Two patterns for distributed systems: Enterprise Service Bus (ESB) and Distributed Publish/Subscribe

Eduardo B. Fernandez[1], Nobukazu Yoshioka[2], and Hironori Washizaki[3]

1 Dept. of Comp. Science and Eng., Florida Atlantic University, Boca Raton, FL, USA, ed@cse.fau.edu
2 GRACE Center, National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo, Japan, nobukazu@nii.ac.jp
3 Waseda University / GRACE Center, National Institute of Informatics, 3-4-1, Okubo, Shinjuku-ku, Tokyo, Japan, washizaki@waseda.jp

1. Introduction.

Most current applications are distributed. An important subset of them use web services and are designed using Service-Oriented Architecture (SOA) principles. We define SOA as an architectural style in which a system is composed from a set of loosely coupled services that interact with each other by sending messages (packets of data). In order to interoperate, each service publishes its description, which defines its interface and expresses constraints and policies that must be respected in order to interact with it. In this architectural style, applications are built by coordinating and assembling services in the form of a workflow that invokes services as needed, as well as standard software components [Zim05]. A service is a logical representation of a business activity that has a specified outcome. A key principle about services is that they should be easily reusable and discoverable, even in an inter-organizational context. Furthermore, the channels of communication between the participating entities in a service-oriented application are much more vulnerable than in operating systems or within the boundaries of an organization's intranet, since they are established on public networks. The complexity of the software used to handle web services adds to the total complexity and can be a source of attacks, which makes security an important concern [Pap07].

Papazoglou, and van den Heuvel [Pap07] give a review of technologies and approaches that unify the principles and concepts of SOA with those of event-based programming. It focuses on the Enterprise Service Bus and describes a range of functions that are designed to offer a manageable, standards based SOA backbone that extends middleware functionality throughout by connecting heterogeneous components and systems and offers integration services. They propose an approach to extend the conventional SOA to cater for essential Enterprise Service Bus (ESB) requirements that include capabilities such as service orchestration, "intelligent" routing, provisioning, integrity and security of messages as well as service management.

The Organization for the Advancement of Structured Information Standards (OASIS) produced a SOA Reference Model (SOA RM) [OAS]. This model is an abstract framework for understanding significant entities and relationships between them, and for

the development of consistent standards or specifications supporting that environment. It is based on unifying concepts of SOA and may be used by architects developing specific service oriented architectures or in training and explaining SOA. It uses the ESB as an important concept.

Because the ESB is a fundamental unit in SOA, it has been described as a pattern, e.g. in [Erl09, Zdu06]. However, these patterns present mostly the idea of the ESB but not enough detail to help a designer use an ESB effectively in her designs. The ESB has a value beyond SOA and can be used for all types of distributed systems. We present here a pattern for ESBs that provides more detail and incorporates explicitly nonfunctional aspects.

Many activities in SOA and distributed systems in general require a Publish/Subscribe (P/S) approach, where subscribers register to receive events produced by a publisher. We know of at least one pattern for Publish/Subscribe [Bus96], which has a more general context, our pattern is explicitly intended for distributed systems.

Our patterns are mainly intended for web services applications and distributed systems architects and designers. In those applications, the ESB and the P/S are architectural units that need to be combined with other architectural units. This work requires an understanding of their functions as well as non-functional aspects (e.g. security or performance). Typically, these units would not be built from scratch but the architect would select a commercial offering and the patterns would help guide this selection. However, the patterns may be also useful to those who build web service products and to the end users of this type of applications.

Section 2 presents the Enterprise Service Bus pattern, while Section 3 describes the Distributed Publish/Subscribe pattern. We end with some conclusions.

2. Enterprise Service Bus.

Intent

Provide a convenient infrastructure to integrate a variety of distributed services and related components in a simple way.

Example

A travel agency interacts with many services to do flight reservations, check hotel availability, check customer credit, and others. This interaction is being done now by direct interaction, which results in many ad hoc interfaces, and requires many format conversions. The system is not scalable and it is hard to support standards.

Context

Distributed applications using web services, as well as related services such as directories, databases, security, and monitoring. There may be also other types of components (J2EE, .NET). There may be different standards applying to specific components and components that do not follow any standards.

Problem
When an organization has many scattered services, how can we aggregate them so they can be used together to assemble applications, at the same time keeping the architectural structure as simple as possible, and apply uniform standards?

The solution to this problem is affected by the following **forces**:

- *Interoperability.* It is fundamental for a business unit in an institution to be able to interact with a variety of services, internal or external.
- *Simplicity of structure*: we want a simple way to interconnect services; this simplifies the work of the integrators.
- *Scalability:* we need to have the ability to expand the number of interconnected services without making changes to the basic architecture.
- *Message flexibility*: we need to provide a variety of message invocation styles (synchronous and asynchronous) and formatting. We can thus accommodate all component needs.
- *Simplicity of management*: we need to monitor and manage many services, perform load balancing, logging, routing, format conversion, and filtering.
- *Flexibility:* New types of services should be accommodated easily.
- *Transparency*: we should be able to find services without needing to know their locations.
- *Quality of service*: we may need to provide different degrees of security, reliability, availability, or performance.
- *Use of policies*: we need a policy-based configuration and management. This allows convenient governance and systematic changes. Policies are high-level guidelines about architectural or institutional aspects and are important in any system that supports systematic governance [Sch06].
- *Standard interfaces*: we need explicit and formal interface contracts.

Solution
Introduce a common bus structure that provides basic brokerage functions as well as a set of other appropriate services. Figure 1 shows a typical structure. One can think of this bus as an intermediate layer of processing that can include services to handle problems associated with reliability, scalability, security, and communications disparity. An ESB is typically part of a Service-Oriented Architecture Implementation Framework, which includes the infrastructure needed to implement a SOA system. This infrastructure may also include support for stateful services.

Structure
Figure 2 shows the class diagram of the ESB pattern. The **ESB** connects **Business Services** with each other providing support for the needs of these services through a Service Infrastructure made up of **Business Application Services (BASs),** which in turn use **Internal Services** to perform their functions. BASs are accesses through **Service Interfaces (SIs).**

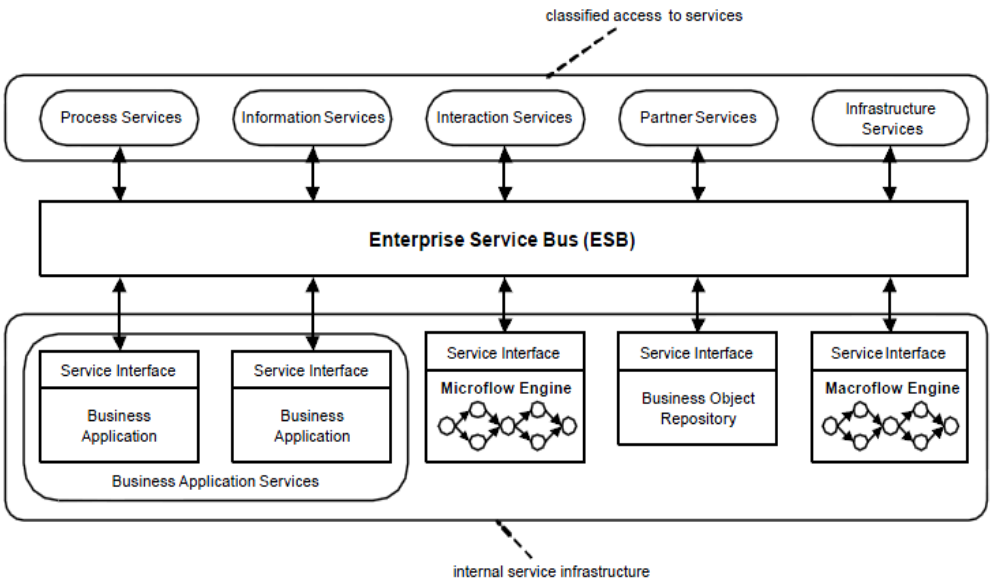

Figure 1: Enterprise Service Bus. (from [Zdu06]).

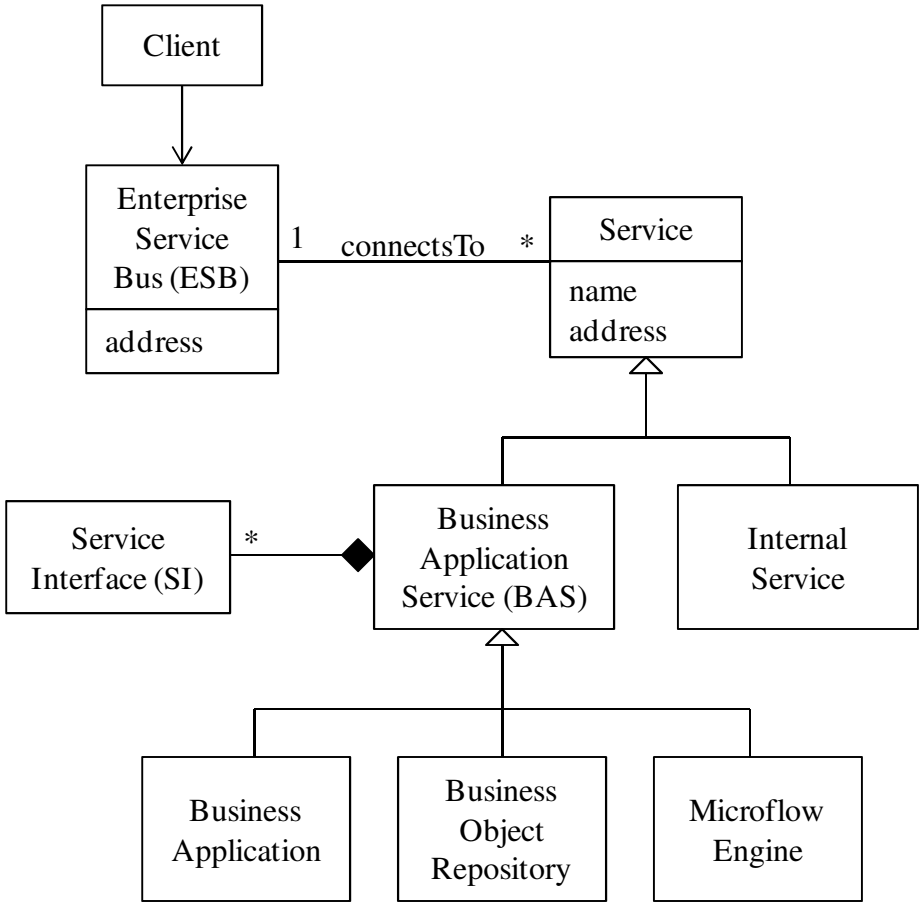

Figure 2. Class model of ESB pattern

Dynamics.
Figure 3 shows the sequence to access a service.

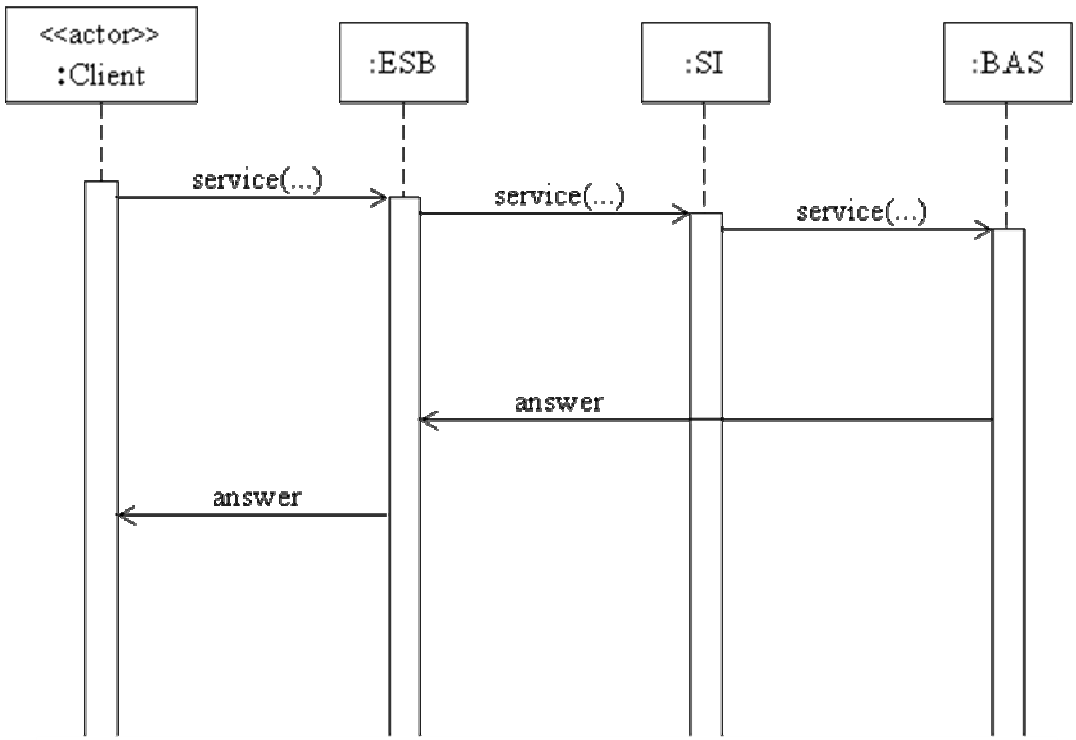

Figure 3. Service Access.

Implementation
The ESB itself is an example of a SOA architecture since it performs its functions using internal services.

An important decision is supporting stateful services or not. Stateless services are easier to design and manage but there are some applications that require stateful execution.

Known uses

- BEA AquaLogic Service Bus, now Oracle Service Bus, has operational service-management. It allows the interaction between services, routing relationships, transformations, and policies [BEA].

- WebSphere Application Server [Sph]. IBM's Business Integration Reference Architecture consisting of products from the WebSphere family. The Service Provider Delivery Environment (SPDE) architecture is an implementation of this reference architecture for the Telecommunications industry [WSE].

- Microsoft BizTalk Server [Biz]. Microsofts's Reference Architecture also uses ESBs.

- Mule ESB Enterprise is a supported version of the open source product Mule ESB [Mul]. [Swa08] shows its use to integrate web services written in Java and Ada using SOAP and REST protocols with an Ada web server.

Variants
According to [Fer], the ESB will evolve into an *Internet Bus*.

The *Secure Broker* has access control for web services, secure channels, and logging [Mor06].

Consequences
This pattern provides the following benefits:

- *Interoperability.* The ESB through its architecture and use of adapters provides a way to interact with a variety of services, internal or external.
- *Simplicity of structure*: much simpler than point-to-point or any other interconnection structure.
- *Scalability:* the number of interconnected services can be increased easily.
- *Message flexibility*: we can provide a variety of message invocation styles (synchronous and asynchronous) by using different message patterns.
- *Flexibility:* New types of services can be accommodated easily since they only need to conform to the interface standards.
- *Simplicity of management*: we can centralize the functions of monitoring and management of services, as well as any other needed functions.
- *Transparency*: we can find services conveniently by having lookup services.
- *Quality of service*: by using appropriate associated services we can provide different degrees of security, reliability, availability, or performance.
- *Use of policies*: we can use institution policies for configuration and management. This allows convenient governance and systematic changes. Security policies can define rights for the users with respect to the services.
- *Standard interfaces*: we can define explicit and formal interface contracts that must be followed by all aggregated functions.

Liabilities include:

- Extra overhead compared to point-to-point, because of the indirection involved and the overhead of the ESB itself.
- The bus is a single point of failure, but this can be overcome using redundancy
- A common interface standard may not be the most convenient for some services. Some applications may need more functions or parameters to interact with others than the ones defined in the common interfaces. Designing such a common interface may not be easy either.
- The bus may hide component dependencies.

Related patterns

The ESB is a type of Message Channel and it is also closely related to the Message Bus pattern, both described in [Hop04]. Because of its role as a communicator, the ESB is related to a variety of patterns that provide communication or adaptation. The ESB can be seen also as a microkernel in that it forwards client requests to a set of services [Bus96].

The Enterprise Service Bus can be considered a composite pattern comprised of the following patterns [Erl09]:

- The (Service) Broker pattern which itself is a composite pattern that consists of a set of integration-centric patterns used to translate between incompatible data models, data formats, and communication protocols [Bus96].
- Asynchronous Queuing pattern which establishes an intermediate queuing mechanism that enables asynchronous message exchanges and increases the reliability of message transmissions when service availability is uncertain.
- Intermediate Routing pattern which provides intelligent agent-based routing options to facilitate various runtime conditions.

Adapters are necessary to connect some services to the bus because their interfaces may not follow the standard interface defined in the bus architecture. Database systems will typically need an Adapter [Gam94].

A Repository for web services and objects is usually attached to and used by the ESB [Gar10]..

Microflows and macroflows can be realized using a Process Manager [Hop04].

A Lookup pattern may be used to find a specific service or a service of some given type [Kir04]

Mediator. Encapsulates how a set of objects interacts [Gam94].

Security Logger, intended to keep track of security-sensitive actions [Fer11].

[Chat04] considers several channel patterns, including point–to-point but not bus channels. His patterns are mostly ideas, without much detail.

[Erl09] considers also Asynchronous Queuing, Event-Driven Messaging, and other patterns.

The Publish/Subscribe pattern can perform its communication functions using an ESB. Inversely, the Publish/Subscribe pattern adds a way for the ESB to communicate events to its services.

Figure 4 shows how some of these patterns are related.

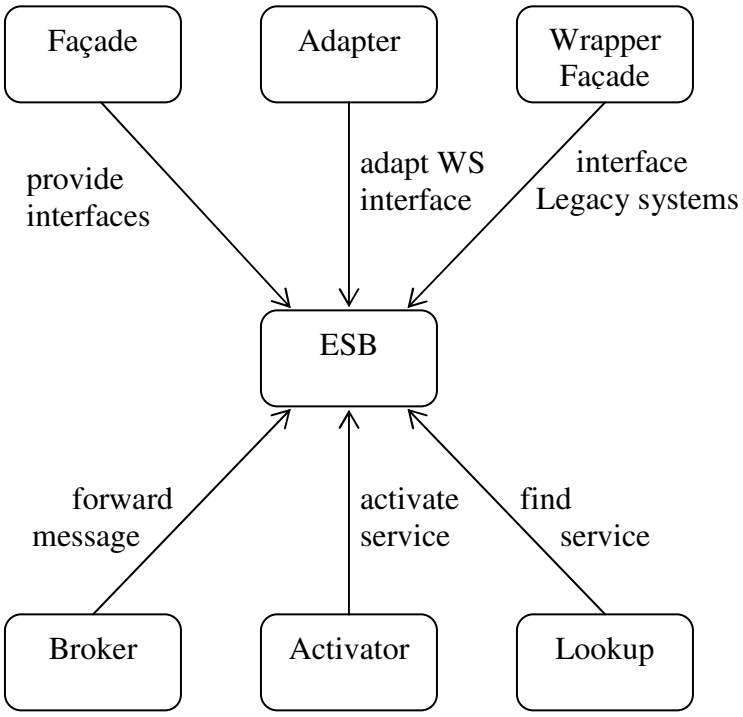

Figure 4: Some of the related patterns of the ESB.

3. Distributed Publish/Subscribe

Intent
In a distributed system, decouple the publishers of events from those interested in the events (subscribers).

Context
Distributed applications using web services, as well as related services such as directories, databases, security, and monitoring. There may be also other types of components (J2EE, .NET). There may be different standards applying to specific components and components that do not follow any standards.

Problem
Having each client call a publisher to find out if they have something of interest to them is inefficient and non-scalable. Also, more than one client could be interested in the same events. How do we organize publishers and subscribers in a more efficient way?

The solution to this problem is affected by the following **forces:**

Interoperability: It is useful for a business or institution to send or recive events from many other places.

Freedom: Clients only need to register to receive some events; after this they can go on their own businesses.

Dynamicity: The number of clients may be dynamic and customers may change their interests along time. We should allow clients come and go and to receive different types of events along time.

Scalability: We should be able to add an arbitrary number of clients without needing to redo the system.

*Loose coupling***:** Publishers are loosely coupled to subscribers, and need not even know of their existence. This allows both publishers and subscribers to evolve independently.

Location Transparency: the subscribers may be remote to the publisher and neither subscribers nor publishers need to know each other's locations.

Security: if events are sensitive we may need to protect the confidentiality of their communication. The sender of events needs also to protect its own information.

Selectivity: there should be different criteria to select the published events.

Role changing: A sender of events may also need to receive events.

Solution
Use an event channel where publishers send their events and interested subscribers can receive the events. Subscribers register for the events on which they are interested.

Structure
Figure 5 shows the participant classes. **Subscribers** can register to receive specitic events. Their conditions are described in the class **Subscription**. The **Channel** represents different ways of publishing events.

Dynamics
Figure 6 shows a sequence diagram for the use case Publish Event. Other use cases include Register Subscriber and Remove Subscriber.

Implementation
For event transmission it is possible to use push and pull approaches----this is incorrect in POSA: the queue is an implementation aspect that belongs to a lower level, one can also use a broker for example. The event channel can be any type of asynchronous channel and may use an ESB. Subscribers usually receive only a subset of the total messages published. The process of selecting messages for reception and processing is called

filtering. There are two common forms of filtering: topic-based and content-based [wik]. An example of implementation is given in [Rou02].

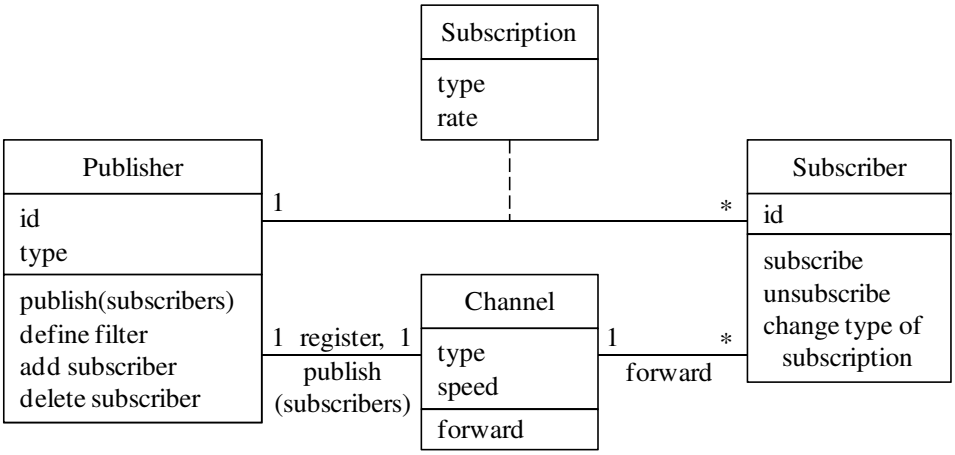

Figure 5 Class diagram for the Publish/Subscribe pattern

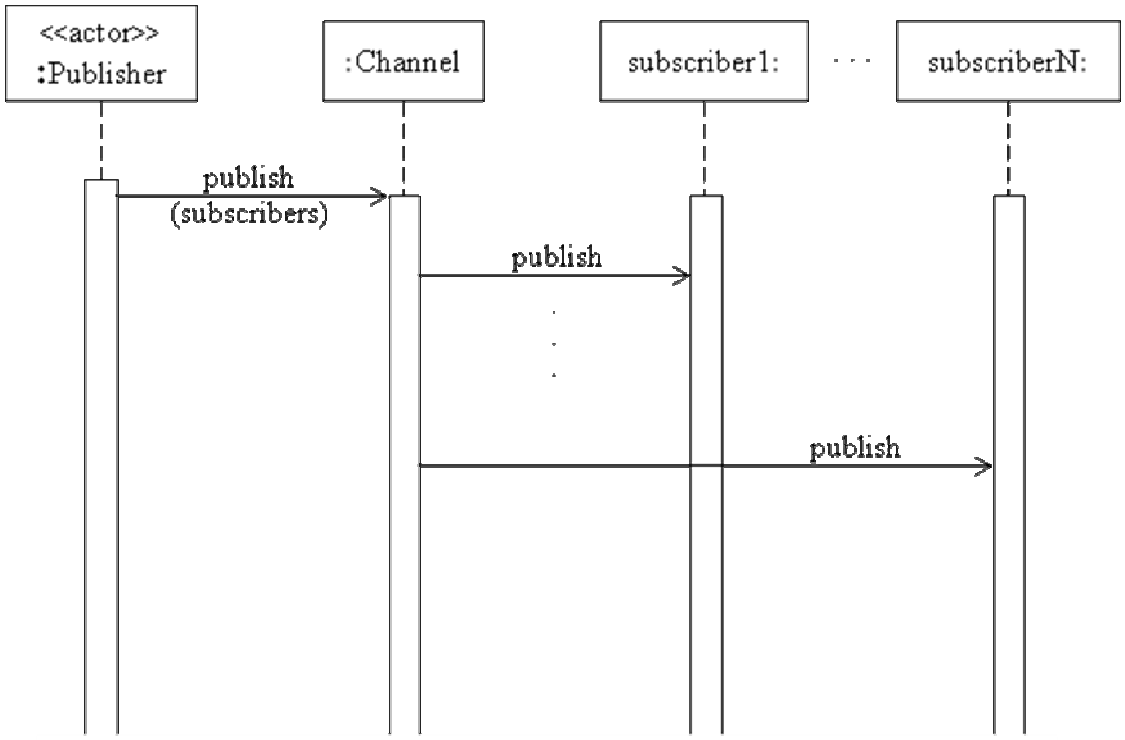

Figure 6. Sequence diagram for use case Publish Event.

Known uses

- The IBM MQSeries provides guaranteed, once-only delivery of messages between IT

systems. It can connect different types of platforms, including those from IBM, Microsoft, Sun, and HP using a variety of communications protocols [MQ].

- Software AG has an Integrator Server that distributes documents using a Broker as publishing channel [Sag09].

- Oracle uses Publish/Subscribers in conjunction with their database architectures [Ora02].

Variants

If we add security mechanisms we can define a **Secure Publish/Subscribe**; which uses the secure Channel pattern for event channel, uses RBAC pattern for control of contents, provides mutual authentication, and includes logging.

[Cor06] describes variants based on the type of service provided: topic-based, content-based, concept-based, and type-based.

We can also define a P/S focusing on its functional or conceptual aspects, not in its software realization as we have done here.

Consequences

The pattern presents the following **advantages** :

Interoperability. Because of its decoupling effect, this pattern allows the interaction of any type of publishers and subscribers.

Freedom: Subscribers only need to register to receive some events; after this they can go on their own businesses and they are notified when there is something new.

Dynamicity: We cadd or remove subscribers at any time. Subscribers can also change their interests by changing their type of subscription.

Scalability: The number of subscribers can be extended by just extending the subscriber list as far as we have appropriate communication channels for the events.

*Loose coupling***:** Publishers can work without knowledge of their subscriber details and vice versa. As far as their interfaces remain constant, both can change independently.

Location Transparency: Neither subscribers nor publishers need to know each other's locations, a lookup service can find their locations.

Security: if events are sensitive we can encrypt the event channel. We can also use digital signatures for authenticity. See the description of the Secure P/S variant.

Selectivity: it is possible for the clients to select the published events according to different criteria, e.g., topic-based, content-based, concept-based, and type-based [Cor06].

Role changing: Publishers and subscribers are just roles that can be taken by any entity.
.
Possible **liabilities** include:

- There is some overhead in the event structure,, i.e. a tight coupling of subscribers to their publishers would have better performance at the cost of flexibility.

- There may be coordination problems because of the decoupling.

Related patterns

- Broker. A Broker can be used as distribution channel. It typically includes a look up service and can distribute events to subscribers in a transparent way.

- The Secure Channel Communication pattern [Bra98], supports the encryption/decryption of data. This pattern describes encryption in general terms. It does not distinguish between asymmetric and symmetric encryption. Another version is given in [Sch06].
- [Erl09] considers also Asynchronous Queuing, Event-Driven Messaging, and other message patterns that could be used in the P/S Channel.

- ESB. An ESB includes all the services needed for the P/S functions and uses the P/S functions for its own functions.

- A Party can be responsible for some Subscribers. Parties can be Individuals or Institutions. The Party pattern is useful to define general subscribers [Fow97]

- The Subscriber is a type of Observer [Gam94], in that it receives updates from the publisher.

- The Lookup pattern is used to locate remote publishers or subscribers [Kir04].

4. Conclusions

We have presented two common patterns used in distributed systems. The ESB has been used mostly for web services but it can be used for any distributed system. The P/S has been described usually in a centralized environment and we have emphasized here its distributed nature. We have provided enough detail for an application designer or integrator to make good use of the functionality of these patterns. Of course, there are many other patterns that are needed to have a catalog really useful for designers.

We are writing these two patterns considering only their functional/conceptual aspects. Those are abstract versions of the patterns presented here, leaving out their software aspects.

Acknowledgements

We thank our shepherd Philipp Bachmann for his precise and careful comments that significantly improved the quality of our paper.

References

[BEA] BEA Aqualogic Service Bus, http://en.wikipedia.org/wiki/AquaLogic (retrieved June 27, 2011)

[Biz] SOA Patterns with BizTalk Server 2009, http://www.packtpub.com/soa-patterns-with-biztalk-server-2009/book (retrieved on July 13, 2011)

[Bra98] Braga, A., Rubira, C., and Dahab, R. 1998. Tropyc: A pattern language for cryptographic object-oriented software. Chapter 16 in *Pattern Languages of Program Design 4* (N. Harrison, B. Foote, and H. Rohnert, Eds.). Also in Procs. of PLoP'98, DOI= http://jerry.cs.uiuc.edu/~plop/plop98/final_submissions/

[Bus96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerland, and M. Stal., *Pattern-oriented software architecture*, Wiley 1996.

[Chap04] David A. Chappell, *Enterprise Service Bus,* O'Reilly, 2004

[Chat04] Soumen Chatterjee, "Messaging patterns in Service-Oriented Architectures" http://msdn.microsoft.com/en-us/library/aa480027.aspx

[Cor06] Angelo Corsaro, "Quality of service in Publish/Subscribe middleware", http://www.omgwiki.org/dds/sites/default/files/Quality_of_Service_in_Publish-Subscribe.pdf

[Erl09] Thomas Erl, *SOA Design Patterns*, Prentice Hall PTR; 1st edition, 2009

[Fer] D.F. Ferguson, D. Pilarinos, and J. Shewchuck, "The Internet Service Bus", *The Architecture Journal* 13, http://www.architecturejournal.net

[Fer11] E.B.Fernandez , Sergio Mujica, and Francisca Valenzuela, "Two security patterns: Least Privilege and Secure Logger/Auditor.", *Procs.of Asian PLoP 2011.*

[Fow97] M. Fowler, *Analysis patterns -- Reusable object models*, Addison- Wesley, 1997.

[Gam94] E. Gamma, R.Helm, R.Johnson, and J.Vlissides. *Design patterns: elements of reusable object-oriented software,* Boston, Mass:Addison-Wesley, 1994.

[Gar10] J.P. Garcia-Gonzalez, Veronica Gacitua, and C. Pahl, “Service registry : a key piece for enhancing reuse in SOA service oriented architecture”, *The Architecture Journal*;21,Microsoft, 2010. 29-36.

[Hop04] G. Hoppe and B. Woolf, *Enterprise integration patterns: Designing, building, and deploying message solutions,* Addison-Wesley 2004.

[Kir04] M. Kircher and P. Jain, *Pattern-oriented software architecture, vol. 3: Patterns for resource management,* J. Wiley & Sons, 2004.

[Mor06] P. Morrison and E.B.Fernandez, "Securing the Broker pattern", *Procs. of the 11th European Conf. on Pattern Languages of Programs (EuroPLoP 2006)* http://www.hillside.net/europlop/

[Mul] MuleSoft, Mule Enterprise Service Bus, http://www.mulesoft.com/mule-esb-open-source-esb

[MQ] http://www.redbooks.ibm.com/redbooks/pdfs/sg246282.pdf

[OAS] OASIS, Reference Model for Service Oriented Architecture, http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf

[Ora02] Oracle, “Using the Publish-Subscribe model for applications”, http://download.oracle.com/docs/cd/B10501_01/appdev.920/a96590/adg15pub.htm

[Pap07] M. P. Papazoglou, and W.-J. Heuvel, ‘Service oriented architectures: approaches, technologies and research issues’, *The VLBD Journal*, Vol 16, No 3, 2007, 389—415.

[Rou02] P. Rousselle, “Implementing the JMS Publish/Subscribe API”, Dr. Dobbs Journal, April 2002, 28-32.

[SAG] Software AG webMethods Integrator Server, December 2009 http://documentation.softwareag.com/webmethods/wmsuites/wmsuite8_ga/Developer/Guides/8-0-SP1_Publish_Subscribe_Developers_Guide.pdf

[Sch06] M. Schumacher, E. B.Fernandez, D. Hybertson, F. Buschmann, and P. Sommerlad, *Security Patterns: Integrating security and systems engineering",* Wiley 2006. Wiley Series on Software Design Patterns.

[Sph] Patterns: SOA with an Enterprise Service Bus in WebSphere Application Server V6. http://www.redbooks.ibm.com/redpieces/abstracts/sg246494.html

[Swa08] R. E. Sward, K. J. Whitacre; “A multi-language service-oriented architecture using an enterprise service bus”, *Proceedings of the 2008 ACM Annual International Conference on SIGAda,* October 26–30, 2008, Portland, Oregon, USA.

[wik] Wikipedia, “Publish/subscribe”, http://en.wikipedia.org/wiki/Publish/subscribe

[WSE] Solution design in WebSphere Process Server and WebSphere ESB http://www.ibm.com/developerworks/websphere/library/techarticles/0908_clark/0908_clark.html

[Zdu06] U. Zdun, C. Hentrich, and W.M.P. van der Aalst, ‘A Survey of Patterns for Service-Oriented Architectures’, *Int. Journal of Internet Protocol Technology*, Vol 1, No 3, 132—143, 2006

[Zim05] O. Zimmermann, V. Doubrovski, J. Grundler, and K. Hogg, “Service-oriented architecture and business process choreography in an order management scenario: Rationale, concepts, lessons learned”, *Procs OOPSLA*, ACM, 2005.