5 Conclusion

The CORBA OMG Common Object Services specification defines a set of event services to support asynchronous communication between these potentially distributed CORBA objects. While the nature of the interaction between clients, event channel objects and event channel proxies is fairly well specified, the COS specification leaves open a variety of implementation and optimization options. This paper has examined a variety of implementation options for event channels, both inside and outside the CORBA invocation model. The EventHub model of event delivery provides an efficient way to implement COS event channels. EventHub also opens a possible path towards interoperability between CORBA and non-CORBA applications by providing a generic and flexible event delivery mechanism that is useful in both domains.
Figure 12: COS event delivery in the EventHub implementation.
to accomplish event transport and reaction triggering. The proxies relationship with their clients is not

affected by the use of EventHub and remains based on CORBA invocation for event supply and delivery. The only changes are in internal implementation of the proxies and the mechanisms through which event delivery between proxies is accomplished.

The manner in which CORBA event channels are implemented is relatively straightforward. After initialization, each ORB with clients participating in the event channel must have its EventHub directly connected to the other participating EventHubs. The event channel itself is represented by an EventHub reaction list in each of the participating ORBs. For each ORB push consumer attached to the event channel there is an entry in the reaction list that causes that consumer’s push method to be invoked upon event arrival. These event arrival invocations can be done without even the involvement of the consumer’s proxy object. Event propagation between ORBs is handled by reaction routines which forward the event to the EventHubs in other participating ORBs. When an event is generated by a local push supplier, it is submitted as a locally generated event to the EventHub and the appropriate reactions are called to push the event on local consumers and forward it to the other participating EventHubs. In order to handle events arriving from the network, the forwarding reactions only forward locally generated events (which are tagged as such with event meta-information). This allows all events to be handled identically without creating event forwarding loops. A graphical representation of COS event delivery using EventHubs appears as Figure 12.

The EventHub implementation of event channels has some key advantages over the CORBA-based implementations. Most significantly, it can propagate an event between ORBs with a single message per ORB. This characteristic is also shared by the Concentrator Object implementation of Section 3.2.4, but the EventHub version has the additional advantage of a single layer of action dispatch processing. In the CORBA implementation, dispatch occurs first when a message arrives from the network and the CORBA run-time decodes it and directs it to the appropriate object in the form of a method call. In the concentrator object implementation, dispatch occurs again in the concentrator object to distribute the methods to the local consumers. EventHub eliminates one level of dispatch processing by performing the method invocations directly in the EventHub run-time.

The EventHub protocol also offers a good opportunity to take advantage of alternative low-level communication protocols that offer multicast communication. While the structure of the EventHub system minimizes communication as much as possible, an implementation built on top of TCP/IP still requires N messages to distribute an event to N EventHubs or ORBs. A key aspect of EventHub that allows this opportunity is that the messages that are sent to each remote EventHub are identical. This is an advantage that CORBA-based protocols are unlikely to possess because their remote object requests must contain the ID of a specific destination object. We will explore the advantages of multi-cast event distribution in future work.
it can be tied to the arrival of only a specific type of event. Multiple functions can be registered and each appropriate function will be called upon each event arrival.

**Association of meta-information with an event.** Some standard pieces of meta-information about an event, such as where it was generated or whether or not it was received from another EventHub or generated locally are automatically provided by the EventHub system. This information is also provided to event reaction routines but is not considered part of the event data. Rather, meta-information is intended to support the implementation of higher layers of event management, such as routing between non-connected EventHubs.

**Reaction can be customized by context.** If an event carries “Context” meta-information, the reactions to the event are not chosen from the default list. Instead the value of the context meta-information field names another reaction list from which the reaction is to be chosen. The system supports the creation, naming and population of multiple reaction lists as well as the unique global reaction list in each EventHub.

**Dynamic EventHub interconnection.** EventHub instances (processes in the prototype) support the dynamic creation and destruction of network connections to other EventHubs. This allows the construction of complex and dynamic topologies of EventHubs. Transmission of events between EventHubs is explicit, with no direct support for routing events between EventHubs which are not directly connected. However such forwarding could be easily accomplished through appropriate reaction specifications.

### 4.2 Implementing CORBA Event Channels

The EventHub system can efficiently implement CORBA event channels through the use of customized proxy objects that use EventHub services rather than CORBA invocations to manage inter-ORB event interaction. As shown in Figure 11, EventHub services do not replace CORBA invocations but exist alongside
channel object is mostly responsible for maintaining and updating channel subscriber information as it was in Section 3.2.3. However, instead of individual proxies performing event distribution directly, they invoke the concentrator objects for event delivery. The event channel has one concentrator object per ORB and these concentrators communicate to accomplish inter-ORB event delivery. Events are sent from the supplier’s proxy to the local concentrator object. That concentrator distributes the event to any local consumers as well as sending the event to remote concentrators. Upon receiving an event from another ORB, each concentrator distributes it to local consumers. Because this model uses only a single remote operation per remote ORB to distribute events to remote consumers, it makes much more efficient use of network resources than prior models. Unfortunately, it accomplishes this at a cost of introducing another level of indirection and once again making event delivery a multi-step process.

Because the basic CORBA operation is a one-to-one method invocation, we cannot both reduce the inter-ORB traffic to one invocation per ORB per event and maintain single-step event delivery. In order to accomplish this we have to use facilities outside CORBA. The next sections describe such a facility.

4 The EventHub Model

The EventHub system provides simple, powerful and flexible support for event handling in distributed applications. It is not specifically tied to object-oriented systems or to the CORBA model, but rather provides generic event functionality that could be used in concert with CORBA facilities or independently to support events in non-CORBA distributed programs. It could also provide a mechanism through which CORBA and non-CORBA systems could interact.

For efficiency the EventHub software is built directly upon the OS provided network transport layer. In the case of the AIX prototype, it is built on top of TCP/IP, but could easily be ported to other network protocols.

4.1 EventHub Architecture

The basic structure of event deliver in EventHub is summarized in Figure 10. When a reaction is called, information about the event is supplied to the reaction routine as a parameter. This information includes the data carried with the event as well as meta-information about the event. EventHub has the following characteristics:

**Facilities for defining the types of architected events.** The data type associated with an architected event is a structure with named fields. The field’s data types may be simple atomic data types such as integers, floats, characters and strings, or they may be previously defined structured data types. This latter facility supports the creation of hierarchical types.

**Low overhead binary data transmission between heterogeneous systems.** EventHub data transmission is based on a simple portable binary I/O library. The library supports one-time data type registration so that subsequent data records do not need to contain full type information, but instead are preceded by a value identifying the type of the incoming data. The library uses a reader-makes-right protocol for heterogeneous data transmission. This protocol results in very low overhead for transmission between homogenous systems. Between heterogenous systems, the library automatically compensates for differences in machine byte-order and differing data type sizes or structure layouts.

**Flexible specification of event reaction.** EventHub supports the registration of functions to be called upon event arrival. This reaction specification can be generic, to be called for any incoming event, or
Figure 8: An event channel implementation strategy where proxies directly perform event distribution.

Figure 9: A more message-efficient event channel implementation.
Multiple remote operations to single ORBs: This implementation requires a remote operation for event delivery to each remote consumer. Thus multiple remote operations are required to deliver a single event to remote ORBs containing multiple consumers.

Because event channel subscribers may come and go dynamically, these disadvantages are unlikely to be significantly ameliorated by compile-time analysis of the application or by optimizations of ORB services. However, other implementation approaches show promise for improvement.

3.2.3 Active Event Channel Proxies

Figure 8 shows a slightly improved event channel implementation scheme. In this model, the event channel object itself takes no active role in event delivery. Instead, the event channel proxies directly invoke each other in order to deliver events to their destinations. The event channel object is still involved in client subscription and as clients join and leave the event channel, it must update the remaining proxies with current subscriber directory information.

While more complex, this implementation model removes two of the disadvantages which plagued the model of Section 3.2.2. Event delivery from proxy to proxy is now performed in one step and delivery between local proxies is performed without remote operations regardless of the location of the event channel object. However, it still requires multiple remote operations to deliver a single event to remote ORBs containing multiple consumers. This disadvantage is addressed in the implementation model in the next section.

3.2.4 Concentrator Objects

In our last model for implementing event channels on top of basic CORBA object services, we introduce additional objects to act as message concentrators, as shown in Figure 9. In this implementation, the event
3.2 Implementations Based on CORBA

Like most of OMGs Common Object Services, event services were designed to be implementable using standard CORBA requests (object method invocations). The next section will present and discuss several options for implementing COS event channels using the CORBA model.

3.2.1 Terminology

The terminology of the COS Event Services specification, while natural when discussing event channels from a client’s point of view, is somewhat less natural for discussing event channel implementations. From an event channel’s point of view, ProxyPushConsumer and ProxyPullConsumer objects are where events enter the event channel. Events leave the event channel at ProxyPushSupplier and ProxyPullSupplier objects. Since this terminology is somewhat counter-intuitive, we use the term supplier’s proxy to name the proxy object attached to clients which supply events to the event channel. Similarly, we use consumer’s proxy to refer to the proxy object associated with clients which consume events from the channel.

3.2.2 Active Event Channel Objects

Perhaps the simplest possible event channel implementation is shown in Figure 7. In this implementation model, the event channel object itself serves as a hub to which all events are sent for further distribution to event consumers. While this implementation is likely to be the most straight-forward and least complex, it has some significant performance disadvantages.

**Event delivery is a two-step process:** Because events must be first delivered to the event channel and then dispatched to the consumer, latency in event deliver is higher than necessary.

**Local event delivery may require remote operations:** Event communication between producers and consumers that are not in the ORB of the event channel object is performed via remote operations, even when the producer and consumer are in the same ORB.
inter-ORB delivery of events must be kept as low as possible. The number of event channels potential required to meet the demands of the object-based decomposition of monitoring computation indicates that the event channel system must also be highly scalable. With these considerations in mind, the next section will present some possible options for event channel implementation.

3 Event Channel Implementation

In the CORBA Event Services specification, event channels support asynchronous event delivery. Like all of the CORBA Common Object Services, Event Services are designed to be implementable using the basic CORBA object invocation functionality. This section will explain the basic nature of event channels, discuss possible strategies for implementing them using basic CORBA services and consider the ways in which these might be inefficient.

3.1 Basics of Event Channels

Abstractly, an event channel defines the domain in which an event is transported. Suppliers attach to an event channel and produce events for it. Consumers attach similarly and receive those events. Producers and consumers can each be either active or passive in the event delivery process. In the CORBA model, event channels themselves are represented by a CORBA object. These objects (and the abstract event transport domains they represent) are created like any other CORBA object. During the attachment phase, producers and consumers invoke a special method in the event channel object to obtain a proxy object representing the event channel. This process is depicted in Figure 5. The proxy object is created by the event channel object and a reference to it is returned to the attaching consumer or supplier. Thereafter, that proxy is the only interface used by the consumer or supplier in their interaction with the event channel. Thus, after the attachment phase, the proxy represents the interface to the abstract event transport domain represented by the event channel. This relationship is depicted in Figure 6. The inner workings of the event channel, such how the events are actually moved from a supplier’s proxy to the consumers’ proxies is hidden inside the implementation of the event channel proxy. This gives the event channel implementor a variety of options for managing event delivery. The remainder of this section will examine possible implementation schemes.

All examples below are directed towards what COS calls the Push model of event communication. In this model, event suppliers call the proxy’s Push() method to submit an event to the event channel. To deliver an event to a Push consumer, the consumer’s proxy invokes the consumer’s Push() method. To avoid overly complicating the implementation discussions we address only the active Push model and assert that
it would be possible to designate a single object in each ORB to receive all events of a particular type and then redistribute them to the appropriate object (or simply to have each object receive all events and ignore those that don’t apply to it), it seems more natural to employ multiple event channels to segregate event delivery. In this case instead of single event channels forming the connections between the applications and monitoring ORBs as shown previously, many event channels would be used, each carrying events specific to a particular application object. This is depicted in Figure 4.

2.3 Some Efficiency Considerations

This analysis of online distributed monitoring leads to some efficiency considerations for COS event channels if this type of application is to be successfully supported. In general, for monitoring event delivery needs to be fast and efficient as any latency and bandwidth limitations will directly affect the quality and extent of monitoring that is possible before the monitoring system saturates. Therefore the overhead involved in
2.2 The CORBA/COS Implementation

To achieve the type of application dataflow shown in Figure 2, the processing associated with nodes in the graph must be distributed among the available processing resources. In the CORBA model, processing is distributed by assigning CORBA objects, and therefore the calculations associated with their methods, to the available ORB servers. Thus the nodes in the graph of Figure 2 become ORB servers containing objects that perform the data gathering, analysis and reduction. Since the monitoring application is naturally event-driven, CORBA event channels are an obvious choice for communication between these objects. However, even given the basic topology and dataflow description, there are a variety of ways in which event channels could be used. We consider two basic scenarios for decomposing the calculations between the objects in the ORBs: functional and object-oriented.

If the data analysis calculations are decomposed functionally then it is likely that individual objects will either be interested in receiving all events or perhaps a subset of events by type. In this circumstance, it is sufficient to use a single event channel in each place where event transport must occur between ORBs. This arrangement is shown in Figure 3.

However, more complex situations may stretch this simple model. For example, in application-level monitoring where the user is employing monitoring to track the high-level behavior of his application, the data reduction and analysis computations required are more likely to reflect the object-oriented structure of the application. For example, at the first level of interface to the application, objects in the monitoring system may “mirror” certain attributes of objects of in the running program. To support this, it would be necessary for changes in the application objects to generate events which would cause corresponding changes in the mirror objects. Those events in turn might cause changes in other monitoring objects which would have to be reflected in yet other monitoring ORBs or displays. If we assume that applications objects of the same type generate monitoring events with the same type, then we have a situation where just the type of an incoming monitoring event is not sufficient to direct it to the right object in the monitoring system. While
2 Event Channel Use

In order to propose and evaluate possible implementation options for CORBA event channels, it is useful to understand how event channels might be used in an application. In this section we describe an online distributed monitoring and steering system and discuss how it might be implemented as a set of CORBA objects and how event channels might be used in its construction.

2.1 Online Monitoring as a Distributed Application

The relationship between a distributed application, a monitoring system, and the displays and interaction mechanisms it drives is shown abstractly in Figure 1. The high-level requirements for an online monitoring system are that it interact with the application, collect, reduce and analyze data and distribute the results to the appropriate displays. Because monitoring systems are typically employed in situations where application perturbation must be kept to a minimum, the monitoring systems interactions with the application are kept as non-intrusive as possible and its processing demands are met on resources outside domain of the application. While it is possible to implement the monitoring system in a centralized way with all raw monitoring data being sent to a central location for processing and distribution to displays, other approaches are more efficient and therefore could handle higher event rates before saturating. For example, it is desirable to perform monitoring data reduction and analysis as close to the point of data production as possible to reduce transmission bandwidth requirements. However because multiple independent data collection processes may be necessary, all the data for a particular calculation may not be available at single processing site. In this circumstance, more levels of data exchange may be necessary before the particular calculation can be performed. Generally, the data reduction and analysis can be viewed as a dataflow graph as shown in Figure 2, with raw monitoring data entering on one side of the graph and processed data driving displays on the other side. If application steering is to be performed, it may be initiated by human users by interacting with displays or algorithmically, through calculations and application changes initiated at some point in the data reduction. However, steering interactions are generally much less frequent than monitoring events, so this discussion will concentrate largely on monitoring event flow.
On the Implementation of CORBA Event Channels

Greg Eisenhauer    Bodhi Mukherjee    Chris Codella
{s5eisen,bodhi,codella@watson.ibm.com}
IBM T.J. Watson Research Center

Abstract

Event channels in the CORBA Common Object Services specification are an abstraction designed to provide asynchronous event delivery and notification for distributed object-oriented programs. However, the COS specification leaves open a variety of event channel implementation and optimization options. This report examines and evaluates possible options for implementing event channels. To provide additional requirements and drive the discussion it examines the manner in which event channels might be used to implement an online distributed monitoring and steering system.

1 Introduction

The CORBA model defined by the Object Management Group (OMG) provides a means by which objects may make and receive requests and responses in a distributed environment. The OMG Common Object Services (COS) specification defines a set of event services to support asynchronous communication between these potentially distributed CORBA objects. Event suppliers produce data that is eventually to be consumed by event consumers. The base model for event communication matches consumers and suppliers in a one-to-one way. An event channel is an intervening CORBA object that allows asynchronous event communication between multiple consumers and suppliers. Rather than directly communicating with the ultimate event source or consumer, event channel clients submit and receive events to and from proxy objects operating on behalf of the event channel.

While the nature of the interaction between clients, event channel objects and event channel proxies is fairly well specified, the COS specification leaves open a variety of implementation and optimization options. Once events have entered the domain of the event channel via a proxy, the nature of event propagation and delivery is unspecified. Because high bandwidth and minimum latency of event services will be key factors affecting performance of distributed event-based applications, this paper is concerned with possible implementation options for COS event channels.

To provide context and requirements for discussing event channel implementations we first examine the manner in which event channels might be used in a sample application, in this case an online distributed monitoring and steering system. The next section examines possible event channel implementation approaches based on CORBA facilities and reaches conclusions on the limits of their efficiency. Section 4 presents EventHub, a set of non-CORBA facilities for providing event services and examines the manner in which it could be used to implement COS event channels.