Publish/Subscribe

Retired Content

This content is outdated and is no longer being maintained. It is provided as a courtesy for individuals who are still using these technologies. This page may contain URLs that were valid when originally published, but now link to sites or pages that no longer exist.

Integration Patterns

Contents

Aliases

Pub/Sub

Context

You have an integration architecture consisting of several applications. A communication infrastructure connects these applications in a bus, broker, or point-to-point topology. Some applications send multiple message types. Other applications are interested in different combinations of these types.

For example, consider a financial system where several applications maintain customer information. A Customer Relationship Management (CRM) application holds the master customer information. However, a typical situation for integration scenarios exists—customer information also resides in other systems that perform their own customer information management functions. A customer-facing application generates update messages for changes to customer information, such as changes to customer addresses. The messages must reach the CRM application as well as the other applications that manage customer information. However, this message type is meaningless to the integrated applications that do not manage customer information.

Problem

How can an application in an integration architecture only send messages to the applications that are interested in receiving the messages without knowing the identities of the receivers?

Forces

Integrating applications so that they receive only the messages they are interested in involves balancing the following forces:

- The applications in an integration architecture consume different message types. For example, applications that manage customer information are interested in customer information updates. Trading applications are interested in buy and sell transactions. Applications that participate in two-phase commit transactions are interested in commit messages.
- An application in an integration architecture may send several message types. For example, the application may send customer information messages and operational messages about its status. (Status is also referred to as *health* in this context). Likewise, an application in an integration architecture is usually interested only in a subset of the messages that are sent by the other applications. For example, a portfolio manager is interested only in the financial transactions that affect the stocks that it manages.
- The extent to which applications let you add information to their messages varies widely. Fixed binary messages usually provide no flexibility or limited flexibility in this area. In contrast, it is usually easy to extend SOAP messages through envelope elements.
- Most integration architectures integrate proprietary applications. These applications often make strong assumptions about the messages that they use to communicate with other applications in the environment. Even with a flexible message format, it may be difficult to insert or to process message elements that the application does not know about.

Solution

Extend the communication infrastructure by creating topics or by dynamically inspecting message content. Enable listening applications to subscribe to specific messages. Create a mechanism that sends messages to all interested subscribers. The three variations of the *Publish/Subscribe* pattern you can use to create a mechanism that sends messages to all interested subscribers are *List-Based Publish/Subscribe*, *Broadcast-Based Publish/Subscribe*, and *Content-Based Publish/Subscribe*.

List-Based Publish/Subscribe

A *List-Based Publish/Subscribe* pattern advises you to identify a subject and to maintain a list of subscribers for that subject. When events occur, you have the subject notify each subscriber on the subscription list. A classic way to implement this design is described in the Observer [Gamma95] pattern. When you use this pattern, you identify two classes: subjects and observers. Assuming you use a push model update, you add three methods to the subject: **Attach()**,**Detach()**, and **Notify()**.You add one method to the observer—**Update()**.

To use an observer, all interested observers register with the subject by using the **Attach()**method. As changes occur to the subject, the subject then calls each registered observer by using the **Notify()**method. For a detailed explanation of the *Observer* pattern, see *Design Patterns: Elements of Reusable Object-Oriented Software* [Gamma95].

An observer works fine if you have created instances of objects that reify all your observers and subjects. An observer is especially well suited to situations where you have one-to-many relationships between your subjects and your observers. However, in the context of integration, you often have many observers that are linked to many subjects, which complicates the basic *Observer* pattern. One way to implement this many-to-many relationship is to create many subjects and to have each subject contain a list of observers.

If you use this object structure to implement *Publish/Subscribe*, you must write these relationships to persistent storage between process executions. To do so within a relational database, you must add an associative table to resolve the many-to-many dependencies between subject and observer. After you write this information to persistent storage in a set of tables, you can directly query the database for the list of subscribers for a topic.

Maintaining lists of published topics (subjects) and subscribers (observers) and then notifying each one individually as events occur is the essence of *List-Based Publish/Subscribe* implementations. A very different means of achieving the same result is a *Broadcast-Based Publish/Subscribe* implementation.

Broadcast-Based Publish/Subscribe

When you use a *Broadcast-Based Publish/Subscribe* approach [Tannebaum01, Oki93], an event publisher creates a message and broadcasts it to the local area network (LAN). A service on each listening node inspects the subject line. If the listening node matches the subject line to a subject that it subscribes to, the listening node processes the message. If the subject does not match, the listening node ignores the message.

Subject lines are hierarchical and may contain multiple fields that are separated by periods. Listening nodes that are interested in a particular subject can subscribe to these fields by using wildcards, if required.

Although this *Broadcast-Based Publish/Subscribe* implementation is an effective method for decoupling producers from consumers, it is sometimes useful to identify particular topic subscribers. To identify topic subscribers, a coordinating process sends a message that asks listening nodes to reply if they subscribe to a particular topic. Responses are then returned by each listening node to the provider to identify the subscribers.

Because all messages are sent to all listening nodes, and because each node is responsible for filtering unwanted messages, some authors refer to this as a publish/subscribe channel with reactive filtering [Hohpe04].

Content-Based Publish/Subscribe

Both *Broadcast-Based Publish/Subscribe* implementations and *List-Based Publish/Subscribe* implementations can be broadly categorized as topic-based because they both use predefined subjects as many-to-many channels. *Publish/Subscribe* implementations have recently evolved to include a new form—*Content-Based Publish/Subscribe*. The difference between topic-based and content-based approaches is as follows:

> In a topic-based system, processes exchange information through a set of predefined subjects (topics) which represent many-to-many distinct (and fixed) logical channels. Content-based systems are more flexible as subscriptions are related to specific information content and, therefore, each combination of information items can actually be seen as a single dynamic logical channel. This exponential enlargement of potential logical channels has changed the way to implement a pub/sub system. [Baldoni03]

The practical implication of this approach is that messages are intelligently routed to their final destination based on the content of the message. This approach overcomes the limitation of a broadcast-based system, where distribution is coupled to a multicast tree that is based on Transmission Control Protocol (TCP). It also gives the integration architect a great deal of flexibility when deciding on content-based routing logic.

Applying Publish/Subscribe

Figure 1 shows an integration solution that consists of four applications. The sender (also called a *publisher*) uses a topic-based approach to publish messages to topic A and to topic B. Three receivers (also called *subscribers*) subscribe to these topics; one receiver subscribes to topic A, one receiver subscribes to topic B, and one receiver subscribes to both topic A and to topic B. The arrows show messages flowing from the publisher to each subscriber according to these subscriptions.

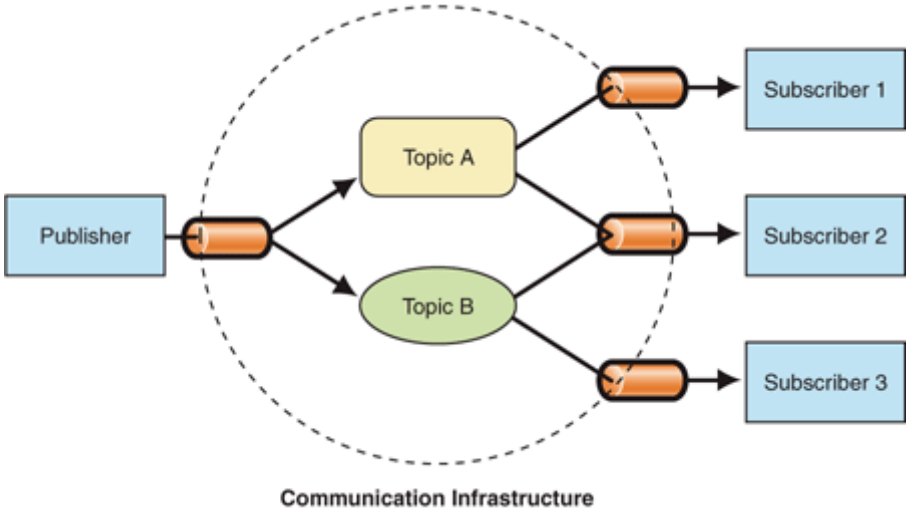

Figure 1. Subscription to topics controls the message types that reach each subscriber

Implementing *Publish/Subscribe* usually affects the messages, the integrated applications, and the communication infrastructure.

First, you must identify the topics or the content of interest to the receiving applications. This translates into partitioning the set of message types into different subsets. For example, consider the types of messages that are sent by a trading system. Some trading applications track buy transactions, some track sell transactions, and other applications track both types of transaction. Separating the message by creating a buy topic and a sell topic partitions the trading system messages into subsets aimed at these applications.

Next, you must add information to the messages that indicates the topic or that identifies specific content information. Sometimes you can store the topic-related information in an unused message field. Alternatively, you may be able to add a new field for the topic. For example, you may be able to insert a new element in a SOAP header. If you can neither use an existing field nor add a new one, you must find other ways to encode the topic into the message, or you must use a content-based approach instead.

You must then extend the communication infrastructure so that it delivers messages according to each subscriber's subscription. The approach that you use depends on the topology of the integration solution. For example, consider the three common topologies. For bus integration, you can implement the subscription mechanism in the bus interface. For broker integration, you can implement the mechanism through subscription lists to the broker. For point-to-point integration, you can implement the mechanism through subscription lists in the publisher.

Finally, you must modify the integrated applications. The publisher must add the topic-related information to each message that it publishes. For example, if the topic is encoded as a header element, the publisher must insert the topic-related information into the appropriate element. Likewise, the subscriber must specify the topics of interest.

Subscriptions can be either fixed or dynamic. With fixed subscriptions, the integration architect sets the topics that an application subscribes to. Applications have no control over their subscriptions. Usually, the subscriptions are specified when each application is added to the integration solution. Figure 2 shows a fixed subscription to Topic A.

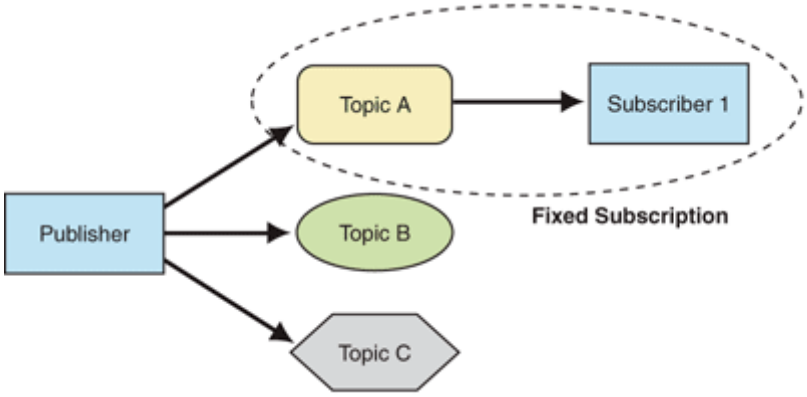

Figure 2. Publish/Subscribe with fixed subscription

In contrast, dynamic subscriptions enable applications to control their own subscriptions through a set of control messages. Applications can remove existing subscriptions by sending messages to the communication infrastructure that remove the application from the subscription list. Applications can add new subscriptions by sending messages to the communication infrastructure that add the application to a subscription list. Most communication infrastructures that have *Publish/Subscribe* capabilities provide this feature. However, supporting dynamic subscriptions is not a requirement.

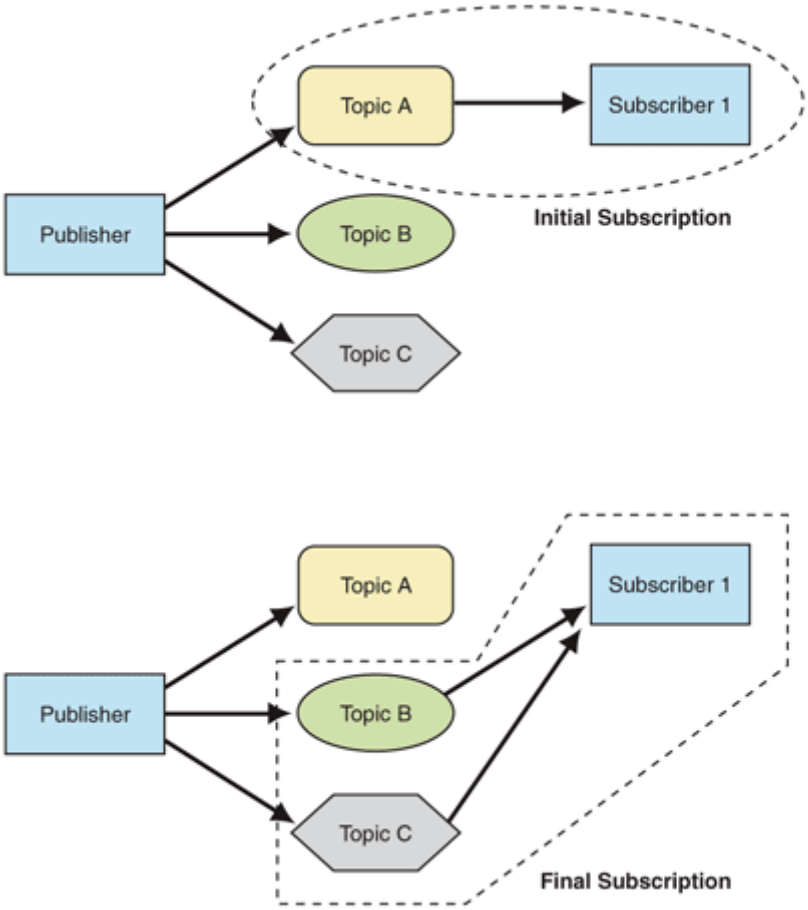

Figure 3. Publish/Subscribe with dynamic subscriptions

Figure 3 shows how dynamic subscriptions function. The top part of Figure 3 shows the initial subscription to topic A. The application then sends a message that removes it from the subscription list for topic A. The application then sends two messages that subscribe the application to topic B and topic C. The bottom part of Figure 3 shows the final subscription after these control messages are sent.

Related Decisions

After you decide to use *Publish/Subscribe*, you must make the following decisions:

- **Initial subscription**. You must decide how subscribers communicate their subscriptions to the communication infrastructure when they are first added to the solution.
- **Wildcard subscriptions**. You must decide if your publish/subscribe mechanism needs to support wildcard subscriptions. Wildcard subscriptions enable subscribers to subscribe to multiple topics through one subscription.
- **Static or dynamic subscriptions**. You must decide if the applications in your integration solution need to change their subscriptions dynamically.
- **Topic discovery**. You must decide how subscribers discover the available topics if the solution supports dynamic subscriptions.

Responsibilities and Collaborations

Table 1 summarizes the responsibilities and collaborations of the parties involved in *Publish/Subscribe*.

Table 1: Responsibilities and Collaborations Among Publish/Subscribe Components

Components	Responsibilities	Collaborations
Communication infrastructure	Maintains the subscribers' subscriptions. Inspects the topic-related information or the content information that is included in each published message. Transports the message to the subscribed applications.	The publisher publishes messages. The subscriber subscribes to topics and receives messages.
Publisher	Inserts topic-related information or content information in each message. Publishes the message to the communication infrastructure.	The communication infrastructure transports messages to subscribers.
Subscriber	Subscribes to one or more topics or message content types. Consumes messages published to the subscribed topics.	The communication infrastructure transports published messages from the publisher.

Example

Microsoft BizTalk Server 2004 uses the *Publish/Subscribe* pattern internally to receive, to route, and to send messages. BizTalk Server receives messages through input ports and stores them in the MessageBox database. Orchestration ports and send ports consume messages from this database based on their subscriptions. Figure 4 illustrates this arrangement.

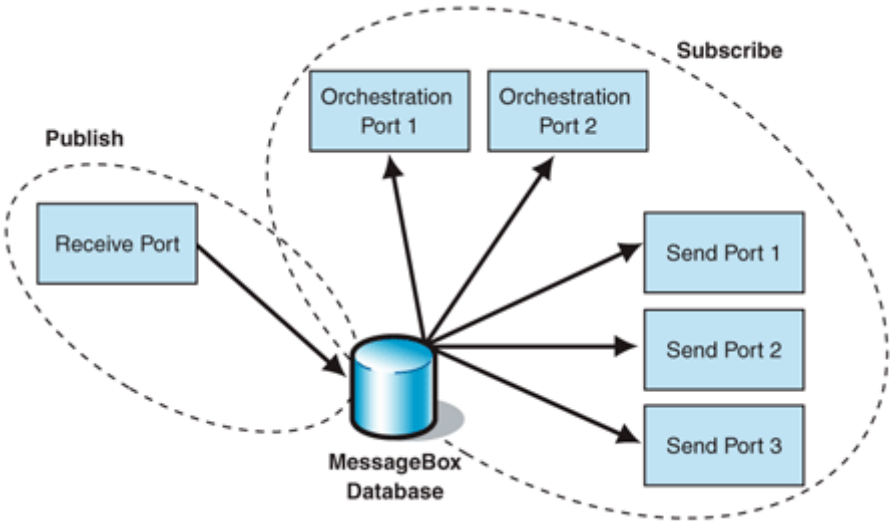

Figure 4. Publish/Subscribe in BizTalk Server 2004

Resulting Context

Using *Publish/Subscribe* has the following benefits and liabilities. Evaluate this information to help you decide whether you should implement *Publish/Subscribe*:

Benefits

- Lowered coupling. The publisher is not aware of the number of subscribers, of the identities of the subscribers, or of the message types that the subscribers are subscribed to.
- Improved security. The communication infrastructure transports the published messages only to the applications that are subscribed to the corresponding topic. Specific applications can exchange messages directly, excluding other applications from the message exchange.
- **Improved testability**. Topics usually reduce the number of messages that are required for testing.

Liabilities

- Increased complexity. *Publish/Subscribe* requires you to address the following:
  - You have to design a message classification scheme for topic implementation.
  - You have to implement the subscription mechanism.
  - You have to modify the publisher and the subscribers.
- Increased maintenance effort. Managing topics requires maintenance work. Organizations that maintain many topics usually have formal procedures for their use.
- Decreased performance. Subscription management adds overhead. This overhead increases the latency of message exchange, and this latency decreases performance.

Testing Considerations

The topics of a *Publish/Subscribe* implementation facilitate the testing of an integration solution. Subscriptions provide isolation by segmenting the message space. By subscribing only to the topics or to the content of interest, testers and testing tools have fewer messages to sift through. Likewise, by subscribing to other topics or content, testers can catch messages that are published to the wrong topic.

Security Considerations

An integration solution that uses *Publish/Subscribe* can restrict the participants of a message exchange, thus enabling applications to have private message exchanges. Depending on the topology, the messages may still be

physically transported to all the applications in the integration architecture. For example, all the messages are transported to all the applications if your integration solution uses the *Message Bus using Broadcast-Based Publish/Subscribe* pattern. However, the interface between the communication infrastructure and the application enforces filtering according to each application's subscriptions.

Operational Considerations

Many integration solutions that use *Publish/Subscribe* have topics or content that is dedicated to messages about the applications' health. This separation facilitates your ability to monitor various operational parameters and to control the applications in the integration solution.

Related Patterns

For more information about *Publish/Subscribe*, see other similar patterns:

- Message Bus and Message Broker describe two common integration topologies.
- Observer [Gamma95] provides a mechanism for decoupling dependencies between applications.
- *Publisher-Subscriber* [Buschmann96] facilitates state synchronization between cooperating components.
- *Publish-Subscribe Channel* [Hohpe04] provides a way to broadcast events to all the receivers (subscribers) that subscribe to a specific topic.

Acknowledgments

[Baldoni03] Baldoni, R.; M. Contenti, and A. Virgillito. "*The Evolution of Publish/Subscribe Communication Systems.*" *Future Directions of Distributed Computing.* Springer Verlag LNCS Vol. 2584, 2003.

[Buschmann96] Buschmann, Frank; Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns.* John Wiley & Sons Ltd, 1996.

[Gamma95] Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Hohpe04] Hohpe, Gregor, and Bobby Woolf. *Enterprise Integration Patterns: Designing, Building and Deploying Messaging Solutions*. Addison-Wesley, 2004.

[Oki93] Oki, B.; M. Pfluegel, A. Siegel, and D. Skeen. "The Information Bus - An Architecture for Extensive Distributed Systems." *Proceedings of the 1993 ACM Symposium on Operating Systems Principles*, December 1993.

[Tannebaum01] Tannebaum, Andrew. *Modern Operating Systems.* 2nd ed. Prentice-Hall, 2001.

Start | **Previous** | **Next**