revised Submission*, OMG Document orbos/99-02-12 ed., Mar 1999.

[3] Object Management Group, *Real-time CORBA 2.0: Dynamic Scheduling Joint Final Submission*, OMG Document orbos/2001-06-09, June 2001.

[4] Object Management Group, *Real-time Notification: Request For Proposals*, OMG Document: ORBOS/00-06-10, June, 2000.

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

[6] Object Management Group, *Event Service Specification Version 1.1*, OMG Document formal/01-03-01 edition, 2001.

[7] Y. Aahlad, B. Martin, M. Marathe, and C. Lee, *Asynchronous Notification Among Distributed Objects*, Proceedings of the 2nd onference on Object-Oriented Technologies and Systems, USENIX, Toronto, Canada, 1996.

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

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

[10] Pradeep Gore, Douglas C. Schmidt, Carlos O'Ryan, and Ron Cytron, *Designing and Optimizing a Scalable CORBA Notification Service*, Proceedings of the ACM SIGPLAN Workshop on Optimization of Middleware and Distributed Systems (OM 2001), Snowbird, Utah, June 18, 2001.

[11] Timothy H. Harrison, David L. Levine, and Douglas C. Schmidt, *The Design and Performance of a Real-time CORBA Event Service*, Proceedings of OOPSLA '97, pp. 184-199, ACM, Atlanta, GA, 1997.

[12] Carlos O'Ryan, Douglas C. Schmidt, and J. Russell Noseworthy, *Patterns and Performance of a CORBA Event Service for Large-scale Distributed Interactive Simulations*, International Journal of Computer Systems Science and Engineering, vol. 17, num. 2, CRL Publishing, 2002.

[13] Irfan Pyarali, Douglas C. Schmidt, and Ron Cytron, Techniques for Enhancing Real-time CORBA Quality of Service, the IEEE Proceedings Special Issue on Real-time Systems, co-editors Yann-Hang Lee and C. M. Krishna, Volume 91, Number 7, July 2003.

[14] Chris Gill, Douglas C. Schmidt, and Ron Cytron, *Multi-Paradigm Scheduling for Distributed Real-Time Embedded Computing*, IEEE Proceedings 91(1), Jan 2003.

[15] Douglas C. Schmidt, David L. Levine, and Sumedh Mungee, *The Design and Performance of Real-Time Object Request Brokers*, Computer Communications, vol. 21, num. 4, pp. 294--324, Elsevier, 1998.

[16] Douglas C. Schmidt and Carlos O'Ryan, *Patterns and Performance of Real-time Publisher/Subscriber Architectures*, Journal of Systems and Software, Special Issue on Software Architecture - Engineering Quality Attributes, Lars Lundberg and Jan Bosch (Ed), Elsevier, 2002.

[17] Object Management Group, *Trading Object Service Specification, 1.0 edition, 1997*

[18] Highlander Engineering Inc., *RT Notification, Initial Submission to Real-time Notification RFP*, 2002.

[19] Objective Interface Systems, Inc., *Real-time Notification Service, Initial Submission In Response to OMG RFP orbos/00-06-10*, 2002.

[20] *An Integrated Experimental Environment for Distributed Systems and Networks*, by White, Lepreau, Stoller, Ricci, Guruprasad, Newbold, Hibler, Barb, and Joglekar, appeared at OSDI 2002, December 2002.

[21] Object Management Group, *CORBA Messaging Specification*, OMG Document orbos/98-05-05 edition, Object Management Group, 1998.

[22] Alexander B. Arulanthu, Carlos O'Ryan, Douglas C. Schmidt, and Michael Kircher and Jeff Parsons, *The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging*, Proceedings of the Middleware 2000 Conference, ACM/IFIP, 2000.

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

[24] Balachandran Natarajan and Aniruddha Gokhale and Douglas~C. Schmidt and Shalini Yajnik, *DOORS: Towards High-performance Fault-Tolerant CORBA*, Proceedings of the 2nd International Symposium on Distributed Objects and Applications (DOA 2000), OMG, Antwerp, Belgium, 2000 .

[25] Christopher D. Gill, David L. Levine, and Douglas C. Schmidt, "The Design and Performance of a Real-Time CORBA Scheduling Service," *The International Journal of Time-Critical Computing Systems 20(2)*, Kluwer, March 2001.