Events and service-oriented architecture: The OASIS Web Services Notification specifications

P. Niblett
S. Graham

The OASIS Web Services Notification (WSN) family of specifications defines a standard interoperable protocol through which Web services can disseminate events. We present here a summary of three WSN specification documents that are currently available: WS-Base Notification, WS-Topics, and WS-Brokered Notification. We conclude with a brief discussion on the use of the notification pattern in the Enterprise Services Bus, a service-oriented infrastructure for mediating requests among cooperating Web services.

INTRODUCTION

Many service-oriented architecture (SOA) implementations are based upon the *request/response* interaction pattern, where a service requestor identifies a service that it wishes to use and then sends it a request message. A second entity, the service provider, accepts the request message, processes it, and then sends a response message. This is a pattern that is familiar to any programmer who has made a procedure or function call in a procedural programming language or who has invoked a method in an object-oriented language or distributed object system. Indeed this pattern is so familiar that programming interfaces and tools (for example, those used with Web services) often hide the underlying message exchange; these tools present a programming model that looks like a simple procedure call.

Event-based programming, which has been around for many years and has been applied in many areas, is frequently used in user-interface systems, for example the Smalltalk Model-View-Controller (MVC) paradigm[1] or the X Window System,[2] where a change in a model can be reflected in various views, or where components react to user interactions such as mouse clicks or key presses. Support for event-based programming is provided in *publish/subscribe* systems available from message-oriented middleware vendors.[3] Event-based programming is also used with distributed objects; Object Management Group, Inc. has published two

CORBA** (Common Object Request Broker Architecture) specifications that relate to event-based programming: the Event Service Specification[4] and the Notification Service Specification.[5]

Event-based programming features an entity that represents an occurrence (something that has happened). In object-oriented systems this is usually termed an event object; in message-oriented systems it is variously referred to as a message, event, or event message. In contrast to the request/response pattern where the request and response messages are frequently hidden from the programmer, in event-based programming the event (be it a message or object) assumes center stage. Applications explicitly produce and consume events, and the producing application has a relationship with the event that it produces, rather than a direct relationship with the applications that consume the event. A consumer of events indicates (through a registration process) the events in which it is interested, and it interacts with the event itself, rather than with the application that produced the event. We use the term *notification pattern* to refer to the interaction pattern that involves registration of consumers and subsequent dissemination of events.

The idea of using the event itself to decouple the event producer and consumer is a significant difference between the request/response pattern and the notification pattern. This decoupling supports one-to-many and many-to-one message exchanges, in addition to the one-to-one exchange found in the request/response pattern. In addition, it may have a more natural fit to the real-world scenario that is being modeled by the application architecture, and it allows further complex event processing to be added in a straightforward fashion.[6]

The OASIS Web Services Notification (WSN)[7] family of specifications defines a standard interoperable protocol through which Web services can disseminate events. These specifications are being developed by OASIS (the Organization for the Advancement of Structured Information Standards), a not-for-profit international consortium that drives the development, convergence, and adoption of e-business standards. The specifications are authored by an OASIS technical committee whose membership comes from a variety of software vendors, users, and other professionals.

The intent of WSN is to define a set of royalty-free, related, interoperable, and modular specifications that allow the notification pattern to be modeled in an explicit and standardized fashion. The benefits of such standardization include interoperation between application entities written by different authors, as well as interoperation between different publish/subscribe messaging middleware providers. The WSN family is made up of four separate specification documents.

The *WS-Base Notification* specification[8] defines the Web Services interfaces for notification producers and notification consumers. It includes standard message exchanges to be implemented by service providers that wish to act in these roles, along with operational requirements expected of them. This is the base document on which the other WSN specification documents depend.

The *WS-Topics* specification[9] defines a mechanism to organize and categorize items of interest for subscription known as *topics*. These are used in conjunction with the notification mechanisms defined in WS-Base Notification. WS-Topics specifies an XML model for describing meta-data associated with topics, and it defines some topic expression dialects that can be used to refer to them.

The *WS-Brokered Notification* specification[10] defines the Web Services interfaces for notification brokers. A notification broker is an intermediary which, among other things, allows publication of messages from entities that are not themselves service providers. It includes standard message exchanges to be implemented by notification-broker service providers along with operational requirements expected of service providers and requestors that participate in brokered notifications.

The *WS-Notification Policy* specification defines a set of policy statements that can be used in conjunction with the other specifications in the family to request particular qualities of service or other behavior.

The first three of these specifications are currently available as drafts and will be the focus of this paper. The WS-Notification Policy specification is currently still at an early stage of development.

In the next section, "Web Services Notification," we present a summary of the first three WSN specifications. A discussion section follows in which how event-based interactions can be used in SOA alongside interactions based on the request/response pattern is discussed. In particular, the use of events within the Enterprise Services Bus (ESB)[11] and in the context of Complex Event Processing (CEP) is discussed.[6]

WEB SERVICES NOTIFICATION

WSN specifications standardize the syntax and semantics of the message exchanges that establish and manage subscriptions and the message exchanges that distribute information to subscribers. An information provider, known as a notification producer, that conforms to WSN can be subscribed to by any WSN-compliant subscriber. If subscriber and producer are using a common Web-service binding—for example SOAP (Simple Object Access Protocol)/HTTP (HyperText Transfer Protocol)—and have appropriate network connectivity, they could in principle interoperate even if they had been designed by different people and were running in different organizations on different continents.

We start our review of WSN with a section on the terminology used in the specifications. Then, we describe in sequence WS-Base Notification, WS-Topics, and WS-Brokered Notification.

To illustrate aspects of the WSN specifications, we use two example scenarios. The first is a stock-trading scenario in which the service Stock Feed is provided to a number of stock-trading applications. The Stock Feed service supplies a stream of messages indicating a change of price in some traded stock or instrument. The trading applications (possibly automated, possibly involving a human trader) receive these messages and react to them. Our second example is from the systems-management world, where an organization has deployed some printer management software to manage and monitor the printers that it owns. The printers in the organization generate events when they encounter particular situations, both normal and abnormal, and these are monitored by the printer management software.

Terminology and concepts

WSN defines a set of terms, the most important of which we list in this section and use throughout the paper. The terms defined here are intended to eliminate certain inconsistencies that used to plague discussions related to events. For example, the term "subscriber" sometimes referred to the entity that received notifications, sometimes to the entity that set up the subscription, and sometimes even to the entity that paid for the service. The specifications avoid using the term *event* because this word is susceptible to multiple interpretations.

Situation—A situation is an occurrence (something has happened) that is noted by one party and is of interest to other parties. A paper jam in a printer and a sports result are two examples of situations. Often a situation reflects a change of state of some object, such as a stock-price change, a temperature change, or a change in the internal state of a running software program. Although the type of occurrence related to the situation is immaterial as far as WS Notification is concerned, it is important that information relating to it can be communicated to other services.

Notification—WS-Notification uses this term to refer to the one-way message that conveys information about a situation to other services. The sender of a notification message could choose to format this information in whatever way it sees fit and could even use a different representation for each time the situation occurs. To keep things simpler for receivers, the sender of information typically chooses a specific message type for each kind of situation that the receiver is interested in. The type of message specifies the information items that it contains; it may also specify the format of this information as a sequence of bytes. In WSN a message type is represented by an XML Schema[12] global element definition.

The association between a situation and the type of corresponding notification message is not necessarily one-to-one. It is possible that an application might associate several different notification message types with a given situation. This could be the case if there are multiple receivers and the aspects of a situation that are of interest to the receiver vary from receiver to receiver. Consider, for example, a "new employee hired" situation. A payroll application requires the employee's name, serial number, job level, and starting salary, whereas a physical security application requires the employee's office location.

Conversely, the same message type could be used for a variety of situations. For example, a generalpurpose error message type could be used for a number of different kinds of error situations.

Publisher—An entity that creates notification message instances. The publisher selects the appropriate type of notification message for the situation and constructs an instance of this type containing information relevant to the situation. In some cases a publisher may be reacting to an external situation, in which case its job might simply be to reformat data from the external source into the format dictated by the notification message type. The publisher does not have the responsibility for sending the message to the appropriate receivers (see *Notification Producer*).

Notification Producer—A service that is responsible for sending notifications to the appropriate consumers. In some cases, a notification producer also assumes the role of publisher and is responsible for detecting situations and creating message instances. If the notification producer does not act as publisher, it is referred to as a *notification broker* (or broker, for short) and does not actually create notification messages, but instead manages the notification process on behalf of one or more publishers.

The notification producer is responsible for maintaining a list of interested consumers and arranging for notification messages to be sent to those receivers. This may involve a matching step that compares each notification against the interests expressed by the individual consumers.

Notification Consumer—The counterpart of a notification producer, an entity that receives the notifications distributed by notification producers. The most common kind of consumer is a *push consumer*, which is able to receive notifications sent directly from the notification producer. WS-Notification also supports *pull consumers*, which interact with the notification producer (or some intermediary) when they wish to receive a notification. A pull consumer might be behind a firewall, which prevents it from operating as a push consumer.

Subscription—An entity that represents the relationship between a notification consumer and a notification producer. It records the fact that the notification consumer is interested in some or all of the notifications that the notification producer can provide. A subscription can contain filter expressions, policies, and context information. Each notification producer holds a list of active subscriptions, and when it has a notification to send, it matches this notification against the interest registered in each subscription in its list. It determines the set of consumers that are interested and notifies them.

A subscription may be long-running, in which case it lasts as long as the notification producer does, or it may have a limited lifetime. In loosely coupled environments such as SOAs, it is often desirable to apply a finite lifetime to a subscription so as to avoid situations where consumers disappear or lose interest in a subscription without canceling it.

Subscriber—Although subscriptions may be defined statically as part of a system design, event-driven architectures typically involve dynamic subscriptions. WS-Notification uses the term *subscriber* to refer to an entity that requests creation of a subscription. A subscriber creates a subscription by sending a *subscribe request* message to a notification producer. This subscribe-request message identifies a notification consumer. If the notification producer is willing to accept this request, it creates a new subscription and adds the subscription to its list of active subscriptions. The notification producer can then start sending relevant notifications to the notification consumer.

Note that a subscriber may play roles of both consumer and subscriber by subscribing on its own behalf. WS-Notification separates the two roles to allow third-party subscriptions, in other words, to allow a service to create a subscription on behalf of a separate notification consumer. Although the notification consumer is required to be a service provider, the subscriber need only be a service requestor.

Subscription Manager—Once a subscription has been created, it is possible that the subscriber, the notification consumer, or even some third party may inquire about the subscription properties, may delete the subscription, or may renew the subscription (in the case where the subscription had a limited lifetime). The subscription manager is a service that manages requests to query, delete, or renew subscriptions. Each subscription manager is subordinate to the notification producer that owns the subscriptions in question. It is possible for a single service to take both the subscription manager and notification producer roles. WS-Notification distinguishes the roles to enable a notification producer to delegate subscription management to another service.

These various roles are illustrated in *Figure 1*. This figure shows a subscriber making a subscribe request to a notification producer on behalf of a notification consumer. As a direct result of this request, the notification producer adds a subscription to its list of subscriptions and sends a response to the subscriber. The list of subscriptions is represented by the scroll; each subscription entry records the notification consumer (NC1 in this case) along with other properties of the subscription. In this figure these other properties are shown in stylized form as "xxx, yyy...", but in practice they include things like the termination time of the subscription and any filter expressions associated with it.

At some later stage, the publisher detects a situation (bottom left), and the notification producer sends a notification to the notification consumer.

WS-Base Notification

WS-Base Notification provides the foundation for the WSN family of specifications. It defines the basic roles and message exchanges needed to express the notification pattern. The specification can be used as it stands, or it can be used in combination with the WS-Topics and WS-Brokered Notification specifications in more sophisticated scenarios.

The specification defines the message exchanges between four of the roles that we described in the section "Terminology and concepts": notification producer, notification consumer, subscriber, and subscription manager.

Figure 2 illustrates the stock trade scenario, a simple one-to-many scenario that involves *direct* notification; that is the producer, the Stock Feed service, also assumes the role of publisher.

Figure 3 illustrates the printer management scenario, which involves direct notification with multiple producers and a single consumer. There is a notification-producer Web service representing each printer in a department or organization and a single manager program that is monitoring the printers and is acting as the notification consumer. Each printer has a set of states (offline, printing, out of paper, paper jammed, etc.) and generates notifications when its state changes. The monitoring application subscribes to each printer in the department. It can then monitor the state of each printer by receiving a notification from that printer when its state changes, rather than having to continually poll each printer.

Any Web service can act as a notification producer, but in order to do so it must meet the following requirements, in addition to any other interfaces or functions that it may provide:

1. It must support the subscribe message exchange defined by WS-Base Notification.
2. It must send a notification to each notification consumer that has a subscription registered with it whenever it has a message to deliver and any

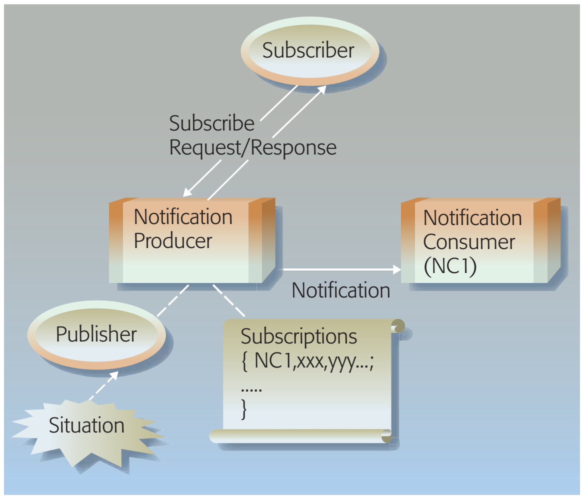

Subscriber
Subscribe Request/Response
Notification Producer
Notification Consumer (NC1)
Notification
Publisher
Subscriptions { NC1,xxx,yyy...; ..... }
Situation

Figure 1
Web Services Notification entities

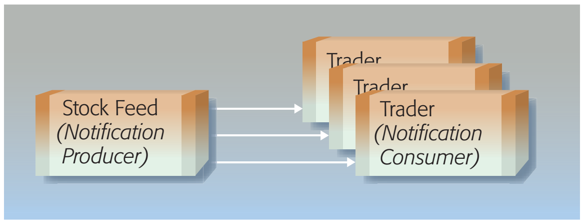

Trader
Trader
Trader (Notification Consumer)
Stock Feed (Notification Producer)

Figure 2
Direct notification with a single producer and multiple consumers

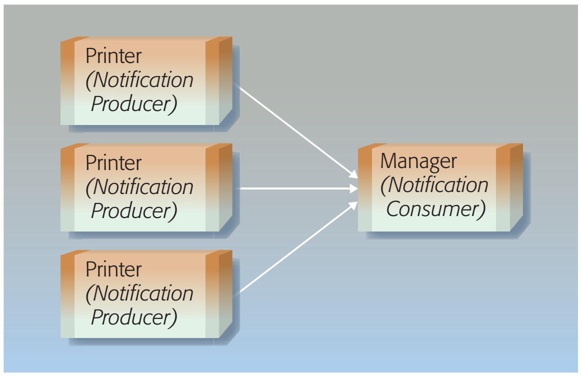

Printer (Notification Producer)
Printer (Notification Producer)
Printer (Notification Producer)
Manager (Notification Consumer)

Figure 3
Direct notification with multiple producers and a single consumer

filter conditions expressed in the subscription are satisfied.

In addition it may support a message exchange that allows other services to determine the set of topics (if any) that are supported by the notification producer. We discuss this in the section "WS-Topics."

The first requirement means that the Web Services Definition Language (WSDL) specification must include in its `portType` definition an operation that contains the subscribe request and response message defined by WS-Base Notification. The specification provides an XML Schema that defines these messages as global elements in the Base Notification XML namespace and, as a convenience, provides a WSDL `portType` definition that uses them. The author of a notification-producer Web service can cut and paste this `portType` definition (or just the operation from the `portType` definition if preferred) into the WSDL for his or her service.

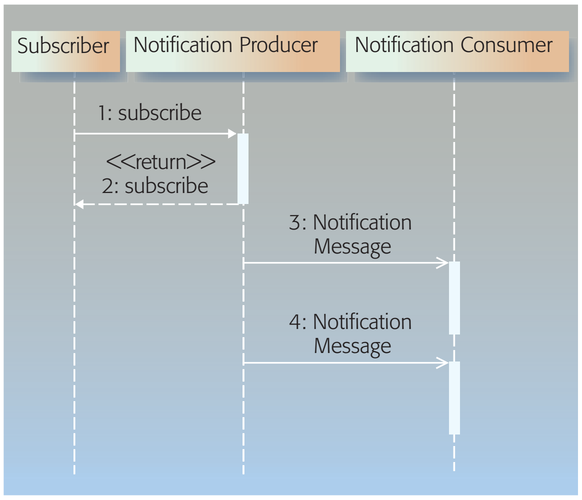


Figure 4
Message flows: creating a subscription, notifying the consumer

Figure 4 shows the message exchanges that take place when a subscriber creates a subscription and when the notification producer sends a few notification messages to the consumer. The subscriber sends a subscribe request message (1) to the notification producer. This is in effect a "control message" whose format is defined by the WS-Base Notification specification. The request message includes the address of the notification consumer, encoded as an endpoint reference as defined by WS-Addressing.[13] The request may also include filter expressions that constrain the kind of notifications that the subscription is to cover. In response to this message, the notification producer creates a subscription resource and sends a subscribe response (2) which contains another endpoint reference. This endpoint reference refers to the subscription itself (more about what this means shortly).

At some later time, the notification producer may detect a situation and issue a notification. This happens twice in our example—interactions (3) and (4).

WSN allows the notification producer to send the notification message in one of two formats:

1. It can use an application-specific one-way message exchange, defined as part of the notification consumer's `portType`. In our printer example, the monitor application might support a number of printer-specific messages, one for busy, one for error, and so on.
2. It can use the notify message format defined by WS-Base Notification. This is an envelope or wrapper message format. It contains the application-specific message along with other control information, such as the endpoint reference of the subscription. The format of the notify message allows the notification producer to package several application-level messages into a single Web service message.

The subscriber may attach a "policy" to the subscribe request to indicate which of these formats the notification producer is to use for the particular subscription.

WS-Base Notification does not specify the details of how notification message instances are created and does not define any interface between a publisher and the notification producer. As far as the specification is concerned, this is hidden behind the notification-producer interface. It is possible that the notification-producer Web service itself takes responsibility for matching notifications to its list of subscriptions and for sending copies of the message to the relevant consumers. Alternatively, the producer service might delegate this work to some other entity, for example, utility classes provided by the application server that is hosting it or a separate notification broker service. We discuss notification brokers in a later section.

Subscription filtering

The simplest form of a subscribe request message just contains an endpoint reference for a notification consumer. This form of request instructs the notification producer to send each and every notification that it produces to the notification consumer. This is satisfactory if the notification producer only produces a limited variety of notifications (for example, if it only detects a single kind of situation), but in more complex cases it has the following disadvantages:

- It is an inefficient use of resources to send messages that the consumer is not interested in.
- The consumer might not be able to understand the format of certain messages. Over time a producer might add support for new types of notification for use by other consumers. The original consumer might not be able to process these new messages successfully.
- The producer might wish to control access to the information in some of its notifications, while allowing open access to others.

To address these concerns, the subscribe request message can optionally contain one or more *filter expressions*. The filter expressions indicate the kind of notification that the consumer requires by restricting the kinds of notification that are to be sent for this subscription. This is on a subscription-by-subscription basis; a given notification producer may have several active subscriptions each with different filter expressions. Moreover, a notification consumer can be the target of multiple subscriptions, each with different filter expressions.

WS-Base Notification defines the following three kinds of filter expression (although implementors are free to augment this set with filter expressions defined outside the standard):

1. *Topic filters*—These provide a convenient way of categorizing kinds of notification. A topic filter excludes all notifications which do not correspond to the specified topic or topics. We will return to the subject of topics in the section on WS-Topics.
2. *Message filters*—This is a Boolean expression evaluated over the content of the notification message—for example, *Payment/amount* > 1000. A message filter excludes all messages that do not evaluate to true.
3. *Producer state filters*—These filters are based on some state of the notification producer itself—for example, `DebugMode=ON`, or `Day=Tuesday`—that is not carried in the message (so a message filter cannot be used). In order to use this kind of filter expression, the subscriber needs to know something about the properties of the notification producer.

Each filter expression evaluates either to True or False; a notification producer only sends a notification to the consumer of a subscription if all the filter expressions evaluate to True.

Subscription manager

We mentioned earlier that when a notification producer accepts a subscription request, it returns an endpoint reference in its response to this request, and we loosely referred to this as a reference to the subscription. The Web service whose address is carried in the endpoint reference is in fact a subscription manager. You will recall our definition of a subscription manager as a service that allows a service requestor to query, delete, or renew subscriptions. The subscription manager provides this query capability by supporting a number of resource properties that return, for example, the subscription's filter expressions, the consumer endpoint reference, and the scheduled termination time.

The subscription manager is an example of a "stateful" Web service, as described in Reference 14, and makes use of WS-Resource Properties and WS-

Resource Lifetime. For more information about resource properties, see References 14 and 15.

Subscription lifetime

In a loosely coupled environment, particularly in an Internet-based deployment, it is possible for an application to submit a subscription request, accept messages for a period of time, and then simply disappear. What should a notification producer do in such circumstances? Should it continue to keep the subscription active just in case the notification consumer reappears? To help answer these questions, WS-Base Notification allows a notification producer to support a time-based expiration scheme. The subscribe request message contains an initial-termination-time parameter. This can take one of the following forms:

- An absolute termination time, in Coordinated Universal Time (UTC), at which the subscriber wishes the subscription to end
- The duration relative to the current time, for which the subscriber wishes the subscription to last
- A special value (nil) meaning that the subscriber does not wish to set a termination time and would prefer the subscription to exist indefinitely

On receipt of the subscribe request, the notification producer decides whether it can honor the request. If it cannot, it rejects the entire request. If it can honor the request, then it sets an initial termination time for the subscription that is at least as long as the time requested by the subscriber. When this time has been reached, the notification producer is free to delete the subscription and stop sending any related notifications.

A notification producer may also allow its subscriptions to be renewed. If it does, then a subscriber is free to request an extension of its subscription at any time by sending a renewal request to the subscription manager. The subscription manager is free to accept or reject the renewal request. If a subscriber wants to ensure that its notification consumer does not miss messages, it needs to renew the subscription before it expires.

This mechanism allows the notification producer a reasonable degree of control. Suppose a producer wishes to impose a requirement that a given consumer check in (provide a "heartbeat") every five minutes. This could be done by accepting only subscription and renewal requests that contain termination times less than five minutes in the future. This means that the subscriber has to issue a renewal request every five minutes or risk losing the subscription.

In addition to this time-based approach, subscription managers also support an explicit subscription-deletion message exchange.

WS-Topics

We briefly introduced the concept of topics in the discussion of filters in the section "WS-Base Notification." In WSN, topics are described in the WS-Topics specification. In summary, a topic is a concept used to categorize kinds of notification and their associated notification message types. Topics are used in WSN to provide the following:

1. A straightforward way for a subscriber to indicate the kinds of notification or the underlying situation in which it is interested. A subscriber does this by supplying a topic filter rather than a filter specified in terms of the message body. This allows more flexibility as topic filters are not tied to the notification message. In particular:
   - The name of the topic might not appear in the message itself;
   - More than one message type might be associated with a given topic. (In our stock news-feed example, we could have a topic for each ticker symbol but have a subscription against a given symbol give rise to several different types of notification message.)
   - A given message type might be associated with more than one topic; for example, the Common Base Event[16] specification defines a common message format that may be used to express a wide range of error, trace, management, and business events.
2. A way for a producer to describe the kinds of notification that it can produce which can be recognized by subscribers without their needing to have detailed knowledge of the producer.
3. A subject that a notification producer can use as a basis for an access control scheme.

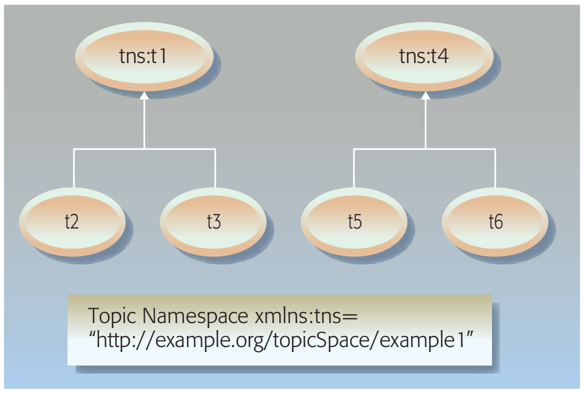
Topic namespaces

Allowing independently developed applications to work together is a WSN design point. These applications might be running in different organizations and might attempt to connect to each other over the Internet. Because all topics in WSN have names, there is the possibility that two such independently developed applications might be using the same topic name—for example, ErrorReport—but describing two different things. In particular the two topics might send radically different kinds of information in their notifications. If one of these applications were to subscribe to the other, then confusion would result. In order to stop this from happening WS-Topics allows every topic to be assigned to an XML namespace.[17] Each XML namespace has a globally unique uniform resource identifier (URI). The combination of the URI of this namespace and the name of the topic is therefore globally unique. You do not need to assign a separate namespace and URI to each topic; in fact, it is often convenient to group together a set of related topic definitions and use the same namespace for all of them.

We refer to the set of topics that share a common namespace as a topic namespace. As well as providing a unique naming scheme for topics, topic namespaces also define meta-data associated with a topic, in particular the type or types of message that a notification producer will send on a given topic. This meta-data in effect augments the notification producer's interface. If a notification consumer recognizes a topic and its meta-data, then it knows what kinds of notification it is going to receive if it subscribes to any notification producer that supports that topic. Although support for the right topics is obviously important, a subscriber might want to use additional criteria when deciding whether to subscribe to a particular notification producer. A subscriber might, for example, be concerned about reliability of delivery, cost, or the quality of the information.

Topic namespaces can be defined by many different people or organizations, for example:

- The designer of a particular notification producer or notification consumer,
- An enterprise application or infrastructure architect, wishing to define a set of standard topics to be used by various applications within the enterprise,
- An enterprise wishing to expose one or more notification producers to suppliers or business partners, or
- A business consortium or standards body defining a specialization of WSN for a particular application domain.


tns:t1
tns:t4
t2
t3
t5
t6
Topic Namespace xmlns:tns=
"http://example.org/topicSpace/example1"

Adapted from *Web Services Topics 1.2, Working Draft 01*. ©OASIS Open (2004). Reprinted with permission.

Figure 5
An example topic namespace

A topic namespace is not tied to a particular notification producer. It contains an abstract set of topic definitions that can be used by many different notification producers. It is also possible for a given notification producer to support topics from several different topic namespaces.

Topics within a topic namespace can be organized into topic trees. As its name might suggest, a topic tree is a hierarchy with a *Root Topic* at the top, and zero or more descendant topics gathered together in a tree-like fashion under this root.

In ***Figure 5*** we illustrate this idea with the simple example topic namespace given in Reference 9. The XML namespace corresponding to this topic namespace is `http://example.org/topicSpace/example1`, and, as with other XML namespaces, it is customary to use a short prefix (in this case `tns:`) to refer to this namespace. The topic namespace has two topic trees, with roots `tns:t1` and `tns:t4`. In this simple example, the trees only contain one level of nesting—Topic `t1` has two child topics, `t2` and `t3`. Topic `t4` has two child topics, `t5` and `t6`.

Each topic in a topic tree has a simple name, and WS-Topics imposes an important restriction on these names. It forbids any topic from having two child topics with the same name—you will see this is obeyed in our example (for instance `t1` has only one child called `t2`). This means that any topic can be uniquely referred to by a combination of its namespace and a simple path expression. For example the path expression for the topic at the bottom right of Figure 5 would be `tns:t4/t6` (in this syntax `tns:` is a prefix identifying the topic namespace, and the forward slash (/) separator character is interpreted as "select a child of the topic identified by the expression preceding the /).

Although our example does not show this, it is possible for two nonroot topics in the same topic space (or even the same topic tree) to have the same name. They are treated as completely separate topics. A company news-feed service might want to define a root topic for each company on which it reports, and then have each topic contain child topics that have the same name, for example: CompanyA/Price, CompanyA/Volume, CompanyB/Price, CompanyB/Volume.

Topic trees are useful for the following reasons:

- They provide a structured naming scheme against which subscribers can issue wild-card subscriptions. A subscriber uses a wild-card subscription to subscribe against multiple topics. It can subscribe against an entire topic tree or a subset of topics in a topic tree in a single subscribe operation.
- They allow related topics to be grouped together for administrative purposes. In particular an administrator might wish to apply a particular security policy to an entire tree or subtree.

WS-Topics defines a way to represent topic namespace meta-data as an XML document. The example shown in ***Figure 6*** is from Reference 9 and matches the example shown in Figure 5.

This document acts as a kind of schema for the topic namespace itself. The `TopicNamespace` element assigns the topic namespace to the `http://example.org/topicSpace/example1` namespace. The topics within the topic namespace are defined by using `Topic` elements, the nesting of these `Topic` elements matching the topic hierarchy.

Each `Topic` element can have a `messageTypes` attribute indicating one or more types of notification message that are associated with the topic. If it supports a given topic from a given topic namespace, a notification producer undertakes to transmit only notifications whose type matches one of the types specified by this attribute. Likewise a notification consumer that is subscribed to a particular topic should be able to process all messages whose types are listed by this attribute.

Topic expressions

Part of the rationale for hierarchical topic namespaces is to allow filter expressions that select multiple topics through the use of wild-card subscription expressions. Wild-card expressions are widely used in publish/subscribe messaging systems, but the characters used and their meaning vary among products. WS-Topics defines a standard set of wild cards based on the search capabilities of the XML Path Language.[18]

Because supporting wild cards is somewhat onerous for a simple notification producer, particularly one that supports only a few root topics, the following levels of topic expression are defined by WS-Topics:

1. *Simple Topic Expression*—An XML QName,[19] consisting of a namespace prefix and a topic name. Simple topic expressions can only be used to refer to root topics (because a forward slash [/] separator is not permitted). A notification producer that only supports root topics might choose only to accept subscriptions that contain simple topic expressions.
2. *Concrete Topic Expression*—A namespace prefix and a sequence of topic names, using a forward slash (/) as the separator character, for example, `tns:t1/t3`. By using a concrete topic expression any topic in a topic namespace can be addressed. A concrete topic expression picks one and only one topic.
3. *Full Topic Expression*—This extends the concrete topic expression to allow it to pick multiple topics. It includes the double forward slash (//), asterisk (*), and period (.) XPath wild-card characters.

WS-Brokered Notification

Figures 2 and 3 in the section "WS-Base Notification" showed examples of direct notification, in which the notification producer also acts as publisher. In fact, the WS-Base Notification specification does not really distinguish between these roles. It merely defines the notification-producer interface and does not concern itself with whether there is a separate publisher entity behind this interface. This means that in addition to providing its normal business functions, detecting situations, and creat-

```
<?xml version="1.0" encoding="UTF-8"?>
<wstop:TopicNamespace name="TopicSpaceExample1"
    targetNamespace="http://example.org/topicSpace/example1"
    xmlns:tns="http://example.org/topicSpace/example1"
    xmlns:xyz="http://example.org/anotherNamespace"
    xmlns:wstop="http://docs.oasis-open.org/wsn/t-1" >
    <wstop:Topic name="t1">
        <wstop:Topic name="t2" messageTypes="xyz:m1 tns:m2"/>
        <wstop:Topic name="t3" messageTypes="xyz:m3"/>
    </wstop:Topic>
    <wstop:Topic name="t4">
        <wstop:Topic name="t5" messageTypes="tns:m3"/>
        <wstop:Topic name="t6"/>
    </wstop:Topic>
</wstop:TopicNamespace>
```

Adapted from *Web Services Topics 1.2, Working Draft 01*. ©OASIS Open (2004). Reprinted with permission.

Figure 6
The topic namespace of Figure 5 as an XML document

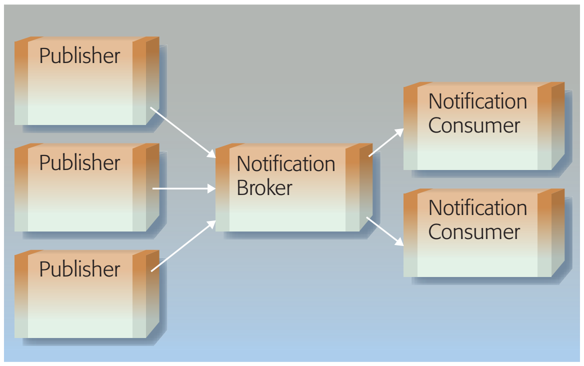
ing notification messages, the notification producer is responsible for matching notifications against the list of subscriptions and for sending notification messages to each appropriately subscribed consumer. This task, however, can be delegated to another entity. WS Brokered Notification, illustrated in ***Figure 7***, defines a new role—the notification broker—that can perform the tasks of a notification producer on behalf of a publisher. Because a notification broker is itself a Web service, it can be hosted remotely from both the publisher and the notification consumers, if required.

The WS-Brokered Notification specification defines the concept of a notification broker as an intermediary Web service that decouples publishers and notification producers. The specification defines the message exchanges that a notification broker is required to support. It also discusses some of the other characteristics that are special to the broker's intermediary role.

The notification-broker role builds on the concepts and message exchanges in WS-Base Notification. In particular a notification broker is itself both a notification producer and a notification consumer. This means that a subscriber creates a subscription with a notification broker by using exactly the same subscribe request as it would when subscribing to a standard notification producer.

The *brokered notification* pattern can provide some or all of the following benefits over the *direct* notification pattern:

1. It relieves a publisher from having to implement message exchanges associated with the notification producer; the notification broker takes on the duties of a subscription manager (managing subscriptions) and notification producer (distributing notifications) on behalf of the publisher.
2. It can reduce the number of interservice connections and references. Consider our printer monitoring scenario and suppose there are two monitor applications and 100 printers. If direct



From *Web Services Brokered Notification 1.3, Public Review Draft 01*. ©OASIS Open (2004-2005). Reprinted with permission.

Figure 7
Brokered notification

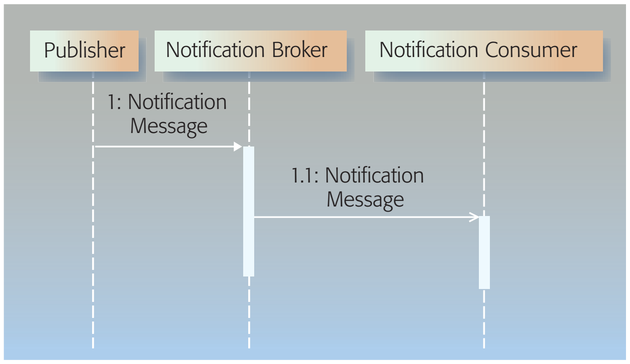
notification is used, the monitor applications both have to issue 100 subscriptions, and each printer has to remember two consumers. Each time a printer changes state, it has to send messages to both consumers. Although this is much more efficient than a polling solution, there are still 200 interservice relationships being maintained. In contrast, if the brokered notification pattern is used, then each monitoring application makes just a single subscription request (to the broker), and the printers simply publish each state change once (again to the broker).

3. A notification broker can act as a kind of *finder service*, putting potential publishers and consumers in touch with each other. Let us return to our printer example and suppose that printer 101 is added. If this printer starts publishing to the broker, then its notifications can be distributed by the broker to the monitor applications without those monitor applications having to issue any new subscription requests.
4. It can provide anonymous notification, so that the publishers and notification consumers need not be aware of each other's identity. This is useful in some, but, of course, not all scenarios. For example, our printer monitoring scenario is one where anonymous notification is not appropriate; a monitor application would usually want to know which printer sent the notification.
5. An implementation of a notification broker may provide additional added-value function, for example, logging notification messages or transforming topics or notification message content. Additional function provided by a notification broker can apply to all publishers that use it.

Publishing to a notification broker

There are several ways in which a publisher can interact with a notification broker. The simplest way, illustrated in *Figure 8*, has the publisher just send one-way notification messages to the broker. By wrapping the notification messages in the WSN notify message, the publisher can associate the message with a topic. The broker then distributes a copy of the notification message to all consumers registered to receive messages on that topic, subject, of course, to any additional filters that the subscriptions may contain.

A broker may choose not to accept just any publisher; instead, it may require that publishers preregister before they can start publishing. In some ways registering as a publisher is similar to registering a subscription. The publisher supplies an endpoint reference for itself as part of the RegisterPublisher request message, and the publication registration can be subject to the same kind of time-based expiry that is used to manage subscriptions. The difference is that instead of undertaking to deliver messages to a consumer, when it accepts a publication registration, the broker undertakes to receive (and pass on) messages from the publisher.



From *Web Services Brokered Notification 1.3, Public Review Draft 01*.
©OASIS Open (2004-2005). Reprinted with permission.

Figure 8
Message flows: simple publisher

There are cases where it is expensive for a publisher to detect a situation or create a notification message instance. A problem with the simple publisher approach is that even when there are no relevant subscriptions, publishers still have to do both of these things. As an optimization, a broker may offer support for *demand-based publishing*, illustrated in *Figure 9*.

A demand-based publisher combines the roles of notification producer and publisher, but only has to manage a single subscription. It implements the subscribe message exchange from the notification producer interface, and the notification broker uses this exchange to establish a subscription with the publisher. The publisher then delivers notifications to the broker by using the standard notify message. If the broker detects that there are no relevant subscriptions, it can pause its subscription with the publisher, resuming it again when it acquires a relevant subscription.

In this way the demand-based publisher does not need to produce messages when there are no consumers, but it can still delegate matching of notification to subscription and other related issues (for example, security) to the broker.

DISCUSSION

In this section we discuss how event-based interactions can be used in SOA alongside interactions based on the request/response pattern. Specifically, we discuss the use of events within the Enterprise Services Bus (ESB) and in the context of Complex Event Processing (CEP).

Notification pattern and SOA

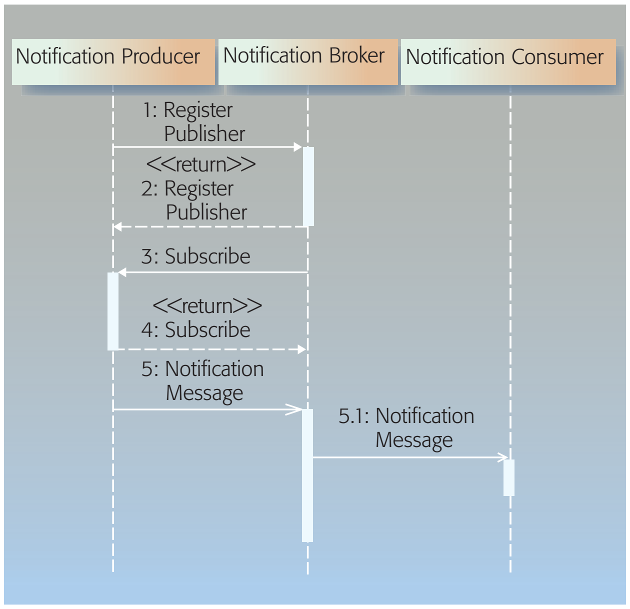
Design patterns are frequently used in software engineering as a way of capturing knowledge about software design and reusing it and also as a mechanism for communicating elements of a design to others. The use of design patterns was pioneered by Gamma, Helm, Johnson, and Vlissides[20] (often referred to as the Gang of Four), primarily in the context of object-oriented software. Their book includes an event-based pattern called the *observer* pattern, inspired by the Smalltalk MVC model.[1] The idea of patterns was applied to message-oriented middleware and enterprise integration by Hohpe and Woolf in Reference 21, and this book also includes a number of patterns for event-based programming.

The WS-Base Notification specification can be viewed as a rendering of the notification pattern, which itself can be viewed as an instantiation of the observer pattern for the SOA environment (and for this reason we could also refer to it as the *SOA notification pattern*). This pattern unifies the principles and concepts of SOA with those of event-based programming.

In the SOA notification pattern, a service distributes information to a set of other services without necessarily having prior knowledge of those other services.

This pattern has the following key characteristics:

1. The services that consume information (notification consumers) are registered (either by themselves or by a third party) with the service that is capable of distributing that information (the notification producer). As part of this registration process, they may provide *filter* expressions that indicate the sort of information that they wish to receive.
2. The notification-producer service disseminates information by sending one-way messages to the relevant notification-consumer services. It is possible that more than one notification-consumer service is registered to consume the same information. In such cases, each notification-consumer service that is registered receives a separate copy of the information.
3. The notification-producer service may send any number of messages to each registered notification-consumer service; it is not limited to sending just a single message.



From *Web Services Brokered Notification 1.3, Public Review Draft 01*.
©OASIS Open (2004-2005). Reprinted with permission.

Figure 9
Message flows: demand-based publisher

The first of these three characteristics lies at the heart of the pattern. The notification producer does not have prior knowledge of the notification consumers, nor does it look them up in a service registry. Instead it has to wait for these notification-consumer services to register with it before it can start distributing its information to them.

The second characteristic states that a notification producer must be able to support configurations where more than one notification consumer is registered to receive its information. An example is the stock-trading scenario. In this scenario there might typically be just one notification producer service that provides the price feed, but at any given time, there may be many stock-trading services registered with it.

In the stock-trading scenario there are typically more notification consumers than producers; in contrast in a system-monitoring application (such as the printer management example) things are usually the other way round. In a typical monitoring scenario, there are many notification producers (representing pieces of computing hardware or software, or even physical devices such as temperature sensors or motion detectors), and relatively few consumers (monitoring applications).

The third characteristic says that, once registered, a notification consumer must expect that it could receive a sequence of successive notification messages. These notification messages are related to the original registration request only in as much as they relate to the interests expressed by that original request.

Readers familiar with the object-oriented software design patterns described in Reference 20 will notice a similarity with the Observer pattern—the notification producer is similar to the Observer pattern's Subject, and the consumer is like the Observer pattern's Observer. The SOA notification pattern differs in a few ways from the Observer pattern, as it relates to services rather than to object-oriented programming:

1. The Observer pattern is primarily concerned with exchanging the state of an object with other objects. In the Observer pattern, an Observer object registers to be notified about changes in the state of another object (the Subject). Because services have a much more loosely defined concept of state than objects, the notification pattern is more general. It allows a notification producer to distribute any kind of information. This may be information about changes in state, but it does not have to be.
2. In the Observer pattern, a consumer deregisters with the Subject when it wishes to stop receiving notifications. As we will see later, the notification pattern allows for time-based expiry of subscriptions. This is because in a loosely coupled environment we cannot always rely on subscribers being able to cancel their subscriptions.
3. The classical version of the Observer pattern does not provide a way for Observers to indicate what kind of state changes they wish to observe, and it does not allow for state-change information to be passed to the Observer in the Update() message. All that the Observer is told is that the Subject's state has changed in some way, and it is then up to the Observer to request the new state from the Subject. Since message exchanges between loosely coupled services can be comparatively costly, the notification pattern allows the consumer to specify that it is only interested in being notified when certain conditions apply, and it allows the notification producer to pass information when it notifies the consumer.

In the past year, various analysts and software vendors have started to use the term event-driven architecture (EDA) to describe software architectures that utilize event-based programming. This has raised the question of whether EDAs can be classified as SOAs. The answer to this question depends upon your definition of a service.

One definition, used by the Gartner group in Reference 22, asserts that a key feature of a service is that it always executes functionality on behalf of a particular requestor. Under this definition, an entity that is driven by an event, rather than by being bound to a requestor, does not qualify as a service. In this paper we follow the definition of service given in Reference 14, "A service has a well-defined interface with a set of messages that the service receives and sends, and a set of named operations or verbs; an implementation of the interface; and, if deployed, a binding to a documented network address." With this broader definition there is no problem in using the term *service* to describe an entity that reacts to an event. We therefore view EDA as being part of the wider SOA concept.

This view is reinforced by observing that the difference between request/response implementations and event-based programming is not as great as it may seem. In particular, there are cases where the messages exchanged are visible in request/response implementations. We have already noted that some SOAs allow request/response services to be invoked in an asynchronous manner, and this requires the sending of the request to be separated from the processing of the reply. In addition, recent years have seen a growth in the use of document-centric request/response services. In a document-centric service, the interface to the service, and often the programming model, is expressed in terms of the request messages and response messages themselves, rather than in terms of a function or procedure call.

The debate about whether EDA is a branch of SOA or whether it is something different is academic. From a practical viewpoint it is important to note the following:

- Many applications need a mixture of request/response and event-based programming.
- Both approaches decouple the services involved, leading to more flexible, loosely coupled systems.
- Both approaches bring with them infrastructural requirements, for example, application registries, service management, monitoring, security, and reliability. Enterprises are better served by having a consistent approach to service orientation that accommodates both patterns, and having a single infrastructure to manage—rather than having separate infrastructures for the two patterns.

A particular example of an application domain that can take advantage of both request/response and event-based programming is system management (for example, see Web Services Distributed Management[23]). The request/response pattern is used for sending commands from a manager to a resource or by a resource to request assistance from a manager; however, a manager can use events to monitor the behavior of resources.

In many monitoring situations the event-based approach results in less network traffic and better responsiveness than the alternative request/response pattern approach. In the request/response approach the manager has to poll each device periodically to see if its status has changed. The monitoring program has to choose the time interval between polls. If a long interval is chosen (say one poll per hour) this increases the average time taken for the monitor to notice changes in the resources that it is managing; alternatively, choosing a short interval increases the amount of network traffic, and there is an increased likelihood that the resources, when polled, will have no new status to report. In contrast, in the event-based approach there is no need to choose a polling interval; resources can send messages when they need to; and resources need only send messages if they have new information to report. For an example of how the event-driven approach has been used to simplify programming and reduce processor utilization, see Reference 24.

The Enterprise Service Bus and Complex Event Processing

An Enterprise Service Bus (ESB), outlined in Reference 11, is a service-oriented infrastructure that mediates requests between participating services. This includes both traditional request/response style message exchanges, as well as the event-based message exchanges. In this final section, we look at how the ESB embodies the notification pattern, and how the ESB can add value by augmenting the pattern with further processing.

The aspects of an ESB that concern us here are the following:

- The ESB provides an infrastructure that virtualizes the services which are made available on it. Services that connect to the bus need have no awareness of the physical realization of the applications with which they communicate. These partner services may be conventional service request-ors and providers, or they may be event-oriented notification producers or consumers. Services that connect to the bus need not care about implementation details (the programming language, runtime environment, hardware platform, network address, etc.) or current availability of their partners.
- This infrastructure is itself virtualized and is capable of spanning wide area networks and involving multiple infrastructure servers, if required.
- The ESB provides a mediation capability that can transform or route messages flowing through the ESB or perform other CEP.

An ESB provides support for the following kinds of event-oriented application:

1. Traditional message-oriented middleware applications coded to use publish/subscribe, for example, C applications written to use IBM WebSphere* MQ, IBM WebSphere Business Integration Message Broker, or Java** applications that use the Java Message Service (JMS) publish/subscribe interfaces.[25]

2. Notification producers or notification consumers as defined by WSN, connecting across a variety of bindings.
3. Applications coded to emit events by using the IBM Common Event Infrastructure interfaces.

An ESB allows these different kinds of application to connect together; for example, a notification message might be published by a JMS application and received by a notification consumer as defined by WSN. In addition, the ESB can mediate between request/response services and event-oriented services. The ESB provides this support by managing collections of topics in its registry and by providing an implementation of one or more distributed notification brokers as defined by WSN.

Traditional message-oriented middleware applications are unaware of the WSN aspect of these brokers—they simply publish or subscribe to topics on the broker of their choice (in JMS, applications are shielded from these details by topic and connection factory administered objects). The ESB administrator can configure one or more WSN-style notification brokers to be used by WSN publishers or subscribers, and the ESB administrator can arrange that these be federated with each other, with traditional publish/subscribe applications, or with both.

To give a concrete example, an administrator could create a topic namespace containing stock price symbols as topics, then create a notification broker instance, and then publish the port for this broker instance for use by subscribers or notification consumers that connect over SOAP/HTTP.

The administrator could then take an existing JMS application that provides the price feed and attach it to the bus as a publisher. In order to do this, the administrator configures a mapping from one or more topics used by the JMS application into topics from the topic namespace. The administrator might also need to configure a mediation to convert the payload of the notification messages into a format that matches the type defined by the topic namespace.

In another example, the administrator might want to configure two notification brokers in two different physical locations and have them interconnected via the ESB messaging infrastructure. Publishers or notification consumers in each location might connect to their local broker via SOAP/HTTP on their internal network. The ESB infrastructure can then optimize the number of notification messages that have to flow between the two sites. For example, if there are 20 consumers at one site, the infrastructure need only flow a single copy of each message, not 20 copies.

The ESB supports both dynamic subscription, where a notification consumer registers itself (or is registered by a third-party subscriber) at runtime, and a more static administrator-defined subscription mechanism where consumers are automatically registered when they start up. The stock price example we used earlier is likely to use dynamic subscription because individual traders can come and go during the day. Alternatively, our other example (printer management) might be more suited to a static configuration because a monitoring application is likely to be long-running and predefined in the ESB registry.

The mediation capability of the ESB can be used to provide additional CEP. Examples of this range from basic logging of notification messages for subsequent retrieval and analysis to more advanced scenarios involving aggregation or correlation to produce additional derived events.

CEP mediations frequently do not modify the notification messages themselves. The ESB delivers notifications from producers and consumers in the normal fashion, but the mediations examine the messages that flow through, logging them in storage, which could be a simple in-memory buffer or a relational database. Each time a new notification message passes through the mediations, they analyze it and compare it with information recorded in storage. The mediations may then detect a new situation and publish a new notification message in addition to the message that originally triggered this behavior. We call this a derived notification, and the content of its message may be totally different from that of the original notification. Derived notifications conform to the notification pattern; notification consumers can subscribe to the ESB to receive them in exactly the same way that they subscribe to receive any other kind of notification.

As an example, let us return for the last time to the printer monitoring scenario. CEP mediations could be added to the ESB to allow the printer monitoring application to subscribe to the following derived events:

- *Floor problem*—All the printers on the same floor are offline at the same time. The mediation uses information in the printer-offline notification to identify the location of the printer (if the floor information is not carried in the message, the mediation can extract a printer ID from the message and discover the floor by looking in a database). It then maintains a count of the number of printers available on each floor and generates a new derived event if this count drops to zero.
- *Frequent jams*—A printer has more than three paper jams within a single working day. This mediation simply records the printer jam messages that it receives from each printer, and each time it receives a new one, it generates a derived event if the threshold for the day (in this case three jams) has been exceeded.
- "*Mean time to repair*" *problem (the mean time that a printer stays offline within a week is more than 45 minutes)*—This is a more complex time-based mediation. The mediation logs error-related printer online and offline messages and computes a weekly rolling average for the repair time, generating a derived event if this exceeds 45 minutes.

For more information on CEP and some discussion of its uses and implementation, see References 6 and 26.

SUMMARY

We have reviewed the OASIS WSN specification as a way of illustrating how event-based programming can be introduced in SOA in a standardized way and have observed how the pattern embodied in these specifications, the SOA notification pattern, can be viewed as an instantiation of the well-known observer design pattern for the SOA environment.

We concluded with a brief discussion of CEP and an examination of the role of the SOA notification pattern within the ESB and saw how the ESB combines request/response and event-oriented SOA into a single infrastructure. This combination brings many advantages. There is a single infrastructure to support, manage, and secure, and there is uniformity in the transport bindings that are available to both styles of application. Quality-of-service options such as reliable delivery, message logging, or store and forward can be applied to either pattern, and it does not make sense to deploy two infrastructures to provide similar functionality for the two patterns. In addition, because it is quite common for a single service to combine both request/response and event-oriented message exchanges, for this sort of application a single infrastructure is a necessity.

One further advantage of combining both patterns in the same infrastructure is that opportunities for *cross-over* emerge. For example, an ESB administrator can insert a mediation into a request/response route through the bus. This mediation does not affect the contents of the request or response messages, but it can analyze the messages and publish a notification if and when it detects an out-of-line situation. Adding and removing such mediations is made more straightforward if the mediation framework, the messaging, and the event-handling fabrics are all a single integrated whole.

*Trademark, service mark, or registered trademark of International Business Machines Corporation.

**Trademark, service mark, or registered trademark of Object Management Group, Inc. and Sun Microsystems, Inc.

CITED REFERENCES

1. G. Krasner and S. Pope, "A Cookbook for Using the Model-View Controller User Interface Paradigm in Smalltalk-80," *Journal of Object Oriented Programming* **1**, Issue 3, 26–49 (August/September 1988).
2. R. Scheifler, J. Gettys, A. Mento, and D. Converse, *X Window System: Core and Extension Protocols 4th Edition*, Butterworth-Heinemann (1997).
3. M. Perry, C. Delporte, F. Demi, A. Ghosh, and M. Luong, *MQSeries Publish/Subscribe Applications*, IBM Redbook (2001).
4. *Event Service Specification*, Object Management Group (2004), http://www.omg.org/technology/documents/formal/event_service.htm.
5. *Notification Service Specification*, Object Management Group (2004), http://www.omg.org/technology/documents/formal/notification_service.htm.
6. D. Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*, Addison Wesley, Reading, MA (2002).
7. *OASIS Web Services Notification Technical Committee*, OASIS, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsn.
8. *Web Services Base Notification 1.3*, OASIS (2005), http://www.oasis-open.org/committees/download.php/13488/wsn-ws-base_notification-1.3-spec-pr-01.pdf.
9. *Web Services Topics 1.2*, OASIS (2004), http://docs.oasis-open.org/wsn/2004/06/wsn-WS-Topics-1.2-draft-01.pdf.
10. *Web Services Brokered Notification 1.3*, OASIS (2005), http://www.oasis-open.org/committees/download.php/13485/wsn-ws-brokered_notification-1.3-spec-pr-01.pdf.

11. M.-T. Schmidt, B. Hutchison, P. Lambros, and R. Phippen, "The Enterprise Service Bus—Making Service-Oriented Architecture Real," *IBM Systems Journal* **44**, No. 4, 781–798 (2005, this issue).

12. *XML Schema Part 1: Structures*, World Wide Web Consortium (2004), http://www.w3.org/TR/xmlschema-1/.

13. *Web Services Addressing 1.0—Core*, World Wide Web Consortium (2005), http://www.w3.org/TR/ws-addr-core/.

14. D. F. Ferguson and M. L. Stockton, "Service Oriented Architecture: Programming Model and Product Architecture," *IBM Systems Journal* **44**, No. 4, 753–780 (2005, this issue).

15. *Web Services Resource Properties*, OASIS (2004), http://docs.oasis-open.org/wsrf/2004/06/wsrf-WS-ResourceProperties-1.2-draft-04.pdf.

16. *Canonical Situation Data Format: The Common Base Event V1.0.1*, The Eclipse consortium (2003), http://dev.eclipse.org/viewcvs/indextools.cgi/~checkout~/hyades-home/docs/components/common_base_event/cbe101spec/CommonBaseEvent_SituationData_V1.0.1.pdf.

17. *Namespaces in XML*, World Wide Web Consortium (1999), http://www.w3.org/TR/1999/REC-xml-names-19990114/.

18. *XML Path Language*, World Wide Web Consortium (1999), http://www.w3.org/TR/xpath.

19. *Extensible Markup Language (XML) 1.0*, World Wide Web Consortium (2004), http://www.w3.org/TR/REC-xml.

20. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*, Addison Wesley, Reading, MA (1995).

21. G. Hohpe and B. Woolf, *Enterprise Integration Patterns*, Addison Wesley, Reading, MA (2003).

22. R. Schulte and Y. Natis, "Event-Driven Architecture Complements SOA" *Gartner Research Note* DF-20–1154, Gartner, Inc. (2003).

23. *Web Services Distributed Management Technical Committee*, OASIS, http://oasis-open.org/committees/tc_home.php?wg_abbrev=wsdm.

24. H. Muller and C. Randell, "An Event-Driven Sensor Architecture for Low Power Wearables," *Proceedings of the International Conference on Software Engineering ICSE2000, Workshop on Software Engineering for Wearable and Pervasive Computing*, ACM, New York (2000), pp. 39–41, http://www.cs.washington.edu/sewpc/papers/muller.pdf.

25. Java Message Service 1.1 Documentation, Sun Microsystems, Inc., http://java.sun.com/products/jms/docs.html.

26. A. Adi and O. Etzion, "Amit—the Situation Manager," *VLDB Journal* **13**, No. 2, 173–203 (May 2004).

Accepted for publication July 12, 2005.
Published online October 26, 2005.

Peter Niblett
IBM United Kingdom Limited, Hursley Park, Winchester, Hampshire SO21 2JN, United Kingdom (peter_niblett@uk.ibm.com). Mr. Niblett, a Senior Technical Staff Member, is lead architect for WebSphere Messaging software. He was the IBM lead on the working group that wrote the Java Message Service specification and is now co-chairing the OASIS Web Service Notification Technical Committee.

Steve Graham
IBM Software Group, 4400 Silicon Dr, Durham NC 27713 (sggraham@us.ibm.com). Steve Graham is a Senior Technical Staff Member in IBM's Software Group and a member of the IBM Academy of Technology. He is a Web services architect involved in standards-related activities within the IBM on demand initiative. He has spent the last five years working on service-oriented architectures. He is the lead author of *Building Web Services with Java: Making Sense of XML, SOAP, WSDL and UDDI (Second Edition)*, published by Sams Publishing. ■