The Design and Performance of TAO's Real-time Notification Service

Pradeep Gore
OOMWorks
Metuchen, NJ
pradeep@oomworks.com

Abstract

This paper describes the design and performance of TAO's Real-time Notification Service.

Keywords

Real-time, Quality of Service (QoS), CORBA, Event Service, Notification, Predictability, Jitter, Throughput.

1. Introduction

The CORBA Notification Service does not address the requirements of real-time event communications that requires timeliness and predictability in the transmission and delivery of events from suppliers to event consumers via Notification Service Event Channels. Moreover, the Notification Service does not make use of the RT-CORBA priority and scheduling capabilities defined in RT-CORBA 1.0 and RT-CORBA 2.0, respectively. Though RT-CORBA provides end-to-end QoS guarantees for direct client-server communication, it does not address end-to-end QoS guarantees for anonymous event communication. TAO's Real-time Notification Service is an extension of TAO's Notification Service, providing real-time distributed event communication for statically scheduled applications.

This rest of the paper is structured as follows:

- Section 2: Examines the motivation and goals
- Section 3: Describes the architecture and design
- Section 4: Evaluates end-to-end predictability through empirical analysis
- Appendix: Provides an overview of Notification Service

The OMG has issued an RFP[1] on Real-time Notification. The requirements from the RFP, analysis of proposed solutions in the RFC[2], RFC[3], and our proposed solutions are discussed throughout this paper.

2. Motivation and Goals

The following are the goals of TAO's Real-time Notification Service:

- Provide end-to-end predictable event communications
- Extend Notification interfaces to utilize RT-CORBA features
- Design and implement real-time extensions to existing Notification Service
- Evaluate critical path to identify sources of unbounded priority inversions
- Use non-multiplexed resources to alleviate sources of unbounded priority inversions where possible
- Ensure predictable behavior for resources that need to be shared
- Evaluate and optimize performance of existing Notification Service

TAO's Real-time Notification Service provides real-time distributed event communication for statically scheduled applications and does not address dynamic scheduling issues.

3. Architecture and Design

This section describes the architecture and design of TAO's Real-time Notification Service and shows how challenges to providing end-to-end QoS guarantees are addressed.

3.1 End-to-end Priority Preservation

When an event enters the event channel of the Real-time Notification Service, it carries a priority. The implementation of the event propagation must ensure that the priority at which the event is processed is maintained correctly as the event traverses the path from supplier to consumers.

Solution: Prioritized Event Propagation Path

Maintaining a per-priority path from suppliers to consumers through the Notification Service preserves end-to-end priority.

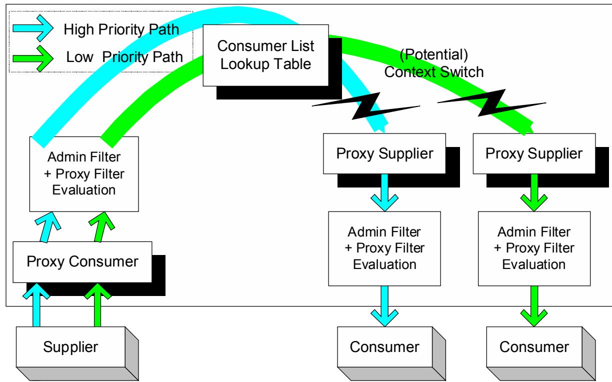

Figure 3.1: Prioritized event propagation path

As shown in Figure 3.1, the consumer and supplier proxies are activated in RT-POAs associated with thread-pools. The priority of the supplier thread pushing events to the Notification Service will match the priority of the event being propagated by the supplier. The proxy consumer in a thread of matching priority will service this event. After being filtered by the 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 which are subscribed to receive the event.

The proxy supplier executes the supplier admin and proxy level filters and delivers the event to the consumer. The proxy supplier can be configured with its own thread.

3.2 Support for Real-time Thread-Pools

RFP[1] requires submissions to define the schedulable entities in the Notification Service and address the client propagated and server declared priority models of RTCORBA 1.0[5].

RFC[2] specifies changes to the factory methods of the Notification Service. Each factory method accepts the name of the POA in which the object should be activated. Specifying the target POA for objects would allow applications to determine how method invocations on the target object would be processed (thread-pools, priority model, etc.).

RFC[3] proposes the structured event as a schedulable entity that is propagated via a distributable thread[6].

Solution: QoS Properties Specify POA Policies

RT-CORBA 1.0 features are supported by specifying new QoS properties for the Notification Service. These properties could be applied to POA's at 3 levels – channel, admin and proxy.

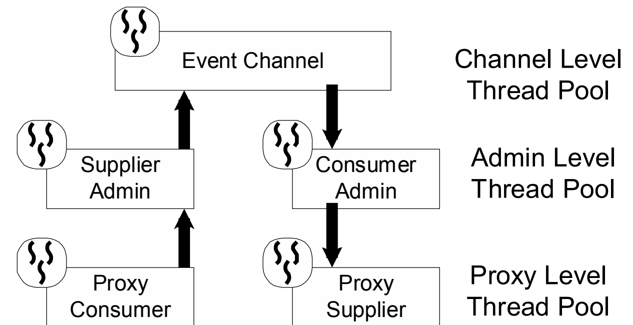

Figure 3.2: Levels for thread-pools

Figure 3.2 shows the thread-pool policy applied to POA's that exist at the 3 levels. All admin and proxy objects share the channel level thread-pool. The admin level threads-pool is only available to proxies that are created by that admin. The proxy level threads-pool is only available exclusively to the proxy with which the thread-pool is associated.

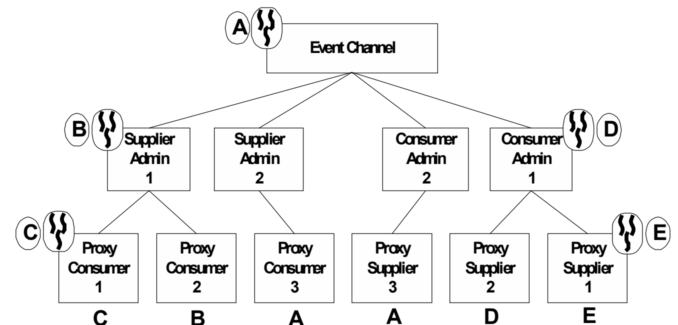

Figure 3.3: Possible thread-pool configurations

Figure 3.3 shows several possible thread-pool configurations in the Notification Service. The following is how the consumer side of the Notification Service configured:

- Proxy consumer 1 is associated with thread-pool C. Thread-pool C is exclusively used by proxy consumer 1.
- Proxy consumer 2 is associated with thread-pool B. Proxy consumer 2 does not have its own exclusive thread-pool. Therefore, it uses the thread-pool from supplier admin 1.
- Proxy consumer 3 is associated with thread-pool A. Proxy consumer 3 does not have its own exclusive thread-pool and either does supplier admin 2. Therefore, it uses the thread-pool from the event channel.

The following is how the supplier side of the Notification Service configured:

- Proxy supplier 1 is associated with thread-pool E. Thread-pool E is exclusively used by proxy supplier 1.
- Proxy supplier 2 is associated with thread-pool D. Proxy supplier 2 does not have its own exclusive thread-pool. Therefore, it uses the thread-pool from consumer admin 1.
- Proxy supplier 3 is associated with thread-pool A. Proxy supplier 3 does not have its own exclusive thread-pool and either does consumer admin 2. Therefore, it uses the thread-pool from the event channel.

Typically an administrative application with knowledge of the overall deployment scenario of the Notification would specify QoS properties to control the thread resource usage in the Notification Service at initialization.

The following IDL definition shows how the "ThreadPool" name and value pairs are specified:

```
module NotifyExt
{
  // Priority defs. same as RTCORBA
  typedef short Priority;
  const Priority minPriority = 0;
  const Priority maxPriority = 32767;

  /*
   * ThreadPool QoS property,
   */
  const string ThreadPool = "ThreadPool";

  // ThreadPoolParams: same as
  // RTCORBA::create_threadpool
  struct ThreadPoolParams
  {
    unsigned long stacksize;
    unsigned long static_threads;
    unsigned long dynamic_threads;
    Priority default_priority;
    boolean allow_request_buffering;
    unsigned long max_buffered_requests;
    unsigned long max_request_buffer_size;
  };
};
```

Similarly, the `CLIENT_PROPAGATED` and `SERVER_DECLARED` priority models can be specified by another QoS property.

RTCORBA 1.0 thread-pools with lanes are also specified in the same manner.

3.3 Avoiding Priority Inversion

The Real-time Notification Service should eliminate unbounded priority inversion and minimize bounded priority inversions.

Solution: Minimizing Shared Data Structures

Though mutex protocols, such as priority inheritance and priority ceiling, can be used to correctly arbitrate access to shared resources; it is desirable that sharing of resources that require mutual exclusion and do not exhibit constant or predictable behavior should be minimized.

The critical code path of TAO's Notification Service utilizes two shared data structures: the subscription lookup table and an internal representation of the event itself.

Threads perform lookups on the subscription lookup table to determine the consumers interested in an event. However, the lookup operation is *read only.* Hence a readers-writer lock is used to provide exclusion for this table. Note that if administrative interfaces enhancements (section 3.9) are used to suspend changes to the subscription table, mutual exclusion of the table will no longer be required.

All dispatching threads that deliver an event to their assigned consumers share an internal representation of the event. The last thread that dispatches the event is responsible for deleting the event. The internal representation of the event uses a thread mutex to protect the reference count on the event.

3.4 Context Switching from Proxy Consumer to Proxy Supplier

Event propagation may need to switch from the proxy consumer thread-pool to the proxy supplier thread-pool if the two proxies are in different thread pools.

Solution: Internal Interface

We define an internal interface, `Event_Forwarder` that extends the `CosNotifyChannelAdmin::StructuredProxyPushSupplier` interface. `Event_Forwarder` supports the following method signature:

```
void forward(
     in CosNotification::StructuredEvent event);
```

The proxy supplier 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` method, passing it the event. Collocation insures that the execution context

switches to the thread configured in the proxy supplier. However, if the proxy supplier is configured in the same thread-pool as the proxy consumer, no context switch occurs.

3.5 Supporting a Single Code Base

TAO's Real-time Notification Service extends the TAO CORBA Notification Service. It needs support one code base, as most functionality is common between the two. However, some CORBA interfaces need to be implemented twice in order to create a servant for the non-real-time and real-time implementations. Implementing the servant implementations twice should be avoided.

Solution: TIE classes

The TAO IDL compiler generates TIE classes that are template classes that use the bridge pattern to forward requests to another object that implements the actual functionality.

The `StructuredProxyPushSupplier` interface is implemented for both the non-real-time and real-time versions using TIE classes: The `TAO_NS_StructuredProxyPushSupplier` class implements the methods in the `CosNotifyChannelAdmin::StructuredProxyPushSupplier` interface.

The non-real-time implementation creates a `POA_CosNotifyChannelAdmin::StructuredProxyPushSupplier_tie <TAO_NS_StructuredProxyPushSupplier>` servant. The real-time implementation creates `POA_Notify_Internal::Event_Forwarder_tie<TAO_NS_StructuredProxyPushSupplier>` instead.

3.6 Minimizing Data Copies

As Notification payloads can be of any size, data copies should be minimized.

Solution: Reference Counting and caching locks

TAO's Notification Service implementation creates only one copy of the event and reference counts it. However, it uses short lived locks to guard the reference count value. By caching these locks, the cost of creating/destroying can be amortized.

3.7 Optimizing Critical Path Performance

The critical path of event propagation needs to be optimized for performance to reduce event latency and maximize throughput.

Solution: TBD

3.8 Limiting Complexity of Filters

Filters can be applied at the proxy and admin levels of the Notification hierarchy. The length of the constraints specified in a filter is unbounded. Moreover, the filters can be changed dynamically and a filter could be a remote object. The RFP seeks to limit the complexity of such filters so that real-time applications do not incur an overhead due to unbounded filter evaluation. The goal is to allow an application developer to set a useful bound on the execution time to evaluate an event against a filter.

RFC[2] proposes removing support for mapping filter objects. Mapping filter objects allow the Notification Service to modify the properties of the events. It proposes a Real-time Default Filter Constraint Language that is a subset of the Extended Trader Constraint Language.

RFC[3] 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.

Solution: Timeouts

The Notification Service supports timeouts on events. Events that have no utility after a certain time can specify a timeout value. In a real-time system, this is akin to a deadline. A supplier can map an event deadline to a timeout QoS parameter of a structured event. The Notification Service can then use the `Messaging::RelativeRoundtripTimeoutPolicy` when making an invocation to a filter object where the timeout value equals the time left for the event to reach it's destination.

Another way to achieve timeouts is to modify the `Filter::match` methods to accept timeouts, i.e.,

```
interface Filter
{
 // returns false if timeout occurs.
 boolean match (
        in any filterable_data,
        TimeBase::UtcT timeout)
    raises  (UnsupportedFilterableData);
 // returns false if timeout occurs.
 boolean match_structured (
        in CosNotification::StructuredEvent data,
        TimeBase::UtcT timeout)
    raises (UnsupportedFilterableData);
};
```

This pushes the responsibility of implementing timeouts to the filter object rather then the caller.

3.9 Functionality subset for Real-time Behavior

RFP[1] requires submissions to identify the subset of the functionality of the Notification that can be used while achieving the goals of predictable resource management, and timeliness predictability. The goal here is to constrain those features of Notification that might affect the predictability of timeliness. The Notification Service has a number of administrative interfaces and methods, e.g., creating proxy and admin objects, subscription updates, as well as methods to traverse the Notification Service object hierarchy. Invoking these methods when event propagation is in progress will introduce unwanted jitter in the timeliness of event delivery.

RFC[2] removes the following features in its proposal: Typed Events, Sequence Event Types, Mapping Filters, the `validate_qos` and `validate_event_qos` operations. QoS properties that are removed are: 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.

RFC[3] does not propose removing any features of the COS Notification Service.

Solution A: Suspend/Resume Administrative Functionality

Typically the administrative features are used when configuring the Notification Service at the start of a deployment. A new method is added that suspend all administrative methods.

```
interface EventChannel {
        void suspend_admin (void);
        void resume_admin (void);
        boolean admin_is_suspended (void);
}
```

Clients of Notification could check the `admin_is_suspended` method to determine if it is appropriate to make admin related invocations. Moreover, when the event channel suspends administrative features, all administrative methods would throw the `CORBA::NO_RESPONSE` system exception.

Solution B: Execute Administrative Methods at Lowest Priority.

This solution does not impose any interface changes but suggests an implementation in which all administrative methods are executed at the lowest available thread priority. This ensures that administrative method execution proceeds only when event propagation is not in progress. Note that OS thread scheduling policies will determine when the lowest priority thread is ultimately scheduled. Hence this solution emphasizes the predictability of event propagation over the predictability of executing administrative methods.

4. Empirical Analysis of End-to-end Predictability

TBD

5. References

[1] Object Management Group, RT Notification, Request For Proposal, OMG Document: ORBOS/00-06-10.

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

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

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

[5] Object Management Group, Real-time CORBA 1.0.

[6] 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.

6. Appendix: Structure and Functionality in the CORBA Notification Service

This section describes the structure of the core components in the CORBA Notification ServiceComponent Structure of the CORBA Notification Service

Figure 6.1 illustrates the components in the standard CORBA Notification Service.

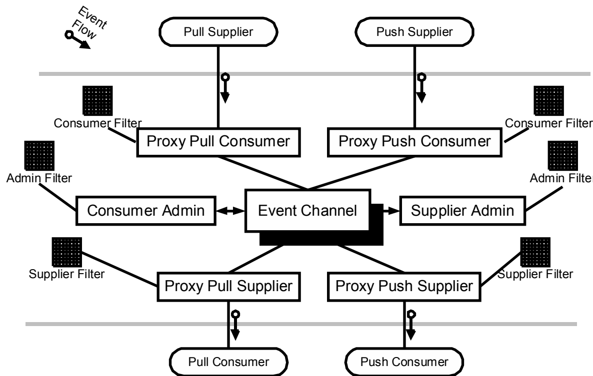

Figure 6.1: Components in the CORBA Notification Service

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. The enhancements in the Notification Service architecture are backwardly compatible to preserve interoperability with clients written for the CORBA Event Service. Each of these components is described below.

6.1.1 Structured Events

Structured events define a standard data structure into which a wide variety of event messages 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.

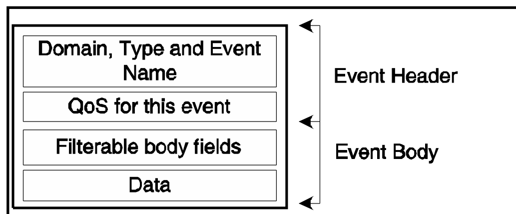

Figure 6.2: Header of the Structured Event

As shown in Figure 6.2, 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.

6.1.2 Proxy Objects

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

6.1.3 Admin Objects

Admin objects are factories that create the proxy interfaces to which clients will connect. Consumer admins create proxy suppliers to which consumers connect. Conversely, supplier admins create proxy consumers to which the suppliers connect.

The 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. Conversely, the set of filter objects associated with a given admin are treated as a unit, which apply at all times to all proxy objects that the admin creates.

Supporting multiple admin objects in a given event channel enables the logical grouping of the proxy objects according to common subscription information. This feature is particularly useful with respect to consumer admin objects, since it enables the channel to optimize the servicing of a group of consumers that are interested in receiving the same set of events.

6.1.4 Filter and Mapping Filter Objects

Filter objects can be associated with all admin and proxy objects. Filter objects encapsulate a set of constraints that affects 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 default constraint language defined by the Notification Service is Extended TCL, which extends the TCL (Trader Constraint Language) specified by the Trading Service.

To enable consumers to affect the priority and lifetime properties of events, the CORBA Notification Service introduces the concept of *mapping filter objects*. Each

proxy supplier within a Notification Service event channel can have associated with it mapping filter objects that affect (1) the priority property of the events it receives and (2) the lifetime property of the events it receives.

6.1.5 Event channels

An event channel is a factory that creates consumer admin and supplier admin objects. This differs slightly from the CORBA Event Service event channels, which only have one instance of admin objects. QoS and admin properties can be set on the event channel during its creation. These parameters are passed as default values to any admin object created by the channel. Consumers and suppliers can change these parameters subsequently.

6.1.6 Event channel factory

An event channel factory creates event channels. Figure 6.3 shows the Notification Service class hierarchy.

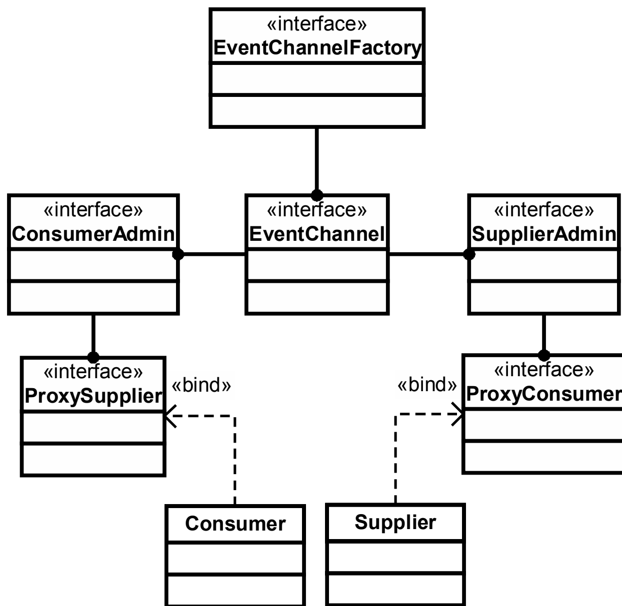

Figure 6.1: Notification Class Hierarchy

A hierarchical object model is introduced improve the administration of the Notification Service by applying the following design principle:

- 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;
- Parents maintain a list of children that can be queried for.

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

6.2 QoS properties

The specification uses properties, i.e. `<String, Any>` tuples, to define QoS properties. QoS properties can be associated with an event channel, admin objects, proxy suppliers and consumers, and individual event messages. The properties defined by the specification are:

- *Reliability* -- The event reliability and connection reliability specify fault tolerance properties to the Notification Service. If these properties are supported then after 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* -- This property controls the order in which events are delivered to consumers. The event channel will attempt to deliver messages to consumers in priority order.
- *Expiration times* -- This property 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.
- *Earliest delivery time* -- This property specifies how long an event must be held in the channel before it is delivered.
- *MaximumEventsPerConsumer* -- This property bounds the maximum number of events the channel will queue on behalf of a given consumer. This property helps avoid the case when the channel fills up its queues with events destined for a misbehaving consumer.
- *Order Policy* -- This property specifies the order in which events are buffered for delivery.
- *Discard Policy* -- This property specifies policies for discarding events when the queues are full.

6.3 Admin properties

The following administrative properties can be set on an event channel:

- *MaxQueueLength* -- This property specifies the maximum number of events that will be queued by the channel before the channel begins discarding events or rejecting new events upon receipt of each new event.

- *MaxConsumers* -- This property specifies the maximum number of consumers that can be connected to the channel at any given time.
- *MaxSuppliers* -- This property specifies the maximum number of suppliers that can be connected to the channel at any given time.