Object Interconnections

The OMG Events Service (Column 9)

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 February 1997 issue of the SIGS C++ Report magazine.

1 Introduction

In our previous column, we modified our stock quote system to implement *distributed callbacks* using CORBA, as shown in Figure 1.

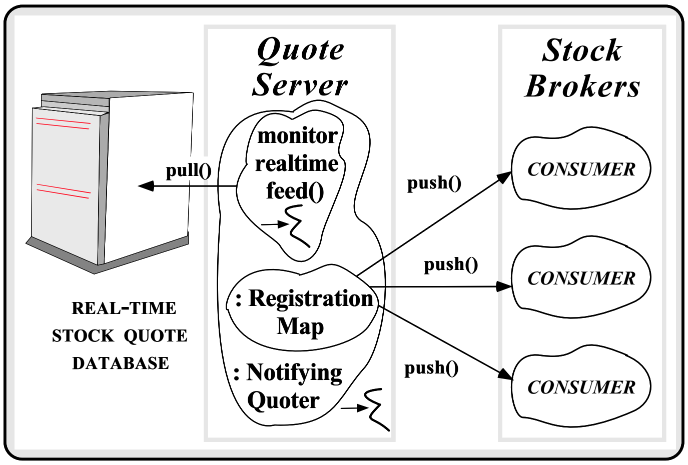

Figure 1: CORBA Distributed Callback Quoter Architecture

Distributed callbacks differ from CORBA's conventional synchronous invocation model because they decouple the request for a service from the response(s). Callbacks are often useful when consumers of events don't need to wait synchronously for suppliers to generate the events. In addition, they can be used to deliver responses to long-running operations, rather than making clients block waiting for operations to complete.

By definition, distributed callbacks flow from the server back to the client. Therefore, they turn a standard client/server relationship into a *peer-to-peer* relationship. This reinforces the fact that terms like "client" and "server" are useful only for describing the roles played for each CORBA request. By re-architecting our distributed stock quote server to employ callbacks, we showed how the flexibility of making requests in "both directions" can solve problems associated with CORBA's synchronous request/response model. Chief among these problems is the network/server saturation caused by polling operations.

As we pointed out last column, however, the distributed callback design introduced a new set of problems, which include the following:

- **Supplier-side polling:** Instead of just looking up stock values upon request, our `Notifying_Quoter` must actively monitor stock values to detect all changes that distributed Consumers have registered interested in. Performing this efficiently requires the use of either a separate monitoring thread that polls the server (as shown in in Figure 1) or an active database that triggers the application when changes occur. In either case, the complexity of the server increases and the portability potentially decreases.

- **Persistence of callback object references:** The Supplier's `Notifying_Quoter` implementation must persistently maintain all the object references in its callback `Registration Map`. Persistence is necessary so that deactivation and subsequent reactivation is transparent to the callback objects registered by Consumers. The need for persistence is new since our original stock quote server had no persistent storage requirements of its own (the stock quote database is considered as being external to the stock quote server).

- **QoS tradeoffs:** Our `Notifying_Quoter` must service multiple Consumers, each of which may have different quality of service (QoS) requirements. For instance, some Consumers may be willing to pay extra to receive stock change notifications immediately, whereas others may want to receive them in batches in order to reduce costs. Therefore, our `Notifying_Quoter` must issue callbacks in a timely manner, taking into account different QoS needs. Meeting all these needs is hard, particularly when we must also explicitly handle variability in network/host workload (*e.g.,* the number of callback objects currently registered with it, the network congestion, etc.).

- **Potential for deadlock:** Stock quote Consumers use the `Notifying_Quoter`'s `unregister_callback` method to remove themselves from the Supplier's callback map. Since `unregister_callback` is a twoway

CORBA call, our distributed callback design can deadlock if the Supplier tries to `push` to a Consumer that is simultaneously trying to unregister. This will almost certain happen if the ORB doesn't support "nested dispatching" of twoway calls.[1] If nesting dispatching is not supported, we would need to restructure the `Handler::push`, `Notifying_Quoter::register_callback`, and the `Notifying_Quoter::unregister_callback` to use oneway semantics. Even this solution is problematic, however, since CORBA oneway calls are not reliable. Therefore, application developers become responsible for ensuring end-to-end reliability, which can be tricky and inefficient.

The problems described above with our notifying stock quoter can be eased somewhat if we separate concerns. In particular, the Supplier's `Notifying_Quoter` implementation has enough to worry about within its own problem domain as it monitors and reports changing stock values. Therefore, we should avoid making it also responsible for delivering notifications to multiple Consumers, handling blocking caused by network congestion and endsystem load, and maintaining a persistent table of callbacks. All these tasks are independent of the stock quoter application domain.

One way to relieve some of the burden we've placed on the stock quoter is to utilize an implementation of the OMG *Events Service* to deliver notifications. The Events Service is one component in the OMG Common Object Services Specification (COSS) Volume 1 [2]. Its purpose is to provide delivery of event data from suppliers to consumers without requiring these participants to know about each other explicitly. Therefore, implementations of the Events Service act as "mediators" that support decoupled communication between objects.

This column is organized as follows: Section 2 outlines the role of key components in the OMG Events Service, Section 3 examines the IDL interfaces of the Events Service components in detail, Section 4 illustrates and evaluates an implementation of the distributed stock quoter system using the Events Service, Section 5 discusses the strengths and weaknesses of the OMG Events Services model and its specification, and Section 6 provides concluding remarks.

2 Overview of the OMG COS Events Service

Figure 2 illustrates how clients and servers interact using the standard CORBA twoway communication model. In this model, CORBA clients invoke operations on a target object located at a server and synchronously wait for the server to reply. One benefit of this request/response model is how well it conforms to the expectations of programmers accustomed to developing stand-alone OO applications. One drawback, however, is that the server must be available to process the client's request.

[1] Nested dispatching enables an ORB to perform upcalls from incoming requests even while it is "blocked" on the request portion of a twoway request/response invocation. There are various ways to implement nested dispatching, such as spawning off a separate thread to handle each incoming request or using a non-blocking, reactive [1] event loop within the ORB.

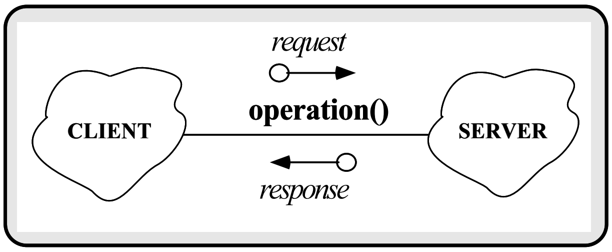

Figure 2: The CORBA Request/Response Model

There are many situations where the standard CORBA synchronous request/response model is too restrictive. For instance, clients in the original implementation of our quoter system had to poll the server repeatedly to retrieve the latest stock prices. Likewise, there was no way for the server to efficiently notify groups of interested clients *en masse* when stock prices changed.

The OMG COS Events Service is designed to alleviate these restrictions by supporting decoupled communication among multiple Suppliers and Consumers. As we noted in our previous column, a Supplier is an entity that produces events, while a Consumer is one that receives event notifications and data. Events are typically represented as messages that contain optional data fields.

The remainder of this section outlines the roles and relationships of key components in the COS Events Service.

2.1 Events Service Components

Figure 3 shows the three primary components in the OMG COS Events Service architecture.

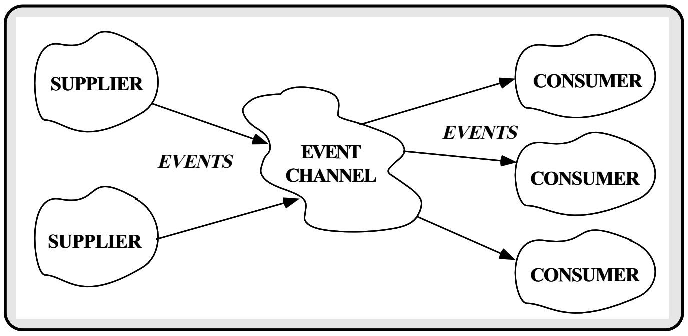

Figure 3: High-level View of the OMG COS Events Service

These three components are described below:

• **Suppliers and Consumers:** Consumers are the ultimate targets of events generated by Suppliers. Suppliers and Consumers can both play active and passive roles. A Push Supplier object can actively *push* an event to a passive Push

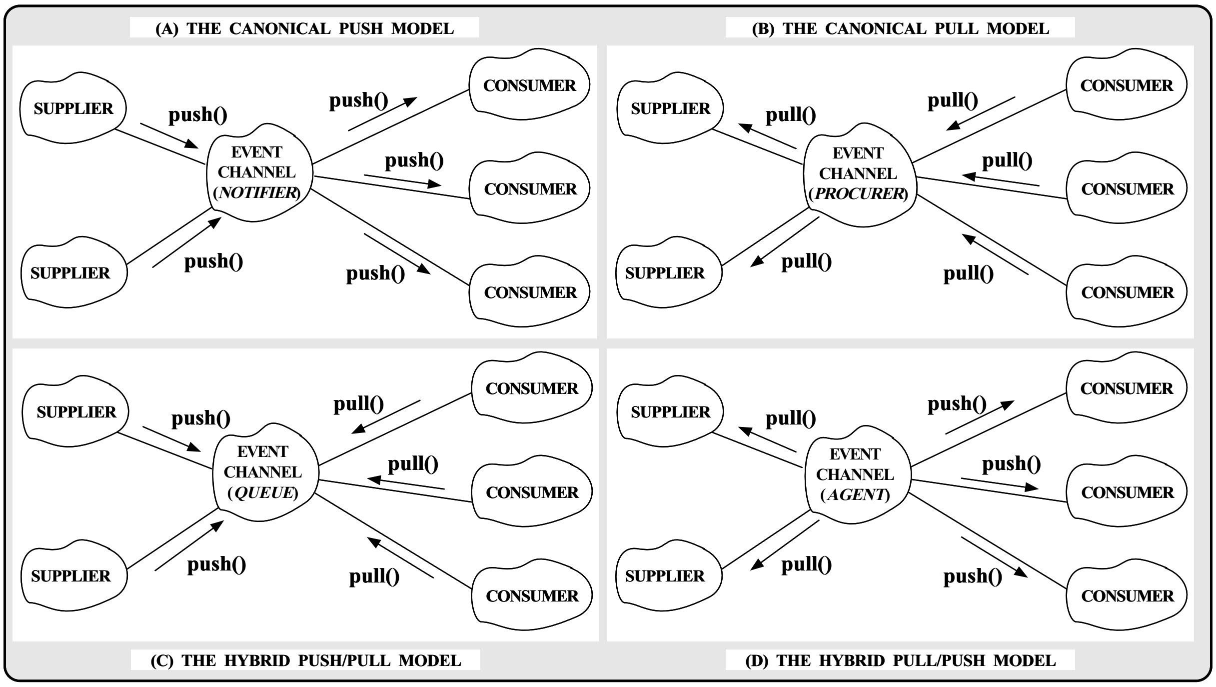

Figure 4: Events Service Communication Models

Consumer object. Likewise, a Pull Supplier object can passively wait for a Pull Consumer object to actively *pull* an event from it.

• **Event Channel:** The central abstraction in the COS Events Service is the Event Channel, which plays the role of a mediator between Consumers and Suppliers. The Event Channel manages object references to Suppliers and Consumers. The Event Channel appears as a "proxy" Consumer to the real Suppliers on one side and as a "proxy" Supplier to the real Consumers on the other side.

Suppliers use Event Channels to push data to Consumers. Likewise, Consumers can explicitly pull data from Suppliers. The push and pull methods of event propagation free Consumers and Suppliers from the synchronous semantics of the standard CORBA "request/response" communication model. In addition, an Event Channel serves as a surrogate to multiple Suppliers and multiple Consumers, which supports group communication [3].

2.2 Events Service Communication Models

There are four general models of component collaboration in the OMG COS Events Service architecture. Figure 4 shows the collaborations between Consumers and Suppliers in each of the models. The following examines these models in detail:

• **The Canonical Push Model:** The canonical Push model shown in Figure 4(A) allows the Suppliers of events to initiate the transfer of event data to Consumers. In this model, Suppliers are the active initiators and Consumers are the passive targets of the requests. Event Channels play the role of *Notifier*, as defined by the Observer pattern [4]. Thus, active Suppliers use Event Channels to push data to Passive Consumers that have registered with the Event Channels.

• **The Canonical Pull Model:** The canonical Pull model shown in Figure 4(B) allows Consumers to request events from Suppliers. In this model, Consumers are the active initiators and Suppliers are the passive targets of the pull requests. Event Channels play the role of *Procurer* since they procure events on behalf of Consumers. Thus, active Consumers can explicitly pull data from Passive Suppliers via the Event Channels.

• **The Hybrid Push/Pull Model:** The Push/Pull model shown in Figure 4(C) is a hybrid that allows Consumers to request events queued at a Channel by Suppliers. In this model, both Suppliers and Consumers are the active initiators of the requests. Event Channels play the role of *Queue*, as defined in the Active Object pattern [5]. Thus, active Consumers can explicitly pull data deposited by active Suppliers via the Event Channels.

• **The Hybrid Pull/Push Model:** The Pull/Push model shown in Figure 4(D) is another hybrid that allows the Channel to pull events from Suppliers and push them to Consumers. In this model, Suppliers are passive targets of pull requests and Consumers are the passive targets of pushes. Event Channels play the role of *intelligent agent*. Thus, active Event Channels can pull data from passive Suppliers and push that data to passive Consumers.

The following table summarizes the role of the Event Channel as a function of the communication model:

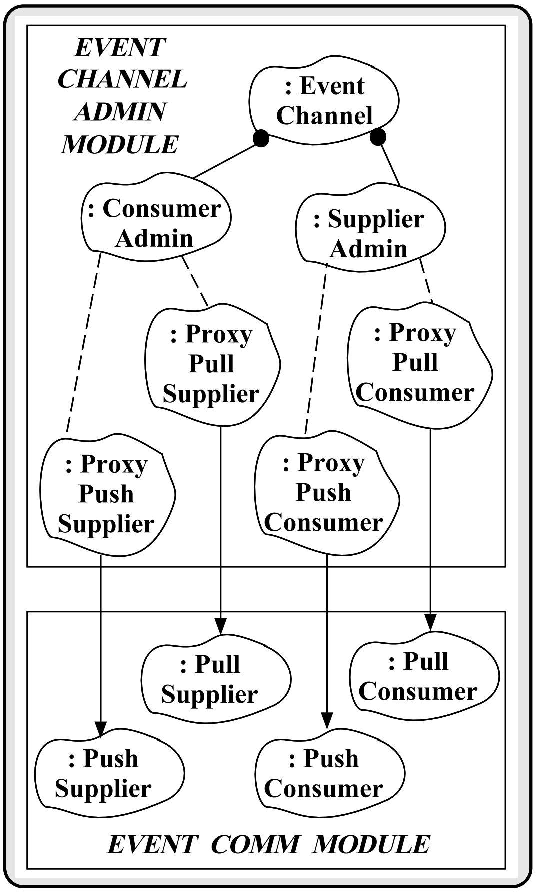

Figure 5: Structure of IDL Interfaces in the OMG COS Events Service

Consumer Role	Supplier Role	
	Push	Pull
Push	Notifier	Agent
Pull	Queue	Procurer

2.3 Events Service Type Systems

The following two standard type systems are defined in the OMG Events Service:

• **Typed event systems:** A "typed" events service is one where Supplier and Consumer applications share knowledge of application-dependent interfaces. These interfaces allow them to pass typed events through an Event Channel. The `Notifying_Quoter` implementation shown in our last column shared some characteristics of a typed event system because both the quoter object and the callback object knew the exact type of data passed in the event notifications. In addition, to simplify our design the OMG IDL interfaces used in the system weren't based on the standard OMG Events Service interfaces. Instead, we customized the interfaces for the specific needs of our stock quoter callback system.

• **Untyped event systems:** In the untyped approach event data is passed through the system in the form of the OMG IDL `any` type. The `any` type can hold an instance of any other built-in or user-defined OMG IDL data type. In addition, it holds a `TypeCode`, which is a runtime type tag that identifies the type of the data. Passing event data as an `any` means that applications using the untyped approach can be based on the standard application-independent interfaces specified in the OMG COS specification.

We use the untyped approach for the examples shown in this column since it is the most widely available approach used in existing Events Services implementations (such as the HP ORB Plus Events Service and IONA's OrbixTalk).

3 IDL Interfaces of the Events Service

The interfaces for all OMG Common Object Services (COS) are defined using OMG IDL. Figure 5 illustrates the structure and the relationships of the IDL interfaces that comprise the OMG COS Events Service. The two primary components in the Events Service architecture are the `CosEventComm` module and the `CosEventChannelAdmin` module. This section examines these modules in detail.

3.1 The CosEventComm Module

This module defines a set of IDL interfaces for event communication between push-style and pull-style Consumers and Suppliers. The following OMG IDL illustrates the key interfaces and operations in this module:

```
module CosEventComm
{
  exception Disconnected {};

  // A push-style consumer implements this
  // interface to receive data from a supplier.
  interface PushConsumer { /* ... */ };

  // A push-style supplier implements this
  // interface to disconnect from a supplier.
  interface PushSupplier { /* ... */ };

  // A pull-style supplier implements this
  // interface to transmit data to a consumer.
  interface PullSupplier { /* ... */ };

  // A pull-style consumer implements this
  // interface to disconnect from a consumer.
  interface PullConsumer { /* ... */ };
};
```

For brevity, only the interfaces are shown. We show more detail (*e.g.,* operations and exceptions) as needed later on.

3.2 The CosEventChannelAdmin Module

This module defines the interfaces for establishing connections between Suppliers and Consumers. Connection establishment is a multi-step process. The OMG IDL shown below illustrates the `EventChannel` operations that Consumers and Suppliers must call first. Only the most relevant interfaces and operations are shown.

```
module CosEventChannelAdmin
{
  // A factory for creating proxies
  // that allows Consumers to connect
  // to an Event Channel.
  interface ConsumerAdmin { /* ... */ };

  // A factory for creating proxies
  // that allows Suppliers to connect
  // to an Event Channel.
  interface SupplierAdmin { /* ... */ };

  interface EventChannel
  {
    // Returns an object reference
    // for creating Supplier proxies.
    ConsumerAdmin for_consumers ();

    // Returns an object reference
    // for creating Consumer proxies.
    SupplierAdmin for_suppliers ();

    // Shutdown a Channel.
    void destroy ();
  };
};
```

Consumer administration and Supplier administration are defined separately for the following reasons:

- *Minimizing "surface area"* – Consumers do not have to be bothered with the additional "surface area" of the interfaces intended for use only by Suppliers, and vice-versa. Consumers and Suppliers deal only with the interfaces they need to get connected and push/pull events. This is useful to simplify applications that don't need the full power (and complexity) of COS Event Channels.
- *Access control* – The creator of a Channel can control the addition of Suppliers and Consumers. This is useful for ensuring certain types of security. For instance, a Network Management Agent implemented using an Event Channel might allow a variety of Consumers to register to receive trap events, but restrict Supplier access to only allow the MIB to update the Channel.
- *Third party connections* – External agents can transparently connect multiple Channels together. This is useful for creating pipelines and graphs of connected Channels.

Next, we examine the key Consumer and Supplier IDL interfaces for the COS Events Service in more detail.

3.3 Interfaces for Event Consumers

In order for Consumer applications to receive events from a Supplier via an `EventChannel`, they must each first connect to the Channel, which requires the steps shown in Figure 6. These steps are explained below:

1. Obtain a ConsumerAdmin factory: Consumers that want to connect to an Event Channel must first invoke the `EventChannel`'s `for_consumers` operation to obtain a `ConsumerAdmin` object reference. The `ConsumerAdmin` is a factory that returns object references to Supplier proxies. Its IDL interface is shown below:

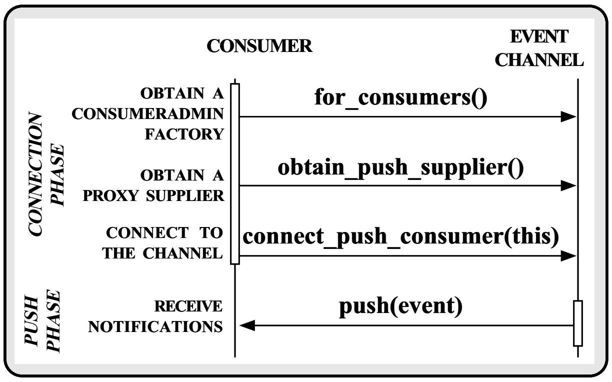

Figure 6: Connecting a Consumer to an Event Channel

```
// The following are defined in the
// CosEventChannelAdmin module.

// Define the second step for connecting
// push Consumers to an Event Channel.
interface ProxyPushSupplier { /* ... */ };

// Define the second step for connecting
// pull Consumers to an Event Channel.
interface ProxyPullSupplier { /* ... */ };

// Define the first step for connecting
// Consumers to an Event Channel.

interface ConsumerAdmin
{
  // Returns an object reference that can be
  // used to connect to a push-style Consumer.
  ProxyPushSupplier obtain_push_supplier ();

  // Returns an object reference that can be
  // used to connect to a pull-style Consumer.
  ProxyPullSupplier obtain_pull_supplier ();
};
```

2. Obtain a Proxy Supplier: After calling the `for_consumers` operation to get an object reference to the `ConsumerAdmin` factory from the `EventChannel`, Consumers 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 the 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 the Event Channel: Once Consumers obtain the appropriate Supplier proxy, they use the proxy to connect themselves to the 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.

As described in our previous column, our stock broker application wants to avoid polling the stock quoter, so we'll design it as a passive `PushConsumer`. Thus, it will connect using the `ProxyPushSupplier` interface, which is shown below along with its base interface:

```
// The following two interfaces are defined
// in the CosEventComm module.

// A push-style supplier implements this
// base interface to disconnect from a supplier.
interface PushSupplier
{
  // Called by the Channel to disconnect
  // the Supplier.
  void disconnect_push_supplier ();
};

// A push-style consumer implements this base
// interface to receive data from a supplier.
interface PushConsumer
{
  // Transfer event data to the Consumer.
  void push (in any data)
    raises (Disconnected);

  // Called by the Channel to disconnect
  // the Consumer.
  void disconnect_push_consumer ();
};

// The following interface is defined
// in the CosEventChannelAdmin module.

// Define the second step for connecting push
// Consumers to an Event Channel.
interface ProxyPushSupplier :
  CosEventComm::PushSupplier
{
  // Connect pc to the Event Channel
  // via the PushSupplier proxy.
  void connect_push_consumer
          (in CosEventComm::PushConsumer pc)
             raises (AlreadyConnected);
};
```

The `ProxyPushSupplier` interface is defined in the `CosEventChannelAdmin` module. It is derived from the base `PushSupplier` interface defined in the `CosEventComm` module. The derived class adds the administrative `connect_push_consumer` operation, which allows Consumer objects to be connected to the Supplier via the Event Channel.

3.4 Interfaces for Event Suppliers

Suppliers connect to an Event Channel in a manner that is symmetrical to the approach used by Consumers. For completeness, key IDL interfaces are shown below:

```
// Define the second step for connecting
// push Suppliers to an Event Channel.
interface ProxyPushConsumer { /* ... */ };

// Define the second step for connecting
// pull Suppliers to an Event Channel.
interface ProxyPullConsumer { /* ... */ };

// Define the first step for connecting
// Suppliers to an Event Channel.
interface SupplierAdmin
{
  // Returns an object reference that can be
  // used to connect to a push-style Supplier.
  ProxyPushConsumer obtain_push_consumer ();
  // Returns an object reference that can be
  // used to connect to a pull-style Supplier.
  ProxyPullConsumer obtain_pull_consumer ();
};
```

As described in our previous column, our quote server wants to notify Consumers when stock updates arrive from a real-time market feed. Therefore, we design it to be an active `PushSupplier` that connects to the Channel using the `ProxyPushConsumer` interface shown below:

```
interface ProxyPushConsumer :
  CosEventComm::PushConsumer
{
  void connect_push_supplier
          (in CosEventComm::PushSupplier ps)
             raises (AlreadyConnected);
};
```

As before, the `ProxyPushConsumer` interface of the `CosEventChannelAdmin` is derived from the base `PushConsumer` interface of the `CosEventComm` module. The derived class adds the administrative `connect_push_supplier` operation, which allows Supplier objects to be connected to the Consumer via the Event Channel.

4 Using the OMG COS Events Service for the Stock Quoter System

The OMG Events Service specification [2] has been available for several years. It concisely describes the IDL interfaces of Event Channels, Consumers, and Suppliers. However, it is beyond the scope of the standard to illustrate how to develop applications using the Events Service. Therefore, the intent of this section is to explain how to program our stock quoter system using the OMG Events Service.

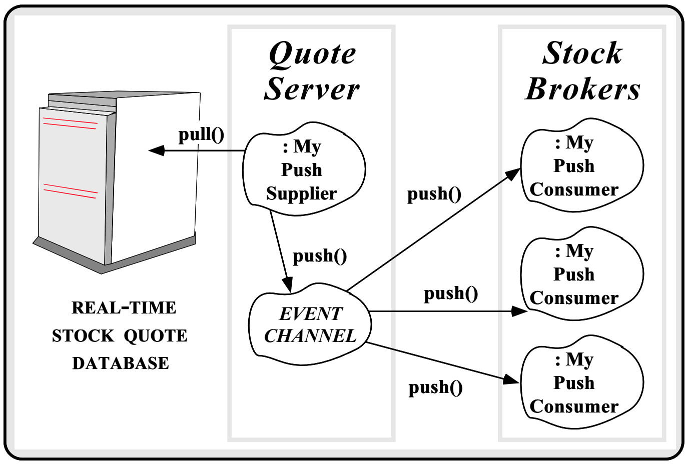

Figure 7: The Stock Quote System Design (Push Model)

All applications that make use of Event Channels must choose one of the usage models described in Section 2.2. As explained in our last several columns, our stock quote system uses the Push model to eliminate the Consumer polling that would be required if the synchronous request/response communication model were used. Figure 7 shows how the Event Channel fits into the overall configuration of our stock quote system.

To use an Event Channel, push Consumer and push Supplier applications need an `EventChannel` object reference and an object reference for an object implementing the `PushConsumer` or `PushSupplier` interface, respectively. The remainder of this section shows how the stock broker application and stock quote server use these object references to decouple the Suppliers from Consumers.

4.1 Using The Event Consumer Interface for the Stock Broker Application

The stock broker application, acting in the role of a push Consumer, first creates itself and then obtains an `EventChannel` object reference (*e.g.,* via the COS Naming or Trading Services, or via a user-defined factory), as follows:

```
// The stock broker application creates
// a push consumer object implementation.
MyPushConsumer *pc_impl = new MyPushConsumer;

// Obtain the PushConsumer object reference.
PushConsumer_var my_pc = pc_impl->_this ();

// EventChannel obtained from the COS
// Naming Service (not shown).
EventChannel_var ec = // ...
```

This code assumes that the `MyPushConsumer` C++ class supports the `PushConsumer` interface and implements the appropriate stock broker logic. Next, we call `_this` on the `MyPushConsumer` instance to get its object reference.[2] The stock application then connects itself to the `EventChannel` in the following manner:

```
// Obtain ConsumerAdmin object reference.
ConsumerAdmin_var ca = ec->for_consumers ();

// Obtain ProxyPushSupplier from
// the ConsumerAdmin object.
ProxyPushSupplier_var pps =
  ca->obtain_push_supplier ();

// Connect our PushConsumer to
// the ProxyPushSupplier.
pps->connect_push_consumer (my_pc);
```

When an event arrives in the Event Channel, it will invoke the `push` operation on the registered `PushConsumer` object reference. This, in turn, will deliver the event `any` data via an invocation of the `push` method of the `pc_impl` instance of the `MyPushConsumer` C++ class. It is important to remember that the `ProxyPushSupplier` object reference used here refers to an object within the Event Channel and is not an object reference for the real Supplier on the "other side" of the Event Channel (*i.e.,* in the stock quote server).

Our example shows only a single Consumer in the system. However, "real world" applications using the OMG Events Service normally have multiple consumers, and sometimes multiple suppliers as well. In a system with only one Consumer it may be easier and more efficient to just let the Supplier call back directly to the Consumer, as the `Notifying_Quoter` did in our previous column. It's easy to accomplish this with only minimal changes to our implementation, *e.g.,* by changing the `ProxyPushConsumer` object reference to point directly to a Consumer rather than to a Channel.

4.2 Using the Event Supplier Interface for the Quote Server

As with the stock broker Consumer application, the quote server Supplier needs an object reference for an `EventChannel` and an object reference for a `PushSupplier` to tie itself to the Channel. The quote server Supplier accomplishes this task in a very similar way to the manner in which the stock broker Consumer application did:

```
// Create the push supplier object implementation.
MyPushSupplier *ps_impl = new MyPushSupplier;

// Obtain a PushSupplier object reference.
PushSupplier_var my_ps = ps_impl->_this ();

// EventChannel obtained from the Naming
// Service (not shown).
EventChannel_var ec = // ...

// Obtain SupplierAdmin object reference.
SupplierAdmin_var sa = ec->for_suppliers ();

// Obtain ProxyPushConsumer from
// SupplierAdmin object.
ProxyPushConsumer_var ppc =
  sa->obtain_push_consumer ();

// Connect our PushSupplier to
// the ProxyPushConsumer.
ppc->connect_push_supplier (my_ps);
```

Once the Consumer registration shown earlier and the Supplier registration code shown here get executed, both the stock broker application and the quote server are connected to the Event Channel. At this point, Consumer(s) will automatically receive stock quote update events that are pushed by the Suppliers through the Channel. Additional Consumers and Suppliers can now register with the Event Channel while the system is running without affecting the quote server.

4.3 Exchanging and Processing Event Data

The events exchanged between Supplier and Consumer must always be specified in OMG IDL so that they can be stored into an `any`. The event data in our last column consisted of the stock name and its value grouped into an OMG IDL struct. We've now added a third field, as shown below:

[2] Calling `_this` in the manner shown here is supported by several ORB products, but it is not required by the CORBA 2.0 specification. With both IONA's Orbix and Expersoft's PowerBroker, for example, object implementations are derived from the object reference class, allowing object references to be obtained by merely taking the address of the implementation object.

```
module Stock {
  // forward declaration
  interface StockTrader;

  struct CallbackInfo {
    // Name of the stock we're
    // interested in.
    string stock_name;

    // Current value of that stock.
    fixed<6,2> value;

    // Object reference to a stock trader
    // that allows us to buy and sell.
    StockTrader trader;
  };
  // ...
};
```

For the stock value, our `CallbackInfo` struct makes use of a new feature of OMG IDL: the `fixed` data type. It is a template type that allows fixed precision values, such as monetary values, to be mapped into programming language types that are easy to manipulate. In addition, we've provided an object reference to a stock trader[3] that enables Consumers to buy and sell stocks at the indicated value.

In order to push an event, the quote server Supplier must create and initialize a `CallbackInfo` struct, put it into a `CORBA::Any`, and call `push` on the Event Channel `PushConsumer` interface:

```
  // Supplier-side implementation.

  using namespace Stock;

  // Create CallbackInfo struct.
  CallbackInfo info;
  info.stock_name =
    CORBA::string_dup ("ACME ORB Inc.");
  info.value = 103;
  info.trader = // Obtain an object reference
                // to a StockTrader (not shown).

  CORBA::Any event_data;

  // Put the value into an Any.
  event_data <<= info;

  try {
    // Push the event to Consumer(s).
    ppc->push (event_data);
  } catch (const Disconnected &) {
    // deal with disconnection
  } catch (const CORBA::SystemException &sx) {
    cerr << "CORBA system exception occured: "
         << sx << endl;
    // ...
  }
```

The insertion of the `CallbackInfo` struct into the `CORBA::Any` is accomplished using the overloaded `operator<<=`, which is defined in the OMG IDL C++ Mapping Specification [6]. The `TypeCode` for the `CORBA::Any` is set as a side-effect of this operation since the `TypeCode` is implied by the C++ `CallbackInfo` struct type.

[3]Note that this use of the term "trader" should not be confused with the OMG Trading Service, which allows applications to obtain object references based on object properties.

Once the Event Channel receives an event from the quote server Supplier, it pushes the event data to the Consumer(s) by invoking the `push` operation on the registered `PushConsumer` object reference. Note that `push` is a twoway call that doesn't return a value. Therefore, it can throw exceptions, so we must surround the call within a `try` block.

The implementation of the stock broker Consumer `push` operation is shown below:

```
// Consumer-side implementation.

void
MyPushConsumer::push
  (const CORBA::Any &event)
{
  Stock::CallbackInfo *info;

  // Extract the value of the Any into
  // the CallbackInfo struct.
  if (event >>= info)
  {
    cout << "Value of "
         << info->stock_name
         << " is "
         << info->value
         << endl;

    // Logic to determine whether to
    // buy or sell goes here...
  }
}
```

The consumer `push` function must ensure the event data it receives is actually the correct type it's expecting. This is accomplished using the overloaded `operator>>=` to extract typed data from a `CORBA::Any`. If the `TypeCode` implied by the second argument to `operator>>=` matches the `TypeCode` in the `Any` the extraction succeeds and the stock name and value from the `CallbackInfo` struct are printed to standard output. If the `Typecode` of the desired type does not match the `TypeCode` of the data within the `any` the extraction fails and the event data is safely and correctly ignored.

Other manipulation of the stock information, such as using it to buy or sell shares of the stock, could be performed at this point as well. In fact, the Consumer could perform these operations using synchronous requests, rather than going back through the Event Channel, as follows:

```
  if (buy)
    info->trader->buy (info->stock_name_,
                       num_shares,
                       info->value_);
  else if (sell)
    info->trader->sell (info->stock_name_,
                        num_shares,
                        info->value_);
  // ...
```

This type of hybrid "asynchronous notification – synchronous invocation" architecture is commonly known as "trap-directed polling" in the network management literature.

4.4 Evaluating the OMG COS Events Services Solution

4.4.1 Benefits

We decided to use a COS Event Channel to handle event deliveries and callback registrations to relieve the `Notifying_Quoter` from many low-level communication details. Our revised `Notifying_Quoter` receives no callback registration invocations. Therefore, it need not maintain any persistent storage for such registrations. It now has to monitor stock values, just as it did before, and generate events for those stock values that change. The `EventChannel` ensures that each event is distributed to all registered Consumers.

The symmetry underlying the Events Service model might also be considered as a benefit. It allows Consumers and Suppliers to connect and register with Event Channels in symmetrical ways. This simplifies application development and allows Event Channels to be chained together for bridging or filtering purposes. However, as we'll see below, this symmetry also has its drawbacks.

4.4.2 Drawbacks

Although our new solution improves some problems from last our column, using Event Channels has its own set of drawbacks. Some drawbacks are new and others are ones that our original `Notifying_Quoter` shown in Figure 1 had already fixed. The following describes all these problems:

• **Complicated Consumer registration:** Instead of using the simple callback registration interface previously provided by our `Notifying_Quoter`, our Consumer application now must know all the details of registering with Event Channels.

• **Lack of persistence:** The COS Events Service standard doesn't mandate that Event Channels provide persistence. Therefore, if problems occur and processes/hosts shut down unexpectedly, it's possible for Event Channels to lose events and connectivity information.

• **Lack of filtering:** The standard OMG COS Events Service specifies no filtering capabilities. Event Channels can have multiple Consumers connected to them, and they deliver all event data they receive to each and every connected Consumer. Because our new `Notifying_Quoter` no longer receives callback registrations from Consumers directly, it has no choice but to push *all* stock value changes into the Event Channel. This, in turn, means that each of our Consumer callback objects must filter its own event data. More importantly, it means that *all* stock value changes get pushed to each and every Consumer.

• **Increased endsystem network utilization:** If stock values change rapidly on the server, the Event Channel may end up sending many notifications to Consumers. However, because all filtering is performed by each callback object, the programs housing such objects use endsystem and network resources just to throw events away. Our original decision to use the callback approach was based on the desire to reduce network traffic by eliminating polling. Ironically, by using an Event Channel we may have actually *increased* network utilization! The problem, of course, is that the Consumers are now notified every time any stock changes value, rather than just when they are interested in reading the latest quote.

• **Multiple Suppliers:** An Event Channel can have multiple Suppliers connected to it, and each event from each Supplier is delivered to all connected Consumers. Depending upon how widely advertised the Event Channel object reference is, it could end up being used for many different purposes, by many different Suppliers. This multiplies the negative effects resulting from the lack of filtering and increased network utilization.

• **Lack of type-safety:** With an untyped event channel, event data is delivered via the OMG IDL `any` type. Multiple Suppliers can be connected to the same Channel, with each Supplier potentially delivering events for a different reason. Therefore, each Consumer must be prepared to actively distinguish the events it understands from those it does not.

• **Over-generalization:** While the symmetry provided by the Events Service model is elegant, it is also the source of unnecessary complexity since a less general event notification scheme would probably not support a "pull" model, and thus it would not need to support a bi-directional object reference handshake with Suppliers.

In our example, distinguishing stock callback data from other event data pushed by other Suppliers is accomplished by using the overloaded `operator>>=` function, which attempts to extract our `Stock::CallbackInfo` struct from the event `any` data. If the event `any` does not contain an instance of `Stock::CallbackInfo`, the extraction fails, and the event can be ignored.

The exchange of stock value notification data through an untyped Event Channel is analogous to two parts of a C++ program exchanging the same data by converting pointers to `Stock::CallbackInfo` instances to `void*`, and then casting the `void*` back to `Stock::CallbackInfo` pointers. The errors associated with the use of `void*` in C++ are well known and widely documented, and similar types of problems can be encountered when using untyped Event Channels.

4.4.3 Workarounds

It seems that our attempt to separate concerns to simplify our `Notifying_Quoter` has resulted in a system with more problems than it solved. The severity of these problems can be reduced somewhat by making a tighter coupling between the `Notifying_Quoter` and its Event Channel. Specifically, we need to:

• **Use a specific registration interface:** Rather than relying on the untyped Event Channel interfaces for Consumer registration, we could completely hide the fact that an Event

Channel is being used by returning to our original (strongly typed) callback registration interface. This approach uses the Event Channel as a transport mechanism, in much the same way that networking services like FTP, TELNET, and HTTP use the TCP protocol as a transport mechanism.

- **Hide the Event Channel:** We could ensure that no other Supplier applications are using our Event Channel by making the `Notifying_Quoter` responsible for creating its own Event Channel. By using the appropriate access control mechanisms of C++, we can ensure that object references are not "leaked" to other Consumers and Suppliers.

- **Provide event filtering:** Once the Event Channel is hidden, we can register special filtering Consumers with it so that our real Consumers only see the callbacks in which they're interested. In practice, some implementations of Event Channels [7] extend the COS Events Service to add event filtering and correlation. More importantly, by the time this column is published, the OMG Telecommunications Domain Task Force will most likely have issued an RFP for a filtering Notification Service that uses an Events Service underneath it.

- **Reduce network traffic:** By making our private Event Channel and our filters local to our `Notifying_Quoter`, we can eliminate the additional network traffic resulting from the use of an untyped remote Event Channel.

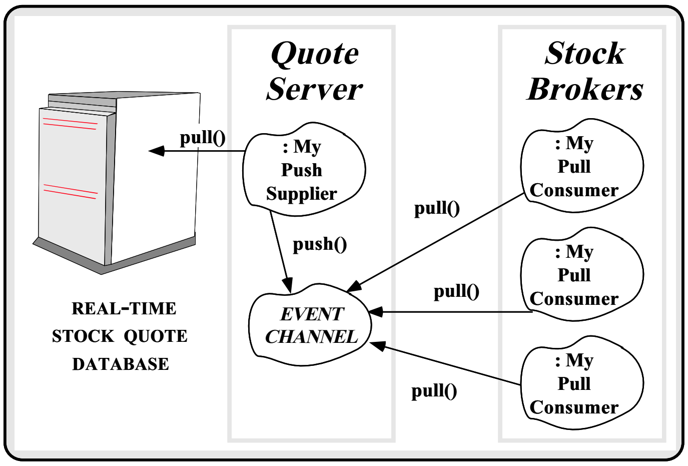

Figure 8: Alternative The Stock Quote System Design (Push/Pull Model)

- **Build an event queueing system:** Figure 8 illustrates an alternative design for the stock quote system. This design uses the Push/Pull model (described in Section 2.2), which queues up the events and allows clients to pull them at their leisure. This results in a potential decrease in network traffic since Consumers can now pull sequences of events instead of requesting them one at a time. In fact, this approach is similar to the Pull-based implementation described in our earlier columns. The Event Channel for such an approach could provide the normal COS Events interfaces for Supplier registration and the pushing of event data, but would require special interfaces for Consumers to allow them to pull sequences of events.

  The use of the Push/Pull model as described above could result in a decrease of network traffic. However, it could also increase the resource utilization of the Event Channel as compared to the Push model. This is because the Event Channel is responsible for storing event data until Consumers pull them. Since resources are finite, the Event Channel must implement some kind of policy or policies by which it throws events away if they are not pulled within some period of time or if a certain number of events accumulate. A high quality Event Channel implementing the Push/Pull model would allow its queuing policies to be configurable for each Consumer.

Our next column will provide an implementation of our stock quote system using some of these features.

5 Evaluating the OMG COS Events Service Specification

For a general events service, the COS Events Service model is quite reasonable. Its flexibility allows different implementations that make very different tradeoffs to be built, thus allowing application developers to avoid "one size fits all" implementations that rarely work well for anything. This flexibility allows applications to avoid responsibility for concerns such as timely notification, multiple registrations, and event data persistence and instead rely on Event Channels to handle such things for them. The Events Service interfaces are fairly simple to understand, and Consumer and Supplier connections and event delivery/procurement are symmetrical and straightforward. A side effect of this symmetry is that specialized Event Channels can be chained together, with one channel serving as either the Supplier or Consumer for another, for purposes of filtering or buffering events.

There are, however, several general limitations with the OMG COS Events Service specification:

- **Overly flexible:** Although the Events Service is highly flexible, it can be hard to use due to multi-step connection establishment process. The complexity stems from the need to use the Events Service within many different application domains, each with differing requirements for decoupling Consumer and Supplier communication. The fact that the Events Service supports the four different models of component collaboration described in Section 2.2 above is a clear indication of the flexibility of the specification. However, for many simple use-cases, this flexibility is overkill.

- **Lack of standard semantics and protocols:** The Events Service specification is intentionally vague, to avoid overconstraining the innovation and opportunity for optimization of implementors. This is beneficial to the extent that it keeps the specification concise and avoids forcing all users to pay for features (*e.g.,* transaction, persistence, filtering, etc.) that they don't need.

However, a downside to the underspecified nature of the COS Events Service is that there is no standard definition of how Event Channels will behave. This makes it difficult to compose Channels written by different vendors, whose filtering and queueing logic may be different. In addition, there no standard definition of the communication protocol used to implement inter-Channel communication.

For example, IONA has released an implementation of the OMG COS Events Service called OrbixTalk. OrbixTalk distributes events via IP multicast, using a negative acknowledgement scheme to ensure delivery to every interested Consumer. On the other hand, other implementations of the Events Service, such as the HP ORB Plus Events Service, are based on IIOP, with simple sequential delivery of event data to Consumers. The existence of Events Service implementations with different design centers provides application developers the opportunity to choose the implementation that best fits in their systems. However, it should be noted that due to their use of different protocols, the ORB Plus Events Service and OrbixTalk do not interoperate yet.

6 Conclusion

In this column we attempted to correct some deficiencies in our `Notifying_Quoter` stock quote callback system. It was our goal to utilize the OMG COS Events Service to separate the concerns of event delivery from the issues involved in monitoring stock quotes. Specifically, we hoped to use the Events Service to eliminate the need for our stock quote server to persistently store callback object references and handle issues related to scalability and deadlock. Unfortunately, our new solution introduces as many or more problems than did our original hand-crafted callback system. Our next column will show how some of the problems introduced by the solution developed here can be alleviated.

We'd like to note a change in our email address. Steve recently left Hewlett-Packard to become a Senior Architect at IONA Technologies, so our old email address, `object_connect@ch.hp.com`, has changed. Comments on this or any of our columns may now be sent to our new address, `object_connect@cs.wustl.edu`. As always, if there are any topics that you'd like us to cover, please send us email.

Thanks to Wolfgang Lugmayr for comments on this article.

References

[1] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching," in *Pattern Languages of Program Design* (J. O. Coplien and D. C. Schmidt, eds.), Reading, MA: Addison-Wesley, 1995.

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

[3] S. Maffeis and D. C. Schmidt, "Constructing Reliable Distributed Communication Systems with CORBA," *IEEE Communications Magazine*, vol. 14, February 1997.

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

[5] R. G. Lavender and D. C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming," in *Pattern Languages of Program Design* (J. O. Coplien, J. Vlissides, and N. Kerth, eds.), (Reading, MA), Addison-Wesley, 1996.

[6] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, 2.0 ed., July 1995.

[7] R. Stewart, J. Storey, and D. Huang, "Event Handling in a CORBA-based Telecommunications Management System Framework," *C++ Report*, vol. 9, February 1997.