Object Interconnections

Overcoming Drawbacks in the OMG Events Service (Column 10)

Douglas C. Schmidt
schmidt@cs.wustl.edu
Department of Computer Science
Washington University, St. Louis, MO 63130

Steve Vinoski
vinoski@iona.com
IONA Technologies, Inc.
60 Aberdeen Ave., Cambridge, MA 02138

This column will appear in the June 1997 issue of the SIGS C++ Report magazine.

1 Introduction

Our last two columns have explored various techniques for using *distributed callbacks* to decouple clients and servers and create peer-to-peer relationships between the objects in a distributed system. We've shown various ways to eliminate the polling required by a stock application client. All these approaches center around direct or indirect callbacks from the Stock Quote Server. Like all engineering solutions, the designs and implementations we've shown have their strengths and weaknesses.

Our last column investigated how to use the OMG Events Service to implement a stock quote callback mechanism. The OMG Events Service is defined in Volume 1 of the OMG Common Object Services (COS) Specification [1]. The following event delivery models are supported by Event Channels:

• **Canonical Push model:** The active event Supplier (in our case, the Stock Quote Server) *pushes* events to the Event Channel, which in turn pushes them to passive event Consumers (in our case, interested stock quote client applications).

• **Canonical Pull model:** The active event Consumers *pull* events from the Event Channel, which in turn pulls them from the passive event Supplier.

• **Hybrid Push/Pull model:** The active event Supplier pushes events to the Event Channel, while the active event Consumers pull events from the Event Channel.

• **Hybrid Pull/Push model:** The Event Channel pulls events from the passive event Supplier and pushes them to passive event Consumers.

As explained in our last column, Event Channels support all these models because the OMG Events Service is intended as a specification for general-purpose event delivery systems.

2 Problems Galore!

Our original goal was to simplify the Stock Quote Server by making the Event Channel responsible for delivering stock value notifications to interested Consumers. While we achieved that particular goal, our plan backfired on us somewhat since the overall integrity and performance of our system were reduced. This turn of events (no pun intended) stemmed from the following problems with Event Channels:

• **Over-generalization:** Event Channels support different event delivery models that are useful for a wide range of applications. One consequence is that Supplier and Consumer registration is more complicated than is necessary for applications using just one delivery model. For instance, Consumers must know all the details of how to register themselves with an Event Channel.

• **Lack of persistence:** The COS Events Service standard doesn't require Event Channels to provide persistence. For instance, conforming Event Channel implementations need not store connectivity information and undelivered events when they shut down. This lack of persistence can significantly reduce the robustness of Event Channels, and in turn reduce their utility for distributed applications.

• **Lack of filtering:** The standard semantics of the COS Event Channel specifies that all events are delivered to all push Consumers. Therefore, each Consumer must filter the events to find the ones it's interested in. In contrast, the `Notifying_Quoter` implementation we described in previous columns only delivered events to Consumers that had explicitly subscribed for them.

• **Lack of correlation:** Some Consumers can execute whenever an event arrives from any Supplier. Other Consumers can execute only when an event arrives from a specific Supplier. Still other Consumers must postpone their execution until multiple events have arrived from a particular set of Suppliers (*i.e.*, they depend on a *correlation* of events). The standard COS Events Service does not address the event correlation needs of Consumers that can't execute until multiple events occur. As before, Consumers are responsible for performing correlations, which is very costly, as described in the following bullet.

• **Increased endsystem network load and Consumer load:** One consequence of delivering all events to all Consumers is that the network load may be higher than with designs that perform some or all of the event filtering and correlation in the Event Channel. Moreover, the workload on the Consumers will also increase since they must perform the filtering and correlation at the destination. This increased workload can be particularly problematic if Consumers run on low-end PCs.

• **Multiple suppliers:** An Event Channel can have multiple suppliers attached to it, thereby increasing the potential for more events in the system. As a result, this may further increase network load and require Consumers to perform even more filtering.

• **Lack of type-safety:** Untyped Event Channels deliver event data using the OMG IDL `any` type. This forces Consumers to perform additional work converting the `any` to a specific type so they can examine and manipulate the data.

3 Avoiding Common Traps and Pitfalls

We originally chose to use an Event Channel to separate the concerns of monitoring stock values from those of delivering notifications about changes in those values. In this column, we'll address each of the problems listed in the previous section to see what changes are needed so that we can use an Event Channel to simplify our Stock Quote Server. Along the way, we'll distinguish between solutions that require the following:

• **Changes to the COS Events Service specification:** For example, the current COS Events Service specification doesn't support event filtering or correlation. Although adding these features can significantly improve performance, it can be difficult to accomplish this in practice due to the long lead times required by the OMG standardization process. Fortunately, the OMG is already working on a new *Notification Service* [2] that will augment the existing Events Service to help address these concerns.

• **Changes to implementations of the COS Events Service specification:** The COS Events Service is intentionally vague, to avoid over-constraining the innovation and opportunity for optimization of implementors. Thus, there are a number of different ways to implement the COS Events Service. Certain implementation decisions make it easier to address the drawbacks we discuss in this article.

• **Changes to applications that use a COS Events Service implementation:** This solution is not always the most aesthetic or efficient. However, it's often the quickest and most pragmatic way to overcome common drawbacks with the existing COS Events Service specification and implementations.

The remainder of this section explains techniques for avoiding the common traps and pitfalls described in Section 1.

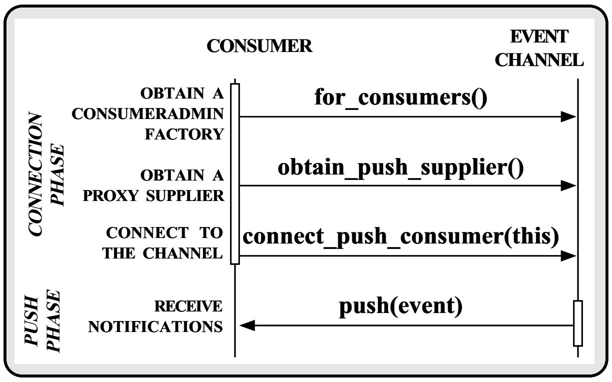

Figure 1: Connecting a Consumer to an Event Channel

3.1 Alleviating Over-generalization

Problem – complex registration process: The COS Events Service is very general, to the point where it is hard to apply for many common use-cases. This is particularly evident when trying to connect a Supplier or Consumer with an Event Channel. As we showed in our last column, registration with an Event Channel requires a "double dispatching" handshake between the Consumer and Supplier proxies. The Channel uses this bi-directional exchange of object references to keep track of its Consumers and Suppliers so it can disconnect them gracefully.

Both Consumers and Suppliers must make three separate operation invocations to register with a Channel. As shown in Figure 1, Consumers that want to register as a push consumer must first call the `for_consumers` operation on the Event Channel to obtain a `ConsumerAdmin` object reference. This object reference is then used to invoke the `obtain_push_supplier` operation to get a proxy from the Event Channel with which to register. Finally, the `ProxyPushSupplier` returned from the previous step is used to invoke the `connect_push_consumer` operation, passing it a reference to its `PushConsumer` object to receive the events.

Unfortunately, this handshake is more complicated than necessary for most applications.

Solution: The solution is obvious: to simplify the Event Channel registration protocol, we must hide it behind a simpler interface. For example, the registration interface we showed several columns ago for our `Notifying_Quoter` was much simpler:

```
// IDL
module Stock {
  // Requested stock does not exist.
  exception Invalid_Stock {};

  // Distributed callback information.
  module Callback {
    interface Handler {
      // ...
    };

    // ...
  };
```

```
  interface Notifying_Quoter {
    // Register a distributed callback handler
    // that is invoked when the given stock
    // reaches the desired threshold value.
    void register_callback
           (in string stock_name,
            in long threshold_value,
            in Callback::Handler handler)
      raises (Invalid_Stock);

    // Remove the handler.
    void unregister_callback
          (in Callback::Handler handler);
  };
};
```

To register using this `Notifying_Quoter` interface, a client simply calls the `register_callback` operation, passing a `Callback::Handler` object reference to be invoked when the named stock reaches the indicated value. This solution doesn't require any changes to the Events Service specification or vendor implementations. It just provides a wrapper around the Event Channel registration protocol that makes it much easier to use.

The Consumer registration protocol supported by our `Notifying_Quoter` is simpler than the one used by the Event Channel because it only supports the Canonical Push Model of event delivery. In particular, the Event Channel registration handshake required to select the delivery model isn't necessary for the `Notifying_Quoter`.

Keep in mind, however, that the `Notifying_Quoter::unregister_handler` operation can have problems of its own. Specifically, it relies on the `CORBA::Object::is_equivalent` operation to compare object references and ensure that the right one is unregistered. However, the semantics of this operation are too weak to allow it to be used for this purpose. The problem is that `is_equivalent` may return *false* even though the two object references identify the same remote object.[1]

3.2 Resolving Persistence Issues

Problem – loss of non-persistent data and connection information: Any time the Event Channel is shut down, or if it fails unexpectedly, non-persistent information can be lost. For instance, the Event Channel could lose information about the Consumers and Suppliers connected to it. Moreover, it could lose undelivered event data.

The italicized labels in Figure 2 depict potential sources of lost data and connection information in an Event Channel.

Solution: Saving and restoring Consumer and Supplier registration information isn't hard, assuming that the rate of connections and disconnections is not too high. Since object references can be changed into string form by the ORB, the Event Channel only needs to utilize a suitable persistent store in which to write stringified object references for Consumers and Suppliers as they register.

[1]Our September 1996 column [3] discusses the reason for these non-intuitive semantics in more detail.

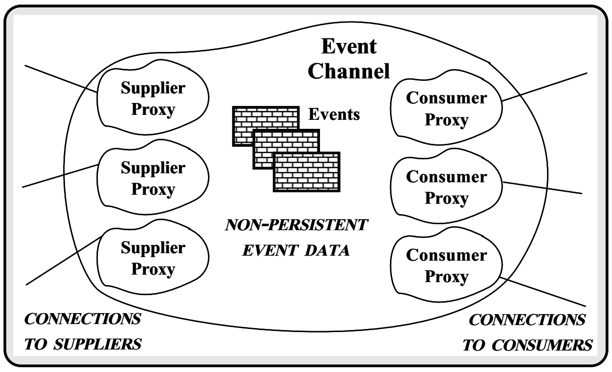

Figure 2: Failure Points in an Event Channel

Storing undelivered event data is more difficult, however. Typically the rate of event delivery is much higher than the rate of connections and disconnections. To be completely reliable, an Event Channel must store a copy of each event it receives until it successfully delivers it to all Consumers. In most network environments, this requires end-to-end acknowledgements between the Channel and all of its Consumers.

There are many protocols for ensuring reliable group communication. However, these protocols are non-trivial to implement. Moreover, they can reduce performance significantly compared with non-reliable group communication protocols (such as IP multicast).

Note that the functionality for ensuring Event Channel reliability must be provided by implementors. It's outside the scope of what end-users and Consumer/Supplier applications can accomplish since they don't program the internal details of an Event Channel.

Problem – storing CORBA anys: A related problem with the storage of event data is the fact that the data arrives at the Event Channel in the form of a CORBA `any`. CORBA `any`s are self-describing types capable of storing any built-in or user-defined OMG IDL type. Storing such types isn't too difficult if one can extract the compiled C++ form from the `CORBA::Any` type. It's not practical, however, to recompile the Event Channel every time a new user-defined event type is added to the distributed system.

Solution: What's required is a way to store instances of the `any` type regardless of whether the type is compiled into the Event Channel or not. Currently, the ability to store instances of the `any` type depends upon which ORB you use. Unfortunately, there is no standard way to decompose an instance of an `any` for storage to disk, though some ORB products support proprietary extensions to solve this problem.

Fortunately, this particular portability problem has already been recognized by the submitters to the OMG Portability Enhancement RFP. The solution will be included in their joint submission that should be completed by the time you read this column. A new IDL interface named `DynAny` will allow `any`s to be created dynamically. It will also allow

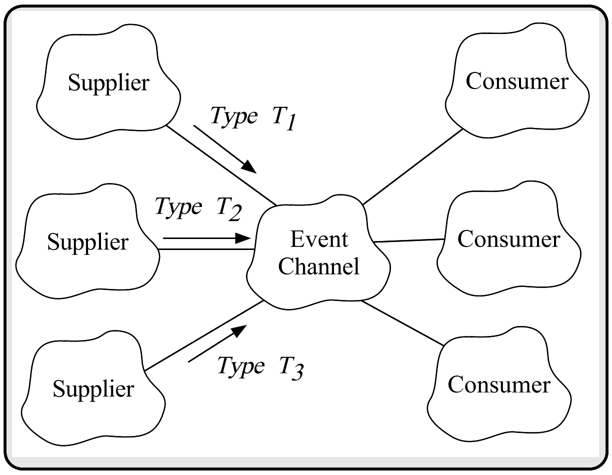

Figure 3: Multiple Suppliers Generating Multiple Types of Events

an instance of an `any` to be decomposed into its constituent built-in IDL types.

Once adopted as a standard, the `DynAny` interface will allow a portable program (such as an Event Channel and event filters) to manipulate instances of the `any` type regardless of what programming language it's written in. More importantly, `DynAny` will work regardless of whether the actual type stored in the `any` is statically known to the program or not.

3.3 Eliminating Multiple Suppliers

Problem – multiple suppliers with multiple type systems: Security implications aside, there's nothing to stop an application from acquiring an object reference to an Event Channel and connecting itself as a Supplier. This is problematic for the following reasons:

- *Increased Channel workload* – As more Suppliers connect to a Channel, there is a greater potential for the Channel to become a bottleneck as Suppliers push more events to the Channel.
- *Increased Consumer workload* – As more Suppliers push events through a Channel, the more events must be propagated to Consumers. Moreover, it's likely that new Suppliers will generate different types of events (as shown in Figure 3). It's possible that many of these types won't be of interest to all the Consumers, however.

Solution: One way to eliminate the problem of multiple Suppliers is to have an application create its own Event Channel and keep it hidden by not advertising its object reference. This prevents any other applications from connecting to it as a Supplier. For example, in our Quote Server example, we can employ the special registration interfaces described in Section 3.1 to ensure that unwanted Suppliers can't access the Event Channel directly.

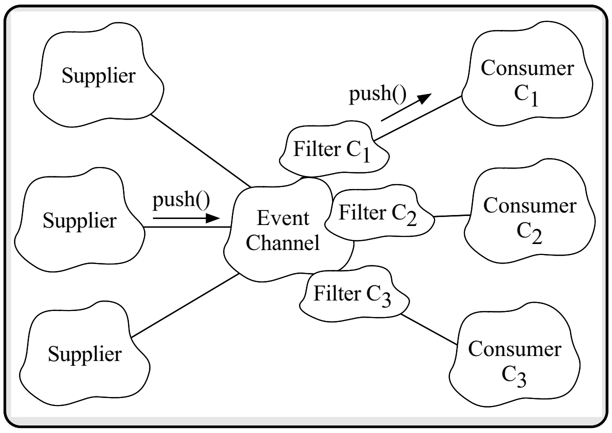

Figure 4: Co-locating Filters with an Event Channel

Note that this solution only requires changes to applications and doesn't require any changes to the Event Channel specification or implementations.

3.4 Performing Filtering and Correlation in Event Channels

Problem – filtering at the consumers: In a standard COS Events Service, each Consumer typically perform its own filtering. COS Event Channels can be chained to create an event filtering graph that allows Consumers to register for a subset of the total events in the system. However, this filter graph increases the number of hops that a message must take between Suppliers and Consumers. This increased overhead may be unacceptable for applications with low latency requirements. In addition, a general-purpose filtering mechanism that interprets `CORBA::Any` types may be too inefficient for high-performance applications.

Solution 1: One solution is to attach co-located filters directly to the Event Channel so that Consumers only receive events they're actually interested in. For instance, we've made the Event Channel private with respect to our Quote Server. Therefore, we can ensure that it only receives events that are relevant for it. Moreover, since the Event Channel is private, we *statically* know the IDL types flowing through it, so filtering is much easier and more efficient.

Note that this solution doesn't require any changes to the Event Channel specification, but it does require extra interfaces on the Event Channel implementation that allows filters to be installed directly within the Event Channel process. The following sketches how this functionality could be implemented by an Event Channel provider:

1. *Filter interposition* – Each Consumer registration causes the creation of a co-located filtering Consumer object that is interposed *locally* between the Event Channel and the actual Consumer (shown in Figure 4). The fact that the filter is located within the Event Channel server means that events are not needlessly transmitted over

the network[2] only to be thrown away, thus helping to decrease the network load.

2. *Event interception* – As events are pushed to the Event Channel from Suppliers they are intercepted and compared against the filtering object. If they match the filtering criteria they are forwarded to the Consumer. If not, they are discarded. Therefore, this mechanism ensures that only those of events interest to the real Consumer actually reaching that Consumer. For instance, Figure 4 illustrates a scenario where an event pushed by a Supplier to the Event Channel only passes the filter installed by Consumer C_1.

There are two drawbacks to this solution:

1. *Filter registration* – This solution requires Event Channel implementations to support special filter registration interfaces. Such interfaces are not yet standardized, so their signatures and their semantics would vary between Event Channel implementations.
2. *Filter implementation* – This solution begs the question of how filters could possibly be implemented. An obvious solution is to pass an object reference reference for each filter, which the Event Channel can invoke before pushing an event to a Consumer. The problem with this approach is that the filter object can't be passed by value to the Event Channel, and thus could not be located directly with the Channel. Therefore, the benefits of co-located filters could not be realized. Other possible implementations for filters are mentioned in Solution 2 below.

Solution 2: A potentially more efficient and scalable solution is to extend the COS Events Service specification to explicitly support event filtering. There are a number of techniques for accomplishing this, such as parallel processing of composite filters, trie-based filter composition, and context-free grammar-based filter composition using "skip-ahead parsing" [4]. It is very likely that the submissions to the OMG Notification Service RFP mentioned above will standardize one or more filtering solutions.

[5] describes a filtering mechanism for a real-time implementation of the COS Events Service. This implementation provides filtering and correlation mechanisms that allow consumers to specify logical OR and AND event dependencies. When those dependencies are met, the real-time Event Service dispatches all events that satisfy the Consumers' dependencies and timing requirements. The Appendix describes additional information on event filtering architectures.

3.5 Minimizing Network and Consumer Load

Problem – excessive load on the network and Consumers: If all events are delivered to all push Consumers, both the network load and the workload on the Consumers may increase. The increase in network load is obviously due to the need to deliver all events to all Consumers. The increase in Consumer workload is due to each Consumer having to perform event filtering and correlation. Increasing the workload of Consumers can be particularly problematic if they run on low-end machines.

Solution 1: Instead of just making the Event Channel private, as suggested above, it can also be created directly within the Quote Server. This eliminates one of the network hops (*i.e.,* Supplier to Channel). However, the resulting decrease in network traffic may be negligible since our Event Channel is private and the Quote Server is the only Supplier attached to it. Moreover, the additional workload of having the Event Channel in the same process as the Quote Server may actually decrease the overall performance of our server, unless careful multi-threading or asynchronous event processing is utilized.

Despite these potential drawbacks, a co-located Event Channel makes it much easier to implement co-located event filters. As described above, registering or creating co-located filters for a stand-alone Event Channel requires that it supports extra proprietary interfaces that go beyond the OMG Events Service Specification. If the Event Channel is local, however, co-located filters can simply be implemented as normal Consumer objects, and thus can registered with the Event Channel using the regular Consumer registration interfaces. Because such filters reside in the same process as the Event Channel, the benefits of co-located filtering are easily achieved without requiring the Event Channel to support a general filter interpreter as described above. From a programming perspective, this solution is desirable since it doesn't require any changes to the specification or implementation of existing Event Channels.

A reasonable tradeoff might be to run the Event Channel on the same system but not in the same process as the Quote Server, and use an ORB capable of communicating via shared memory. This keeps the Supplier-to-Channel message traffic off the network, but does not require an Event Channel implementation that can be linked into and run within another program. With this solution, filters could still be implemented as regular Consumer objects. Even though they would no longer be co-located within the Event Channel process, communication with the filters from the Event Channel via shared memory would still be quite efficient.

In any case, implementing a server with its own local Event Channel can be simplified greatly if shared library or DLL-based Event Channel implementations are available. Currently, only stand-alone server-based Event Channels are common.

Solution 2: Another solution is to use "batching." This approach is shown in Figure 5, where the Channel queues up groups of events destined to the same Consumer and delivers them *en masse*, rather than individually. This results in lower

[2] This assumes that the ORB performs "short-circuited" local dispatching (such as direct or near-direct function calls) for messages to objects in the same address space.

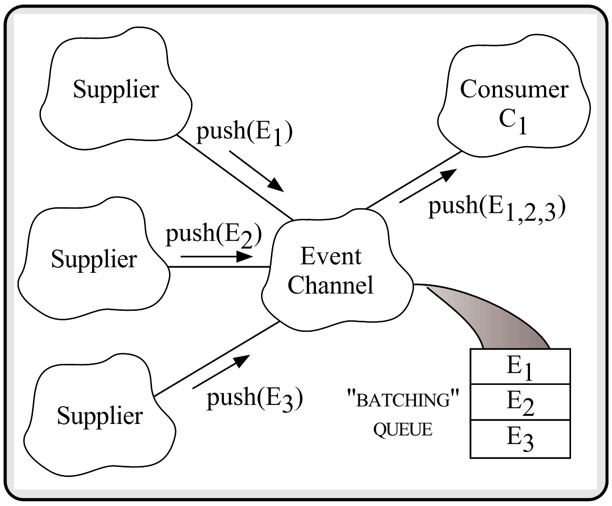

Figure 5: Batching Requests to Consumers

network utilization since the fixed costs (*e.g.,* interrupts, context switching, and protocol processing) of transmitting an event are amortized over a larger payload. The main drawback is an increase in latency due to delay incurred while batching up the events.

Batching can be implemented with minor changes to the Event Channel implementation and specification. Because the Event Channel for our Quote Server is private, we know that all events flowing through the system are actually the same IDL type. Since events are delivered in the form of a `CORBA::Any`, our Event Channel can either put a single event into an `any`, which is the norm, or can bend the rules slightly and actually store a *sequence* of events in the `any`. Pushing a sequence of events allows the Event Channel to deliver multiple events to a given Consumer with a single push operation. However, it requires that Consumers be aware that the `anys` they receive can hold either a single event or a sequence of events.

3.6 Improving Type Safety

Problem – `CORBA::Any` **can be error-prone:** With an untyped event channel, event data is delivered via the OMG IDL `any` type. An `any` is similar to the C/C++ `void` pointer in that it can contain the state of any OMG IDL type. It also shares some drawbacks with `void` pointers in that using an `any` can be error-prone.

Fortunately, a `CORBA::Any` keeps a TypeCode along with the data so that it is possible to detect type errors at run-time. However, applications written with `any` can be complex since each Consumer must be prepared to actively distinguish the events it understands from those it does not.

Solution: We can create our own event type system and use it via our private interfaces. As shown in Figure 6, this solution interposes a type-safe software layer that hides the insertion and extraction of event data into and out of `CORBA::Any`. The benefit of this approach is that the code

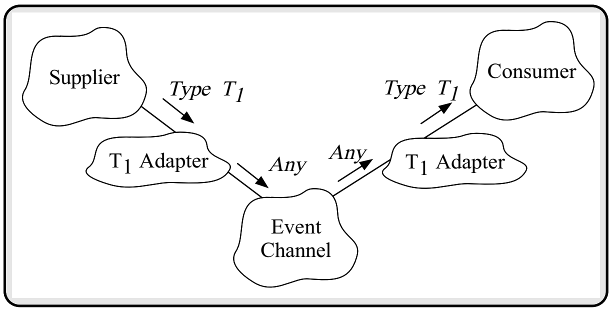

Figure 6: Interposing a Typed Interface over Untyped Events

to handle the `Anys` can isolate the handling of `Any` values from the rest of the application. In addition, this solution doesn't require any changes to the specification or implementation of the COS Events Service.

Thus far, we have focused solely on the *untyped* interfaces of the OMG Events Service Specification. However, the specification also describes how an Event Channel can support *typed* interfaces. In theory, using a typed Event Channel interface is essentially equivalent to a solution that involves wrapping an untyped Event Channel with private typed interfaces. In practice, the specification for typed Event Channels is vague and confusing.

To the best of our knowledge no ORB vendors support typed Event Channels. Until support for typed Event Channels becomes available, it's best to encapsulate the Event Channel with your own private C++ wrapper interfaces.

4 Conclusion

This column concludes our investigation of distributed callbacks and event delivery services – we hope you've found our exploration of these issues and design tradeoffs useful. Along the way, we've suggested various solutions to many drawbacks with the COS Events Service via a combination of application changes, Event Channel implementation enhancements, and proposed extensions to the COS Events Service specification. Not surprisingly, there are still many challenges awaiting those who use the COS Events Service in practice.

In our next column, we'll start presenting issues surrounding CORBA *Object Adapters*, which is where programming language object implementations meet the world of CORBA objects. In particular, we'll describe the new *Portable Object Adapter* currently being added to CORBA. The POA solves many issues with existing non-portable object implementations, which is the bane of cross-vendor CORBA development today.

References

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

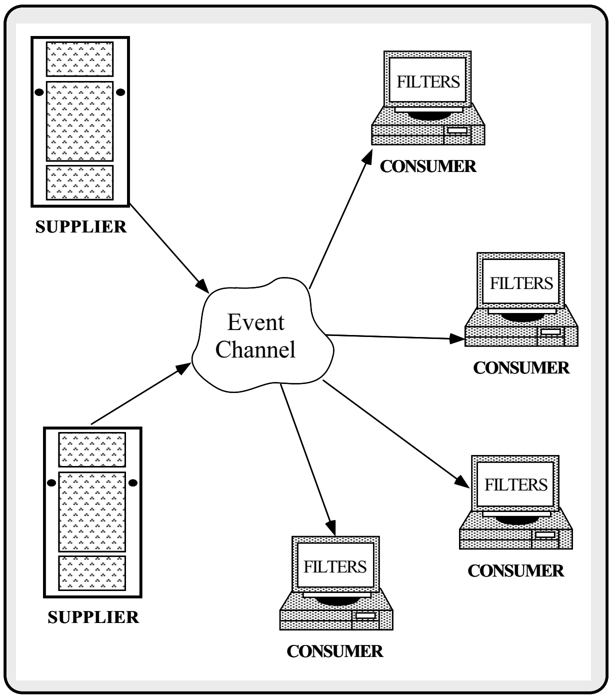

Figure 7: Decentralized Event Filtering

[2] Object Management Group, *Notification Service Request For Proposal*, OMG Document telecom/97-01-03 ed., January 1997.

[3] D. Schmidt and S. Vinoski, "Distributed Callbacks and Decoupled Communication in CORBA," *C++ Report*, vol. 8, October 1996.

[4] D. C. Schmidt, "High-Performance Event Filtering for Dynamic Multi-point Applications," in 1^{st} *Workshop on High Performance Protocol Architectures (HIPPARCH)*, (Sophia Antipolis, France), INRIA, December 1994.

[5] T. Harrison, D. Levine, and D. C. Schmidt, "The Design and Performance of a Real-time Event Service," in *Submitted to OOPSLA '97*, (Atlanta, GA), ACM, October 1997. http://www.cs.wustl.edu/∼schmidt/oopsla.html.

A Event Filtering Architectures

There are several types of event filtering architectures illustrated in Figures 7, 8, and 9. This appendix outlines the advantages and disadvantages of each architecture.

• **Decentralized Event Filtering:** In certain environments, it is beneficial to decentralize event filtering by performing it on consumer hosts (shown in Figure 7). This configuration is appropriate when the following conditions exist:

- The Consumer hosts are powerful computing platforms;
- A high-speed network is available to connect the Suppliers to the Consumer hosts;
- Consumers subscribe to most events;
- Event filters are relatively complex.

When these conditions exist it may become more efficient to perform filtering in the Consumer endsystems.

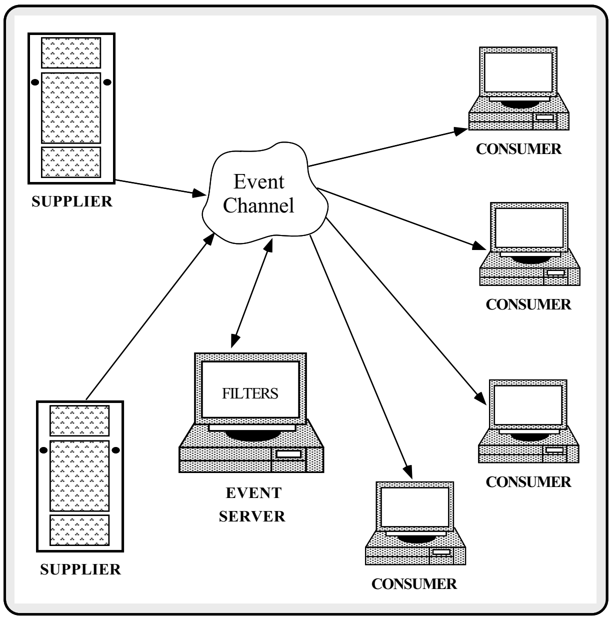

Figure 8: Centralized Event Filtering

• **Centralized Event Filtering:** In other environments, it is beneficial to centralize the event filtering in one Event Channel located on a single event server (shown in Figure 8). This configuration is appropriate when the following conditions exist:

- An Event Channel is installed on a high-performance event server platform (such as a multi-processor);
- The Consumer hosts are run on less powerful platforms (such as inexpensive PCs);
- A relatively low-bandwidth (or highly congested) network connects the event server to the Consumer hosts;
- Consumers subscribe to a relatively limited subset of events;
- The complexity and number of event filters subscribed to by Consumers does not produce a major processing bottleneck at the event server.

When these conditions exist, the network and the Consumer hosts at the edges of the network are typically the processing bottleneck, rather than the Event Channel running on the event server. Therefore, a centralized event filtering architecture helps to off-load work from the network and the Consumer hosts.

• **Distributed Event Filtering:** More complex event filtering scenarios are also possible (shown in Figure 9). For example, network topology in complex systems may interconnect Suppliers, Event Channels running on event servers, and Consumers that span multiple computers across local-area networks and wide-area networks.

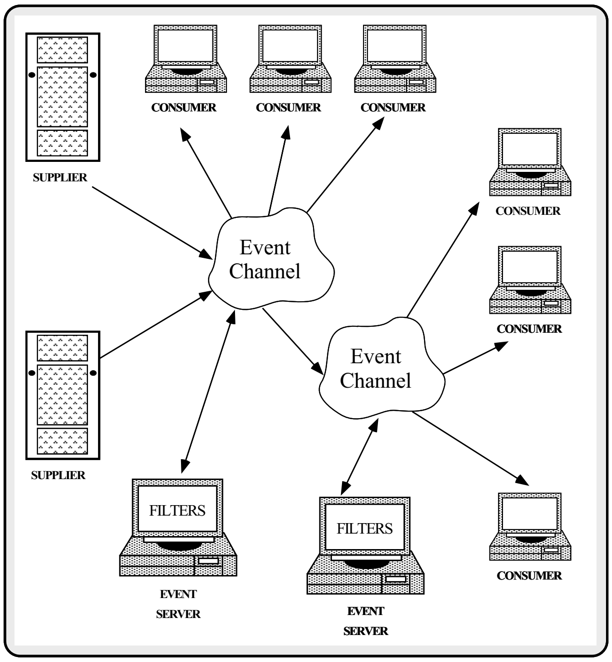

Figure 9: Distributed Event Filtering