Designing and Optimizing a Scalable CORBA Notification Service

Pradeep Gore and Ron Cytron

{pradeep, cytron}@cs.wustl.edu
Department of Computer Science
Washington University, St.Louis

Douglas Schmidt and Carlos O'Ryan

{schmidt, coryan}@uci.edu
Electrical & Computer Engineering
University of California, Irvine*

Abstract

Many distributed 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 a publish/subscribe mechanism that is designed to support scalable event-driven communication by routing events efficiently between many suppliers and consumers, enforcing various QoS properties (such as reliability, priority, ordering, and timeliness), and filtering events at multiple points in a distributed system.

This paper provides several contributions to research on scalable notification services. First, we present the CORBA Notification Service architecture and illustrate how it addresses limitations with the earlier CORBA Event Service. Second, we explain how we addressed key design challenges faced when implementing the Notification Service in TAO, which is our high-performance, real-time ORB. Finally, we discuss the optimizations used to improve the scalability of TAO's Notification Service.

1 Introduction

Many distributed applications, such as real-time avionics mission computing systems, distributed interactive simulations, and computer-assisted stock trading, require an event-based communication model. The CORBA [1] Notification Service provides developers of these applications with a standards-based, QoS- and filtering-enabled event distribution mechanism. Our work on the Notification Service leverages the experience we gained developing TAO's Real-Time Event Service [2] to provide a flexible, extensible, and predictable implementation. In this paper, we explore the key design challenges faced in providing a *scalable* notification service. We also discuss optimization issues related to footprint reduction and configurability.

Limitations with client/server communication models: The most common invocation model for client/server communication in distributed object computing (DOC) middleware is based on synchronous method invocations (SMI). For example, CORBA, COM+, and Java RMI all support invocation models where a client invokes a two-way operation on a target object implemented by a server and then blocks waiting for the response. Although this invocation model is widely used, it has the following limitations:

- **Tight coupling of client and server lifetimes:** For a client request to be successful, the server must be available to process the request. If a request fails because the server is unavailable, the client receives an exception and must take some corrective action, such as notifying an end-user or system administrator. The CORBA Messaging specification [3] also introduced time-independent invocations (TII). Although the TII supports disconnected communication, it still requires clients to know the object references of their target objects.

- **Synchronous communication:** A client must wait synchronously until the server finishes processing the request and returns the result(s) to the client. Although CORBA now supports asynchronous method invocation (AMI) [4], this model still requires the server to be available when a client invokes a request.

- **Point-to-point communication:** A client invocation is typically destined for a single target object on a particular server. The CORBA Fault Tolerance [5] specification relaxes this point-to-point architecture, but is not widely implemented [6] or used at this time.

Potential Solution: The CORBA Event Service: The CORBA Event Service provides a decoupled communication model that addresses the limitations with the CORBA SMI (and AMI) invocation mechanisms outlined above. As shown in Figure 1, the Event Service defines three roles:

Figure 1: Components in the CORBA Event Service

- *Suppliers*, which produce event data;
- *Consumers*, which receive and process event data;

*This work was funded in part by ATD, Cisco, Siemens MED, and DARPA ITO

- *Event channels*, which are mediators [7] through which multiple consumers and suppliers communicate asynchronously.

Events are transferred via standard CORBA two-way requests[1] from suppliers to an event channel, which in turn forwards the events to consumers.

In general, the CORBA Event Service addresses the limitations of standard CORBA SMI and AMI invocation models, as follows:

• **Decoupling of event suppliers and event consumers:** By using an event channel, events can be delivered from suppliers to consumers without requiring these participants to know about each other explicitly.

• **Asynchronous communication:** By introducing the event channel between the suppliers and consumers, suppliers need not wait until an event is delivered to the consumers.

• **Transparent group communication:** Event channels can simplify application software by implementing group communication and serving as a replicator, broadcaster, or multicaster that forwards events from one or more suppliers to multiple consumers.

Limitations with the CORBA Event Service: Although the CORBA Event Service addresses many limitations with the standard SMI and AMI invocation models, the following requirements of distributed applications are not specified explicitly in the standard CORBA Event Service [2]:

• **QoS property support:** In real-time systems, events must be processed so that consumers can meet their QoS needs, such as reliability, priority, ordering, and timeliness. However, the CORBA Event Service provides no interfaces or policies for supporting these QoS properties. For example, there is no interface that consumers can use to specify their execution deadlines or other scheduling requirements.

• **Event filtering support:** By federating event channels, it is possible to create an event filtering graph that consumers register with to receive subsets of supplier events. However, this design increases the number of hops that an event must travel between suppliers and consumers, thereby degrading scalability. Thus, there is a need for centralized filtering to improve scalability. As shown in Figure 2, without the support for centralized filtering, an event must cross the network boundary before it can be rejected. With centralized filtering, each consumer can set filtering properties at setup time. Events not required by a consumer are rejected rapidly by an event channel.

Historically, implementations of the CORBA Event Service [2, 8] addressed these limitations by defining proprietary QoS properties and event filtering mechanisms. Proprietary extensions make it hard to build interoperable heterogeneous systems due to non-standard features. These extensions also tightly couple users to a particular implementation of an Event Service, which restricts the use of interchangeable solutions from other middleware providers.

[1] Some Event Service implementations use one-way requests, but this can cause flow control and reliability problems due to the semantics of CORBA one-way operations.

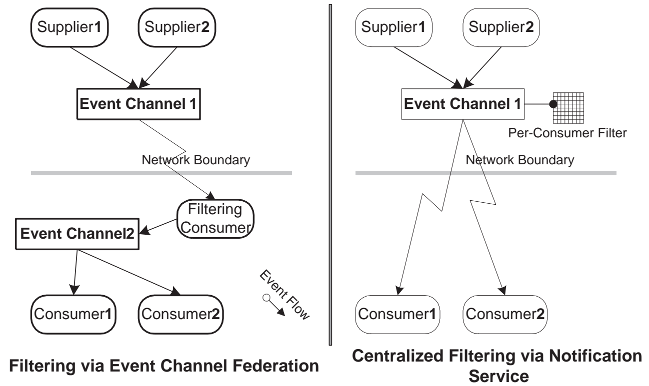

Figure 2: Alternative Event Filtering Architectures

Better Solution: The CORBA Notification Service: The CORBA Notification Service [9] was specified by the OMG to address the limitations with the CORBA Event Service outlined above. The CORBA Notification Service is a proper superset of the CORBA Event Service. Thus, in addition to providing all the interfaces and functionality specified in the Event Service, the CORBA Notification Service introduces the following capabilities:

• **Structured events:** Unlike the CORBA Event Service, whose `push` operations take generic `Any` data types, the structured events in the Notification Service can carry filtering and QoS parameters that influence the delivery of the payload data to its destination.

• **Filtering:** The use of structured events enables consumers to attach efficient filters to each proxy in a channel, thereby specifying which events they are interested in receiving. In addition, the Notification Service defines a constraint language that allows consumers to expressing arbitrarily complex filtering constraints.

• **Sharing subscription information between event channels and consumers:** This capability allows suppliers to determine the event types required by consumers of a channel. Thus, suppliers only must produce events for consumers that are interested in them. Likewise, consumers can determine when new event types are offered by suppliers so they can subscribe to new event types that become available.

• **QoS properties:** This capability allows suppliers, consumers, or administrators of an event channel to configure var-

ious QoS properties, such as reliability, priority, ordering, and timeliness, on a per-channel, per-proxy, or per-event basis.

Paper organization: The remainder of this paper is organized as follows: Section 2 describes the structure and functionality of the CORBA Notification Service in more depth; Section 3 describes the design challenges we addressed and optimizations we applied when implementing this service in TAO; Section 4 compares our work on the CORBA Notification and Event Services with related work; and Section 5 presents concluding remarks and lessons learned.

2 Structure and Functionality in the CORBA Notification Service

This section first describes the structure of the core components[2] in the CORBA Notification Service. We then outline the dynamic interactions in the CORBA Notification Service to illustrate how it operates in common use-cases.

2.1 Component Structure of the CORBA Notification Service

Figure 3 illustrates the components in the standard 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.

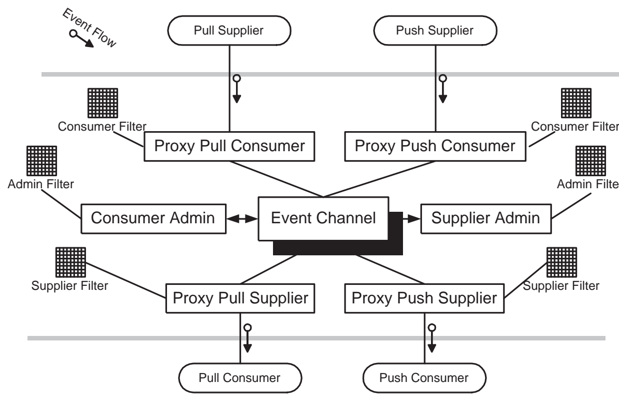

Figure 3: Components in the CORBA Notification Service

[2]The term *component* used throughout this paper refers to a "component" in the general sense, *i.e.*, an identifiable entity in a program, rather than in the more specific sense of the CORBA Component Model [10].

Structured Events: Structured events define a standard data structure into which a wide variety of event messages can be stored. The schema for structured events is known to the Notification Service and its clients. 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 4, 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.

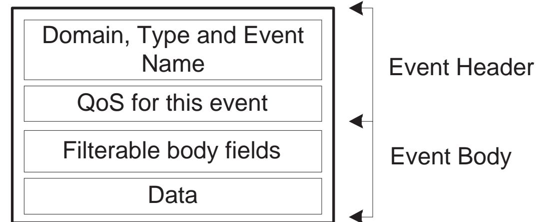

Figure 4: Structured Event

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.

Admin objects: Each admin object is a factory [7] that creates the proxy interface to which each client will ultimately 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 a given admin object are assigned to each proxy object when the admin object creates it, 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 have been created by an admin object.

Supporting multiple admin objects in a given event channel enables the logical grouping of the proxy objects associated with the channel 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.

Filter and Mapping Filter Objects: Filter objects can be associated with all admin and proxy objects. Filter objects that affect the event forwarding decisions made by proxy objects encapsulate a set of constraints. 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 [11].

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.

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. These parameters can be changed subsequently by consumers and suppliers.

Event channel factory: An event channel factory is a well defined interface for creating new instances of event channels. Figure 5 shows the Notification Service class hierarchy. A

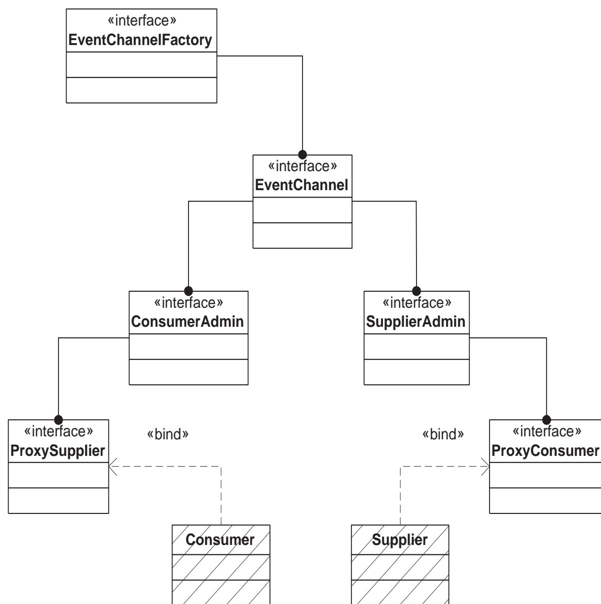

Figure 5: Notification Class Hierarchy

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.

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.

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.

2.2 Dynamic Interactions in the CORBA Notification Service

In order for consumers to receive events from a supplier via an event channel, they must each first connect to the channel, which requires the steps shown in Figure 6. These steps are

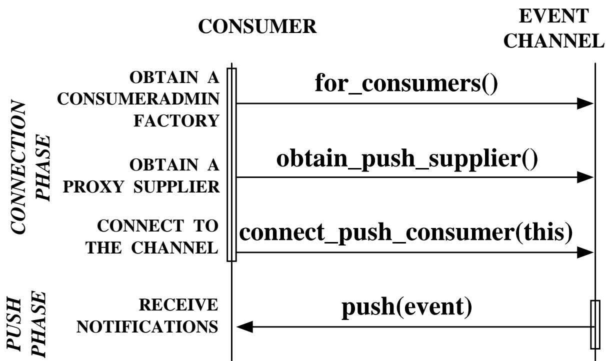

Figure 6: Connecting a Consumer to an Event Channel

explained below:

1. Obtain a ConsumerAdmin factory: Consumers that want to connect to an event channel must first invoke the event channel's `for_consumers` operation to obtain a `ConsumerAdmin` object reference. The `ConsumerAdmin` is a factory that returns object references to supplier proxies.

2. Obtain a proxy supplier: After a consumer calls the `for_consumers` operation to get an object reference to the `ConsumerAdmin` factory from the event channel, it must decide whether to be *passive* or *active* with respect to obtaining event notifications. The `obtain_push_supplier` operation is invoked by consumers that want to receive events passively from active `PushSuppliers` via a channel. This operation returns an object reference to a `ProxyPushSupplier`. Conversely, the `obtain_pull_supplier` operation is invoked by consumers that want to pull events actively from a `PullSupplier`.

3. Connect to an event channel: After consumers obtain the appropriate supplier proxy, they use the proxy to connect themselves to an event channel. At first glance, this "double dispatching" handshake between the consumer and the supplier proxies seems unnecessary and overly complex. However, the channel uses this bi-directional exchange of object references to keep track of its consumers and suppliers so it can disconnect them gracefully.

3 Designing and Optimizing TAO's Notification Service

This section describes how we implemented the CORBA Notification Service in TAO [12], which is a CORBA-compliant ORB that supports applications with stringent QoS requirements. TAO's Notification Service makes it easier to develop distributed applications in heterogeneous environments by providing application transparency, high flexibility, scalability, interoperability, bounded resource consumption, filtering of events, and FIFO/deadline/priority-based event delivery.

The following design challenges were identified prior to and during the development of TAO's Notification Service:

1. Handling multiple event, supplier and consumer types uniformly.
2. Efficiently propagating different event types to different types of consumers.
3. Minimizing interference between the event channel participants.
4. Ensuring fairness in event processing.
5. Optimizing the performance of the CORBA `Any` type.
6. Optimizations for footprint reduction.
7. Customizing event channels for particular deployment environments.
8. Optimizing event filter evaluation.

These challenges and our solutions are discussed below. To enhance the generality of our solutions, we describe them in terms of the patterns [7, 13] we used to resolve the design challenges.[3]

3.1 Challenge 1: Handling Multiple Event, Supplier and Consumer Types Uniformly

Context: Events transmitted between event channel participants can have different representation, such as `Anys`, structured events, and sequences of structured events.

[3]Note to OM'01 reviewers: the final version of this paper will contain the results of benchmarks that will demonstrate empirically the benefits of these design techniques and optimizations.

Problem: An event channel must propagate events from suppliers that feed its events in any form to consumers that want the event in any other form. When events are of the same type, however, we do not want to perform needless conversions to a canonical format, nor do we want to copy the event multiple times in memory. Similarly, there are different types of consumers and suppliers that can connect to the event channel. Since IDL interfaces are similar we do not want to write a specific implementation for each type with redundant code in each one of them.

Solution → the Adapter pattern: This pattern converts the interface of a class into another interface that clients expect [7]. The Adapter pattern lets classes work together that could not otherwise due to incompatible interfaces. This pattern relies on object composition, *i.e.*, the adaptee object is contained by the adapter object. The adapter also implements a target interface, which is the interface expected by a client class. The client deals with and invokes methods, only on the target interface. The adapter implementation of the target interface delegates operations to the adaptee. Hence, various adaptees can conform to the target interface and thus maintain a single interface to a client.

Applying the Adapter pattern in TAO: Figure 7 shows how TAO uses a base class to represent an event. The `Any_Event` and `Structured_Event`

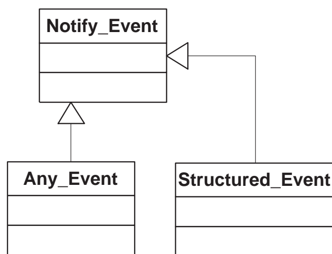

Figure 7: The `Notify_Event` Abstraction

classes (Adapters) "adapt" the `CORBA::Any` and `CosNotification::StructuredEvent` (Adaptees) to the `Notify_Event` interface (Target). These classes know how to manage memory and convert between the various types, *i.e.*, `Any` and `Structured`. This design results in a uniform treatment of events throughout TAO's Notification Service event channel implementation, thereby minimizing code duplication and allowing the integration of different strategies to process all types of events.

Figure 8 shows the classes TAO's Notification Service uses to represent proxy objects. The `Any_ProxyConsumer`, `Structured_ProxyConsumer` and `Sequence_ProxyConsumer` classes (Adapters) are used to adapt the `CosEvent::ProxyConsumer`,

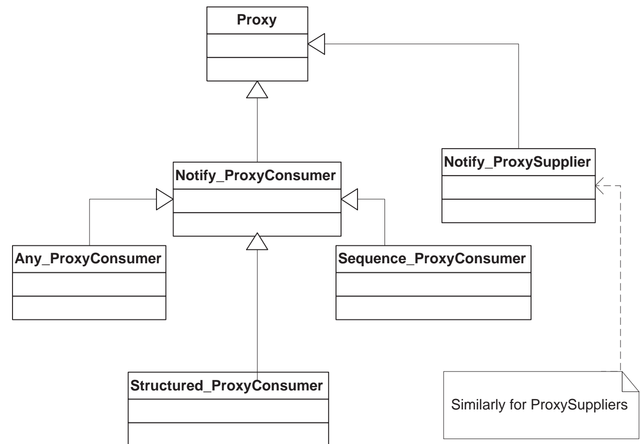

Figure 8: Abstraction for Proxy Objects

`CosNotifyAdmin::StructuredProxyConsumer`, and `CosNotifyAdmin::SequenceProxyConsumer` interfaces (Adaptees) to the `Notify_ProxyConsumer` (Target) interface. Similarly, the `ProxySupplier` classes are adapted to the `Notify_ProxySupplier` interface.

3.2 Challenge 2: Efficiently Propagating Different Event Types to Different Types of Consumers

Context: When an event is send from a supplier to the event channel, the receiving consumer(s) might not accept the same type of event, *e.g.*, an event send as an `Any` type could be received as an `Any`, a structured event, or a sequence of structured events.

Problem: We need to propagate different event types to different types of consumers efficiently.

Solution → the Visitor pattern: This pattern decouples operations that traverse elements in a complex object structure from the object structure itself [7]. The Visitor pattern lets us define a new operation without changing the classes of the elements on which it operates. When a visitor object calls the `accept` method of an element in an object structure, the implementation of the `accept` operation calls back the visitor object's `visit` method and passes information about its own concrete type.

Applying the Visitor pattern to TAO: Figure 9 illustrates how each consumer type implements the `dispatch_event` method, which in turn invokes the `push_event` method for consumer-specific event dispatching. When an event is selected to be dispatched to a consumer, the event processing engine invokes the `dispatch_event` method of the proxy

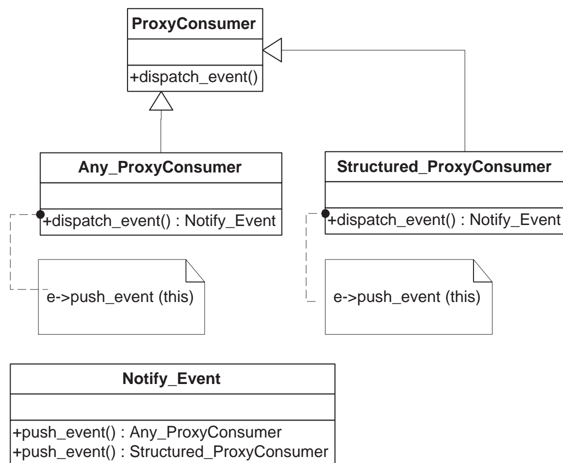

Figure 9: Double Dispatching for Event Propagation

consumer. In turn, the specific implementation of this virtual method invokes the correct `push_event` method of the event. This method then performs any necessary type conversion and initiates event dispatching to the remote consumer.

3.3 Challenge 3: Minimizing Interference Between Event Channel Participants

Context: An Event channel should be able to handle event delivery from suppliers and should be able to perform event forwarding with the minimum possible latency, *i.e.*, suppliers delivering an event to the channel should not have to wait while the channel forwards events to recipient consumers. Similarly when the event channel forwards events to multiple consumers, each consumer might spend an unbounded amount of time in the implementation of its `push` method.

Since events are forwarded by the event channel via a CORBA two-way method, the channel has no choice but to have the dispatching thread wait while the consumer returns from the call. Another case of such a coupling occurs when a method on a filter object is invoked to check if an event's properties match the filter constraints. A constraint could be arbitrarily complex and the filter itself could be a remote object. These factors can affect the event channel's event processing time.

Problem: A reasonable implementation should strive to minimize the interference between different participants of the event channel.

Solution → the Active Object pattern: This pattern decouples method execution from method invocation in order to simplify synchronized access to an object that resides in its own thread of control [13]. The Active Object pattern allows one or more independent threads of execution to interleave their access to data modeled as a single object.

Applying the Active Object pattern to TAO: Using the Active Object pattern at the various stages of event processing enables the minimization of the interference between the event channel participants. As shown in Figure 10, events and the operation performed on them are encapsulated as command objects. Enqueueing thread(s) place events into a command queue according to a buffering order policy. An Active object with worker threads dequeues and executes the command objects in the queue. Note that the ORB itself can be configured to increase concurrency by using the Leader/Followers pattern [13]. In this case, each ORB invocation is handled by a separate thread, allowing multiple events to be delivered concurrently to the event channel.

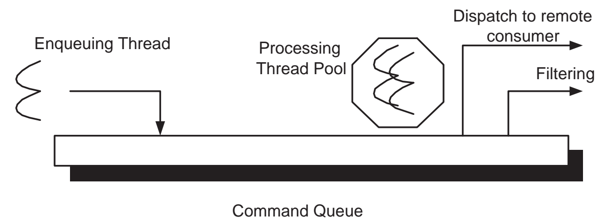

Figure 10: Asynchronous Event Processing Using the Active Object Pattern

3.4 Challenge 4: Ensuring Fairness in Event Processing

Context: The priority QoS property specifies the relative importance of an event. The Notification Service ensures that priorities are respected by enqueueing events in an internal buffer according to priority.

Problem: A long-duration filter evaluation operation involving a maximum of four remote filters (1 each for the Proxy Objects and 1 each at the Admin Objects) can starve other events in the queue and prevent them from being processed promptly.

Solution → the Command Object pattern: This pattern encapsulates a request as an object, thereby allowing parameterization of different requests [7]. The actual nature of the request is hidden by the Command Object interface. Different concrete implementation of the Command interface implement the request and provide semantics to it. This pattern can be used to decompose the internal event processing within an event channel into stages to ensure fairness.

Applying the solution in TAO: The filter evaluation, subscription lookup and event dispatching operations are encapsulated as command objects. As shown in Figure 11 instead

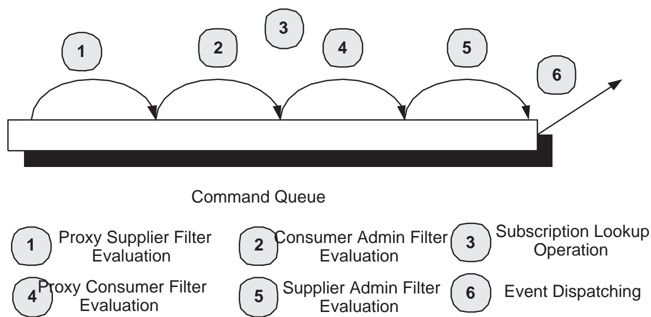

Figure 11: Processing Command Objects to Ensure Fairness

of performing these six operations synchronously, the evaluation is broken up into discrete operations. If the event is still eligible for further processing after executing an command operation, it is enqueued back into the command queue as the following command Object and subsequently dequeued for further processing. Finally, the dispatching command send out the event to the remote consumer.

3.5 Challenge 5: Optimizing the Performance of `Any`s

Context: A CORBA-compliant Notification Service must be able to process events containing `Any`s.

Problem: In CORBA, `Any`s are expensive data types because they can have many levels of nesting. For example, an `Any` can be contained within a structure, which can itself be contained within an `Any` and so forth. When a demarshaling engine decodes this expensive type from a common data representation (CDR) stream, it makes a copy of the entire data buffer used to represent the `Any`. Likewise, copying an `Any` can require several memory allocations and buffer copys to obtain a new representation of the CDR stream. Moreover, the C++ mapping of CORBA `Any`s requires them to be responsible for any memory returned to the application. Optimized ORBs should share the `Any` contents even if there are multiple copies of the `Any` object.

Solution → Reference counting via the Handle/Body idiom: This presents multiple logical copies of the same data while sharing the same physical copy [14]. In C++, this idiom is often used to automate the memory management in conjunction with reference counting and smart pointers.

Applying the solution in TAO: In TAO the CDR marshaling engine does not copy the CDR stream into the `Any`, instead, all CDR streams are reference counted, and the `Any`only increments the reference count to maintain a logical copy of the buffer. Likewise, once the contents of the `Any` are extracted by the application the `Any` object becomes responsible for deallocating the extracted object. This extracted object can be shared by multiple instances of the `CORBA::Any` object, minimizing the cost of copying and extracting the contents repeatedly. The use of the Handle/Body idiom implements this optimization without changing the semantics required by the standard C++ mapping.

3.6 Challenge 6: Optimizations for Footprint Reduction

Context: The Notification Service specification has many features that might be required by all applications, *e.g.*, some deeply embedded systems may not want to incur the increase in memory footprint for certain unneeded features.

Problem: A required set of Notification features should be "composable" by users.

Solution → the Builder pattern: This pattern separates the construction of a complex object from its representation so that the same construction process can create different representations [7].

Applying the solution in TAO: The Builder pattern uses the appropriate sets of libraries to compose a configuration required in a particular use-case. For example, a configuration containing *no-filtering* + *AnyProxySupplier* + *AnyProxyConsumer* + *reactive dispatching strategy* would yield the semantics of the CORBA Event Service.

These features are separated into libraries as follows:

- The three different pairs of proxy supplier and proxy consumer types are separated into different libraries. In a specific configuration, only the required type (*e.g.*, the push proxy supplier) may be loaded by a builder at startup. Hence, the other types of proxy implementations are not loaded since they are not needed.
- An application may not require filtering, in which case the filtering engine library is not loaded by the builder.
- The Dispatching Strategies could be simple reactive dispatching, or asynchronous dispatching as described in Section 3.2.

3.7 Challenge 7: Customizing Event Channels for Particular Deployment Environments

Context: The standard CORBA Notification Service is configurable in the following manner:

1. A user can specify features required by configuration.

2. A specific class implementation can be modified by the user to enhance or customize behavior.
3. Users can vary default properties, such as thread pool size and locking strategy.

Problem: A mechanism is needed to allow the application developers to use the various configurable option in the Notification Service.

Solution → the Component Configurator pattern: This pattern decouples the behavior of component services from the point in time at which service implementation are configured into an application [13].

Applying the solution in TAO: All objects in TAO's Notification Service implementation are created via factory objects. These factories can be loaded statically or dynamically by using the ACE framework [15]. Figure 12 shows a schematic of the how the Component Configurator is used in the Notification Service.

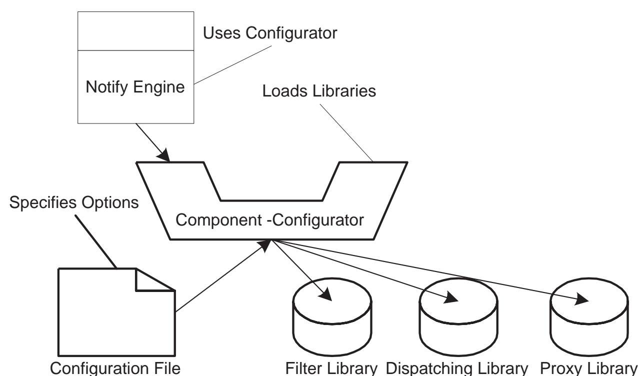

Figure 12: Apply the Component Configurator Pattern in the Notification Service

3.8 Challenge 8: Optimizing Filter Evaluation

Context: Admin objects in the Notification Service can be associated with filter objects. All proxy's connected to such an admin share the list of filter objects associated with the admin. Moreover, the proxys themselves are associated with filter objects. An event must satisfy the constraints specified by the filters at both the proxy and admin level.

Problem: After an admin level filter has been matched successfully against an event, we do not want to repeat this matching operation for each proxy connected to the same admin.

Solution → Optimization principle pattern of "passing hints": We use a variation of the optimization principle pattern "passing information between layers" [16]. This pattern is commonly used in protocol stack optimizations where each protocol layer passes certain information to the layer on top to help it avoid demultiplexing overhead.

Applying the solution in TAO: In TAO's Notification Service, we pass a hint to proxy objects to skip filter evaluation of their parent admin object if this has already been performed. We can optimize the filtering of a given event by a group of proxies since each member of the group logically applies the same filters to the same event. Thus, the results of the evaluation of a given event against a given filter can be shared by all proxy objects managed by a given admin object.

Figure 13 shows the configuration of a consumer admin (with filter `A`) of proxy supplier 1 (with filter `B`) and proxy supplier 2 (with filter `C`). An event must pass filters at both levels to be dispatched to consumers[4]. Filter `A` is evaluated first and the result of that evaluation is passed as a hint to both the proxy suppliers. This hint is used subsequently to determine if a proxy filter should be evaluated.

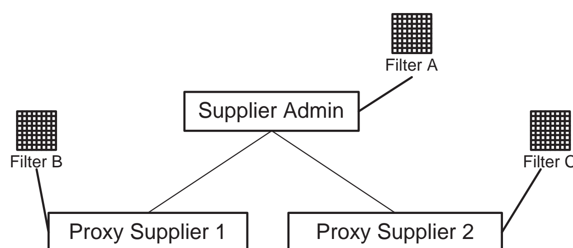

Figure 13: The Optimizing Filter Evaluation

4 Related Work

A number of research projects have focused on distributed publish/subscribe mechanisms. For example, Rajkumar *et al.*, describe a real-time publisher/subscriber prototype developed at CMU SEI [17]. Their Publisher/Subscriber model is functionally similar to the CORBA Event and Notification Services, though it uses real-time threads to prevent priority inversion within the communication framework. One 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, TAO's Real-time Event Service [2] utilizes QoS parameters from suppliers and consumers to guarantee the event

[4] An inter-filter group operator specifies if the results of evaluating the proxy and admin filters should be logically AND or OR

delivery semantics determined by a real-time scheduling service.

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 at the service, at which point an access control check is carried out. Subsequently, whenever a matching event occurs, the client is notified. This COBEA project is similar to the TAO Notification Service, the main difference being that TAO's Notification Service is based on the OMG standard.

There are several commercial CORBA-compliant Notification Service implementations available from vendors, such as Iona, Inprise, SunSoft [18], and DTSC. Iona also sells OrbixTalk, which is a messaging technology based on IP multicast. Unfortunately, since the CORBA Notification Service specification does not address issues critical for real-time applications, these implementations are not acceptable solutions for many domains.

5 Concluding Remarks

Many distributed applications, such as real-time avionics mission computing systems, distributed interactive simulation, and computer-assisted stock trading, require an event-based communication model. By using the Notification Service described in this paper, these applications can be built effectively by leveraging a middleware solution that is standards-based, flexible, and optimized for high-performance and scalability. The CORBA Notification Service builds upon the CORBA Event Service, which delivers events to all consumers connected to it on a best-effort basis. The Notification Service extends this service by providing the following two general mechanisms:

- *Event filtering*, which allow applications to control which supplier events are disseminated to which consumers. This selective control helps reduce redundant network traffic, *i.e.*, events that would be rejected by consumer applications after traversing the network are not sent in the first place.
- *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 implementation of the Notification Service addresses the issue of scalability and high-performance by applying suitable patterns and reusable framework components, optimizing the critical path of event propagation at the service- and ORB-levels, and providing configurability to reduce memory footprint and customize deployment.

All the source code, documentation, examples, and tests for TAO and its Notification Service and Real-time Event Service mechanisms are open-source and can be downloaded from `www.cs.wustl.edu/~schmidt/TAO.html`.

References

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

[2] 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.

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

[4] A. B. Arulanthu, C. O'Ryan, D. C. Schmidt, M. Kircher, and J. Parsons, "The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging," in *Proceedings of the Middleware 2000 Conference*, ACM/IFIP, Apr. 2000.

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

[6] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith, "Using Interceptors to Enhance CORBA," *IEEE Computer*, vol. 32, pp. 64–68, July 1999.

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

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

[10] BEA Systems, *et al.*, *CORBA Component Model Joint Revised Submission*. Object Management Group, OMG Document orbos/99-07-01 ed., July 1999.

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

[12] 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.

[13] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann, *Pattern-Oriented Software Architecture: Patterns for Concurrency and Distributed Objects, Volume 2*. New York, NY: Wiley & Sons, 2000.

[14] Jim Coplien, *Advanced C++ – Programming Styles and Idioms*. Addison-Wesley, 1992.

[15] D. C. Schmidt and T. Suda, "An Object-Oriented Framework for Dynamically Configuring Extensible Distributed Communication Systems," *IEE/BCS Distributed Systems Engineering Journal (Special Issue on Configurable Distributed Systems)*, vol. 2, pp. 280–293, December 1994.

[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* 5th *Conference on Object-Oriented Technologies and Systems*, (San Diego, CA), USENIX, May 1999.

[17] 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.

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