Publish and subscribe services are useful for updating multiple client nodes with information from different sources. **Kersasp D. Shekhdar** proposes a new interface for publish and subscribe services in Corba

Subscribing to a P&S model

THE OBJECT MANAGEMENT GROUP'S CORBAservices (COS) EventService (ES) specification is written to please everybody—and nobody. The specification presents various modes for connection and event transmission and, as a result, allows a plethora of supplier-consumer models. Oddly, it simultaneously fails to develop any specification for a true publish/subscribe system. A publish/subscribe concept shouldn't incorporate the dissimilar notions of 'push' and 'pull'; there should only be one mode of event transmission. This article describes the publish/subscribe concept and proposes the `PAndSService` with interfaces and specifications.

In search of true publish and subscribe models

The fundamental premise behind classical client server systems is clearly visible in the ES specification and is encompassed by its pull-related syntax. It was with the advent of RDBMS that client server became a popular and well-understood paradigm that played a part in triggering the popularity of distributed computing. Essentially, intercomponent messages are always strongly directed from the client to the server.

As client server evolved, the callback concept was introduced in which the client calls the server and passes a reference of itself under certain cases. Later, the server may turn around and call the client that had passed the server its reference. Callbacks were a transitory step towards the later peer-to-peer paradigm. The messages are requests for some data that may or may not be available at that time, and so this type of message became known as 'request'. If the client did not get the data, it would keep making repeated requests; this is known as 'polling'. Client server architectures predate event-based architectures in which an event happens to an object and may cause a state-change and an associated output.

Client server models do not lend themselves well to an event-based system. There is no way for an interested party to know when an event occurred and, therefore, when to pull it. A third-party could find out if the specific event occurred upon some object by polling it as in a client server system, but this introduces two design problems. First, the third party will not learn of the event when it happened, but only when it makes a request, thus allowing the event to age. The age of the event when it is finally recognised could almost match the time interval between successive requests. The second problem is that the affected object must somehow retain the event until all interested third parties have recognised the event. It also introduces the practical drawback of redundant messages and, therefore, wasteful network traffic. Regardless of how the ES specification is implemented, if the pull syntax is in the mix then at least one of the above disadvantages is incurred.

It's worth pointing out that only events are relevant to this article because events are the only messages used in the publish/subscribe

FACTS AT A GLANCE

- **The Object Management Group's EventService and NotificationService specifications are inadequate as blueprints from which to produce a product of any reasonable use.**
- **These specifications are deficient in their Interface Definition Language operations partially because they neither provide nor build upon a meaningful model of any sort.**
- **A true event-based publish and subscribe model is presented, and the interrelationships and functions of its various components—among them publishers, subscribers and two kinds of event hubs—are described.**
- **Based upon the preceding model, we derive a new IDL-based service that is comprised of two structures and 14 operations within four compact interfaces. Slightly different flavours of an operation can be selected depending on need.**
- **We propose the IDL and description of this service, the PAndSService, as an addition to the CORBAServices.**

model. The other popular message types, notification and request, cannot be components of an event-based publish/subscribe service.

COS-ES examined

The opening presentation of the OMG documentation illustrates the `CosEventComm` module and event-based semantics with a tightly-coupled design, but this has nothing to do with an event-based publish/subscribe model. It is no more than a representation of two software processes in which one is directly messaging the other. Now consider the decoupled model that includes the event channel. The pull model is simply a semantic layer over the classical client server module.

Now consider COS-ES-defined suppliers and consumers decoupled by an event channel and communicating through the push variants declared in the `CosEventComm` module. One of the main problems is that the operation's parameters have no discriminator to identify which event occurred. Indeed, there is no way for a consumer to subscribe to one or more particular types of events. The consumer will receive all events without discrimination from whichever event channels it has attached to. Though the ES certainly does not dictate an implementation design, considering our various implementation design alternatives, we still run into trouble. These are the alternatives:

1. **Inherit from the specified interfaces and refine the push and pull variants by overriding, therefore discriminating between events.**
   This approach alters the COS-ES design at its most basic levels, which is an implicit acceptance that a severe enough problem exists to prevent its usage. Instead, one might as well model and design a better alternative from scratch.
2. **Allow different kinds of events to be posted to one event channel and different kinds of events will therefore be sent to the consumer. The consumer takes the responsibility to inspect the 'any' data component of the event to determine the kind of event received and discard those that are of no interest.**
   This impedes performance and efficiency because unnecessary messages are sent from the event channel to the subscribers and the subscriber has to inspect even the unwanted ones. The second objection is that it doesn't make sense to send events to subscribers even if they have no interest in them.
3. **Implement a design in which an event channel instance is dedicated to one type of event.**
   This is also inefficient. If there are *n* types of events, it would dictate *n* event-channels; moreover, it also imposes undue costs in terms of the instantiation and existence of objects. Consider: if there are *o* publishers and *p* subscribers, it would dictate instantiation of $n*o + n*p$ admin objects and the creation and maintenance of the same number of proxy objects. And it is quite possible that traffic for some events may be so low (e.g. about five per day) that some channels would be drastically underused.
4. **Use the `DynAny` type as the data type that is used to send the event data and have the event channel filter the subsequent forwarding to consumers based upon the specific `DynAny`.**
   Here, the event channel has to inspect the undefined `DynAny` of every incoming event instead of merely a single discrete property. Furthermore, the overkill of introducing a new data type to solve a COS design problem only alleviates the symptoms.

Combining these four options to arrive at an implementation design does not eliminate any of the disadvantages, but they could be minimised, given a specific usage by an exact setup of event suppliers and event consumers. So the ES specification adds new semantics, defines a decoupling and, of course, allows interface declaration via the Interface Definition Language (IDL). But it does not address the publish/subscribe problem and nor does it address event-based communication because the interfaces are not derived from any underlying model. The specification does not carry any significant value and is therefore deficient. Finally, the `NotificationService` is also a disappointment, because not only are many of the same problems present, but a few of them are magnified. New concerns have also been introduced.

Implementation vs design

Given the drawbacks and failures of COS-ES, we present a publish/subscribe model, build a set of IDL operations on it and provide a specification.

Components:
1. Event hub
2. Event publisher(s) (hereafter 'publisher')
3. Event subscriber(s) (hereafter 'subscriber')

Components for implementation are nearly always constructed one-to-one from the design. This practice often limits an implementation, because the implementation must only follow from the design—it does not have to be the concrete representation of the design.

Consequently, we shouldn't assume that the event hub implies a single object or process, as this could leave us with an inferior implementation that could suffer from performance problems. The design specifies the presence and responsibilities of something, but the creation, maintenance and arrangement of that thing are implementation design decisions. The IDLs below do not contradict the preceding statement, nor do they detract from the goal. But as far as the publishers and subscribers know, only the event hub exists.

How publish and subscribe works

In the publish/subscribe system, both publishers and subscribers first obtain a reference to the event hub. The precise way in which this is done may vary by implementation. A single implementation may provide a choice of methods for doing so. Publishers then post events to the event hub, regardless of whether or not there are any subscribers. Such information is superfluous to the publisher. An event hub may mandate that publishers must first register with it and subsequently make themselves known upon each posting using an agreed identification. Subscribers subscribe to the event hub for one or more events. When the subscribed-to event occurs, the event hub messages the subscriber with the properties of that event (see `Event.idl`). Any client of the event hub may be a publisher as well as a subscriber.

When a publisher posts an event, its attributes consist of the event name, its generator's name and its data. The event hub may make internal use of the generator name, say for logging and statistics

purposes. When the event hub forwards an event to subscribers, the event's attributes are a unique ID in the form of an alphanumeric sequence, and then the event name and event data. These are commonly considered to be the three components of an event. Hence, the structure types `Event_In_t` and `Event_Out_t`, which respectively define events generated by publishers and the counterpart events forwarded to subscribers.

Events are transitory and are to be despatched instantaneously by the event hub. By definition, events are not persistent. They are a point-in-time occurrence, and so this specification does not provide for event persistence. Even persistence, however, is a matter of pragmatics. It may not be possible for some subscriber to always be up, yet they may require receipt of all events. Hence, event persistence may be included as a value-added feature by an implementer to his `PAndSService` product in any way seen fit. Doing so would not impact the interfaces or their specifications.

The remainder of this article addresses the `PAndSService` interface. Similar to event persistence, ancillary properties such as event lifetimes and levels of service are neither mandated nor discussed, although the interfaces account for them. The reason is that these too are implementation design or value-added features, and so their definition is not relevant to this discussion. From an object orientation standpoint, it would be sensible to have integral and reusable IDLs for these properties so that the same interface to control them could be available in a variety of services.

Describing the PAndSService interfaces

There shall be two types of event hub within the `PAndSService`: one that implements the `EventHub` interface and another that implements the `RegistrarEventHub` interface. The latter requires event publishers to successfully register before posting events. Through registration, it imposes a form of validity checking on publishers. A `PAndSService` implementation shall include both because the type of event hub that would be used in a specific publish/subscribe setting depends on the end user's needs. This choice would be made by the systems designer and both interfaces support the essential publish/subscribe paradigm.

Listed below are the behaviours of the operations (with specifications and requirements), along with rationales, where necessary or helpful.

The EventHub interface:

This interface is implemented by the event hub and its operations are invoked by publishers and subscribers.

`attempt_post()` is a one-way invocation that shall not block the caller and the event hub wouldn't return any status of event delivery. The return value in `post()` blocks the invoker and allows the event hub to return a value to indicate correctness of parameters and successful receipt. It may raise a `SystemException`. Other than that, it shall not raise any exceptions to any event publisher. All event publications must be accepted.

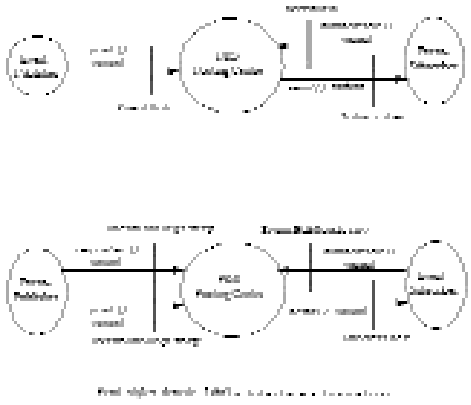
A subscriber may request that events of particular type(s) be despatched to it by invoking either `subscribe_to()` or `subscribe_to_events()`. In the first operation, it subscribes to one type of event. Using the second, it can subscribe to multiple event types in one shot. Additionally, the subscriber informs the event hub how it wants to receive events. If the boolean `oneway_attempt` is `true`, the `Subscriber::attempt_send()` operation shall be invoked for quick and dirty despatch. If it is `false`, `Subscriber::send()` shall be invoked in which exceptions can be raised and a boolean status can be returned for guaranteed delivery.

At request for subscription, the event hub does not check for validity of the event name because it cannot know if and when some publisher at some future point might start publishing the subscribed-to event. However, in the case of `subscribe_to()`, it shall indicate to the subscriber whether or not the event is `currently_publishing`. An implementer of this service should decide and document how fresh an event must be for `currently_publishing` to evaluate to `true`. Compare the same syntax in the `RegistrarEventHub` interface, where it has a well-defined and more sophisticated semantic.

The RegistrarEventHub Interface:

All publishers using this interface must register with a unique `publisherId` and with their intent to publish specific events before the event hub will accept any event postings. The event hub also maintains a list of all registered publishers and the events they can post. The subscription operations are inherited directly, although one is implemented differently.

Publishers may invoke either variation of `register()`. These are to register a new publisher along with the event names it will publish, as well as to register new event names for a previously registered publisher. The publisher passes in a `publisherId` string, which is the unique tag by which it identifies itself and which it shall include in all subsequent calls. The first time a publisher registers, it shall pass `true` in `newPublisher`. If a pre-existing publisher is registering new event names, it shall pass `false`. If the event hub receives a registration by a new publisher, the return value shall be `true` if registration was successful, `false` otherwise. A `false` value may indicate that the requested `publisherId` string is already in use, in which case the event hub shall substitute another tag in the same parameter as a suggestion. The publisher can try to register again using the suggested tag or with something else of its own choosing. If the event hub receives a registration from a pre-existing publisher (`newPublisher = false`) and that publisher is indeed registered, it shall add the passed event names to any that the publisher had previously registered and return `true`. If a registration is received from a publisher that claims to be pre-existing but is found not to be so, the event hub shall take no action and shall return `false`. In the singular operation,



Schematic of communication between the posting-centre, event-publisher and event-subscriber

one event name is supplied. In the plural, multiple event names may be registered in one shot. If a publisher registers the same event name again after having successfully registered it, nothing shall be done; the subsequent request is simply ignored.

The inherited `post()` operations are overridden. They include the added parameter `publisherId`. If the tag is unknown or is not registered for `theEvent.name`, the appropriate exception shall be raised. Its purpose is not so much for strict authorisation, because it can be subverted by a publisher who wants to do so. Rather, it is meant to safeguard against errors and track publisher activity. Furthermore, monitoring and statistics may be desirable within a publish/subscribe system.

A `de_register()` operation shall be invoked by a publisher when it will no longer be publishing one or more events. In the singular variant, the publisher identifies itself in `publisherId`, along with the event it is deregistering. If a publisher object is de-allocating, the publisher process is terminating, or it has finished publishing events for any reason, it may invoke `de_register_events()`. Upon receiving this invocation, the event hub shall deregister that publisher for all the events it had registered for. If a deregister notification is received with an unknown `publisherId` string, the `UnknownClient` exception is raised.

If a known publisher attempts to deregister for an event that it is not currently registered for, an exception shall be raised. Indeed, if it so happens that a publisher deregisters a second time for an event, it may seem that it isn't necessary to raise an exception because the event hub could simply ignore the redundant notification. However, the event hub is not required to maintain the states of publishers. Therefore, the event hub cannot know if the publisher had never registered in the first place for the event it is asking to be deregistered for. Such a scenario would imply erroneous programming within the publisher, and so `NotRegisteredForEvent` is raised upon the described condition.

Unlike `register_publisher()` operarations, `subscribe_to()` operations do not accept or return a subscriber ID. This is because, in the case of publishers, some artefact is needed for identification purposes only and a tag serves the purpose. But in the case of subscribers, a reference to the actual object is necessary for invocations. The object itself may as well serve the purpose of a tag also, considering the reference will not be passed redundantly or in frequent messages.

An extra operation, `registered_Events()`, is made possible in this interface because publishers register their intent to publish events *a priori*. This function shall return a set of all event names currently registered. The same event name registered by *n* publishers shall cause *n* occurrences of that event name in the returned list. If no events are currently registered, an empty list shall be returned. This operation may be used by a subscriber (actual or potential) or a statistical process to check currently publishing events at any time.

The remaining specifications for the subscribe and unsubscribe operations are analogous to those for the deregister operations. If a subscriber subscribes to the same event it already subscribed to, nothing is done; if an unsubscribe invocation contains an unknown `Subscriber` object, a `UnknownClient` exception is raised. If a known subscriber attempts to unsubscribe to (an) event(s) it is not currently subscribed to, a `NotRegisteredForEvent` exception is raised (the rationale is the same as that given above for a publisher deregistering an event it is not registered for). Finally, in the case of `subscribe_to()`, the `currently_publishing` 'out' parameter has a different semantic than the same syntactic element in the superclass interface because of certainty in this interface. The event hub shall return `true` if at least one publisher presently connected has registered the event the subscriber has requested subscription to, and `false` otherwise.

The subscriber interface

The subscriber interface must be implemented by the subscribers within the application system and its operations shall be invoked by the event hub. The two variations of `send()` are analogous to the `post()` operations—one blocks and has a return value, while the other does not block and has no return value. They each have their advantages and disadvantages. Considering the realities of a subscriber failing to handle an event for any internal reason related to programming or context, the subscriber may raise a `Failed` exception. The event hub shall catch this exception and later resend the event once more to that subscriber. The elapsed time is dependent on preset configurations. Besides this, exception `Disconnected` may also be raised. Upon catching this exception, the event hub shall automatically unsubscribe the subscriber for all events. It is a subscriber's responsibility to subscribe to events each time it connects.

In conclusion, we submit that these few interfaces with a handful of operations allow an orderly but flexible way of implementing a true publish/subscribe system by providing a feature-complete interface set and specification, while leaving out extraneous superfluities. ■ *Acknowledgements: Michael Gentry*

Listing 1: PAndSService IDL interfaces

```
/* Begin Event.idl  */
interface Event  {
struct Event_In_t  {
               string   name;
               string   generator;
               any              data;
};
struct Event_Out_t  {
       string   id;
               string   name;
               any              data;
};
```

```
};
/*  End Event.idl  */

_____________________________________________________


/*  Begin Subscriber.idl  */

#include Event.idl

exception Failed {};

interface Subscriber  {
       oneway void attempt_send(in Event_Out_t anEvent);
       boolean send(in Event_Out_t anEvent) raises (Disconnected, Failed);
};
/*  End Subscriber.idl  */

_____________________________________________________


/*  Begin PAndS.idl  */

#include Event.idl
#include Subscriber.idl

module PAndS  {
typedef sequence <string> strings;

exception UnknownClient {};
exception NotRegisteredForEvent {};

interface EventHub  {
       oneway void attempt_post(in Event_In_t theEvent);
       boolean post(in Event_In_t theEvent) raises (SystemException);

       boolean subscribe_to(in Subscriber recipient, in string anEventName, in boolean oneway_attempt, out boolean
currently_publishing);
       boolean subscribe_to_events(in Subscriber recipient, in strings eventNames, in boolean oneway_attempt);

       boolean unsubscribe_for(in Subscriber recipient, in string anEventName) raises (UnknownClient,
NotRegisteredForEvent);
       boolean unsubscribe_for_events(in Subscriber recipient, in strings eventNames) raises (UnknownClient,
NotRegisteredForEvent);
};

interface RegistrarEventHub : EventHub  {
       boolean register(inout string publisherId, in boolean newPublisher, in string anEventName);
       boolean register_events(inout string publisherId, in boolean newPublisher, in strings eventNames);

       boolean de_register(in string publisherId, in string anEventName) raises (UnknownClient,
NotRegisteredForEvent);
       boolean de_register_events(in string publisherId) raises (UnknownClient);

       oneway void attempt_post(in string publisherId; in Event_In_t theEvent) raises (UnknownClient,
NotRegisteredForEvent);
       boolean post(in string publisherId; in Event_In_t theEvent) raises (UnknownClient, NotRegisteredForEvent);

       strings registered_Events();
};
};
/*  End PAndS.idl  */
```