An Analysis of Notification Related Specifications for Web/Grid applications

Shrideep Pallickara and Geoffrey Fox
(spallick, gcf)@indiana.edu
Community Grids Laboratory, Indiana University

Abstract

Notification is especially important in the Service Oriented Architecture (SOA) model engendered by Web Services. where Web Services interact with each other through the exchange of messages. In this paper we compare and contrast two competing specifications in the area of notifications. The first one, WS-Notification, is part of the Web Service Resource Framework (WSRF). The second one is the WS-Eventing specification. These specifications are expected to have far reaching implications on the development of asynchronous, complex, dynamic systems.

Keywords: notifications, publish/subscribe, middleware systems, Web Services, Grid Services, WSRF

1. Introduction

Messaging is a fundamental primitive in distributed systems. Entities communicate with each other through the exchange of messages, which can encapsulate information of interest such as application data, errors and faults, system conditions, search and discovery of resources. A related concept is that of *notifications* where entities receive messages based on their registered interest in certain occurrences or situations. Messaging and notifications are especially important in the Service Oriented Architecture (SOA) model engendered by Web Services. Here, Web Services interact with each other through the exchange of messages.

In this paper we compare and contrast two competing specifications in the area of notifications. The first one, WS-Notification [1], is part of the Web Service Resource Framework (WSRF) [2]. WSRF is a realignment of the dominant Open Grid Service Infrastructure [3, 4] to be more in line with the emerging consensus [5] within the Web Services community. The second one is the WS-Eventing [6] specification. These specifications are expected to have far reaching implications on the development of asynchronous, loosely-coupled, dynamic systems.

This paper is organized as follows. In section 2 we present some background information on notifications. In section 3 we describe related work in the area of notifications, which spans the gamut from distributed objects to peer-to-peer messaging systems. In section 4 we describe the message exchange patterns available in both WSDL 1.1 and WSDL 2.0. We then provide an overview of both WS-Notification and WS-Eventing in section 5. Comparison of the strategies and concepts provided in these specifications is included in section 6. In section 7 we identify problems stemming from issues not supported in either specifications. Finally, we include a strategy to federate between these specifications. Section 9 outlines our conclusions.

2. A background on notifications

There are two main entities involved in a notification: the *source* which is the generator of notifications and the *sink* which is interested in these notifications. A sink first needs to register its *interest* in a situation, this operation is generally referred to as a *subscribe* operation. The source first wraps *occurrences* into notification messages. Next, the source checks to see if the message satisfies the constraints specified in the previously registered subscriptions. If so, the source routes the message to the sink. This routing of the message from the source to the sink is referred to as a notification.

It should be noted that there could be multiple sources and sinks within the system. Furthermore, each sink could register its interests with multiple sources, while a given source can manage multiple sinks. The complexity of the subscriptions registered by a sink could vary from simple strings such as “Weather/Warnings” to complex XPath or SQL queries.

Typically a source comprises two distinct roles: *producer* and *publisher*. A producer is responsible for packing occurrences into notification messages, while the publisher is responsible for publishing these notifications. Similarly, a sink comprises two distinct roles: *subscriber* and *consumer*. The subscriber is responsible for registering the consumer’s interests with a source, while the consumer is responsible for consuming notifications received from a source.

Depending on the nature of the underlying frameworks the coupling between the sources and sinks can vary. In loosely-coupled systems a source need not be aware of the sinks: the source generates events and an intermediary, typically a messaging middleware, is responsible for routing the message to appropriate sinks. In tightly-coupled systems there is no intermediary between the source and the sink.

3. Related Work

The area of notifications and messaging in distributed systems has been very well studied. Here we briefly review efforts in the areas of distributed objects, queuing systems, publish-subscribe systems and finally P2P style messaging.

3.1. Distributed objects

The CORBA Event Service [7] approach adopted by the OMG is one of establishing channels and subsequently registering suppliers and consumers to the event channels. In the CORBA Event Service suppliers, consumers and the event channel itself are all distributed objects. Furthermore, both suppliers and consumers can choose one of two modes – push or pull – to interact with the event channel. The Notification Service [8] addresses limitations pertaining to the lack of event filtering capability in the CORBA Event Service. TAO [9] is a real-time event service that extends the CORBA event service and provides rate-based event processing, efficient filtering and correlation.

3.2. Message queuing systems

Message queuing systems such as IBM MQSeries [10] and Microsoft's MSMQ [11] involve the creation of queues that are statically pre-configured to forward messages from one queue to another. Queuing systems employ the *store-and-forward* approach with a queue storing messages to a stable storage before forwarding them to another queue. Typically, these systems do not allow the specification of subscription constraints for delivery and are generally deployed in systems where the interests are static and the delivery requirements are stringent.

3.3. Publish subscribe systems

In publish/subscribe systems the routing of messages from the publisher to the subscriber is within the purview of the message oriented middleware (MOM), which is responsible for routing the right content from the producer to the right consumers. Publish/Subscribe systems provide a clear decoupling of the message producer and consumer roles that interacting entities might have. This is especially useful if there are a large number of potential consumers for a given message. In such cases a producer need not keep track of the large number of consumers that a message could potentially be routed to. The middleware performs this function for the publisher. Examples of messaging infrastructures based on the publish/subscribe paradigm include NaradaBrokering [12, 13], Gryphon [14], Elvin [15] and Sienna [16]. Different systems allow for different subscription constraints. For e.g. in NaradaBrokering one can specify SQL, XPath and Regular expression queries as part of subscriptions.

3.4. Peer to Peer systems

Peer to peer (P2P) style messaging involves peers interacting directly with each other. Some peer interactions may traverse through multiple peers before reaching the targeted peer. Several P2P systems use a simple forwarding approach, with the propagations being attenuated by the use of TTL (time-to-live) indicators. Other systems such as FLAPPS [17] provide a generalized infrastructure for peer network design with peers being organized into a peer network comprised of overlapping peer groups with transit peers efficient routing requests. Routing here is quite efficient. In Distributed Hash Table (DHT) based P2P overlay networks the nodes are organized based on the content that they possess. This is then used to locate, distribute, retrieve and manage data in these settings. Examples of DHT based systems include Pastry [18], Squid [19] and JXTA [20]. One advantage of these DHT-based systems is that the number of *hops* for communications is bounded.

4. WSDL and Message Exchange Patterns

Messaging is fundamental to Web Services, and WSDL [21] which describes these services facilitate the description of various message exchange patterns (hereafter MEP) that are possible between service endpoints. Since these MEPs are defined to be part of the WSDL document, any node wishing to interact with the service knows both the *sequence* and the *cardinality* of messages associated with a given WSDL operation. WSDL 1.1 defined a basic set of MEPs; this has been expanded upon in WSDL 2.0.

WSDL 1.1 describes four MEPs defining the sequence and cardinality of abstract messages –- *In, Out, Fault* – that are part of a WSDL operation. The MEPs governing the exchanges between a service **S** and a node **N** are *one-way*, *request/response*, *notification* and *solicit*. A one-way message comprises a single *Out* message from a service **S** to node **N**. A request/response comprises an *In* message sent by a node **N** that is followed by an *Out* message by the service **S**. The notification MEP is simply an *Out* message from a service **S** to a node **N**. Finally, a solicit MEP is an *Out* message from service **S** followed by an *In* message from node **N**. It must be noted that the *Out* message in the notification MEP and the *In* message in the solicit MEP can also be a Fault message.

WSDL 2.0 has defined 4 additional MEPs *Robust In-Only*, *In-Optional-Out*, *Robust Out-Only* and *Out-Optional-In* which are extensions to the four MEPs that were defined in WSDL 1.1. These patterns occur because of the new fault propagation rules that are part of WSDL 2.0. The MEPs with the *optional* tag within them are patterns that comprise one or two messages, with the second message being a *Fault* that was triggered because of the first message in the pattern. The MEPs with the *robust* tag within them are patterns with exactly one message, however a fault may be triggered because of the first message.

5. An Overview of WS-Notification and WS-Eventing

In this section we provide an overview of WS-Notification and WS-Eventing. Both these specifications leverage WSDL, SOAP [22] and WS-Addressing [23] in their specifications. These specifications outline a set of SOAP message exchanges between various components

5.1. WS-Notification

The WS-Notification specification refers to a set of specifications comprising WS-BaseNotification [24], WS-Brokered Notification [25] and WS-Topics [26]. WS-BaseNotification standardizes exchanges and interfaces for producers and consumers of notifications. WS-Brokered Notification facilitates the deployment of Message Oriented Middleware (MOM) to enable brokered notifications between producers and consumers of the notifications. WS-Topics deals with the organization of subscriptions and defines dialects associated with subscription expressions; this is used in the conjunction with exchanges that take place in WS-BaseNotification and WS-Brokered Notification. WS-Notification currently also uses two related specifications from the WSRF specification; WS-ResourceProperties [27] to describe data associated with resources, and WS-ResourceLifetime [28] to manage lifetimes associated with subscriptions and publisher registrations (in WS-BrokeredNotifications).

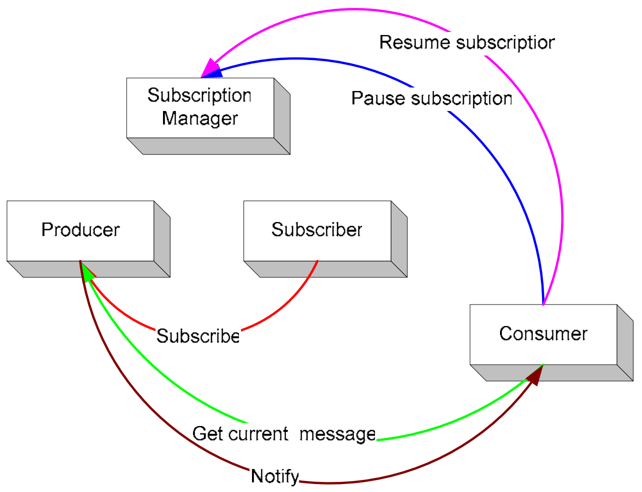

Figure 1: WS-BaseNotification - Chief components

Figure 1 depicts the chief components of the WS-BaseNotification specification. Also, depicted in this figure are the interactions (along with the directions) that these components have with each other. In WS-BaseNotification, a subscriber registers a consumer with a producer, which in turn includes information regarding the subscription manager in its response. Consumers can pause and resume subscriptions, with no messages being delivered while the subscription is in a paused state. Resumption of subscriptions after a pause can entail replay of all notifications that occurred in the interim. After a disconnect, either due to a scheduled downtime or failure, a consumer may also retrieve the last message issued by a producer. Finally, notifications from the producer are issued directly to the consumer. In WS-Notification each subscription is considered to be a resource (more appropriately a WS-Resource [29]). A consumer can use WS-ResourceLifetime or WS-ResourceProperties to manage lifetimes and properties associated with these subscriptions.

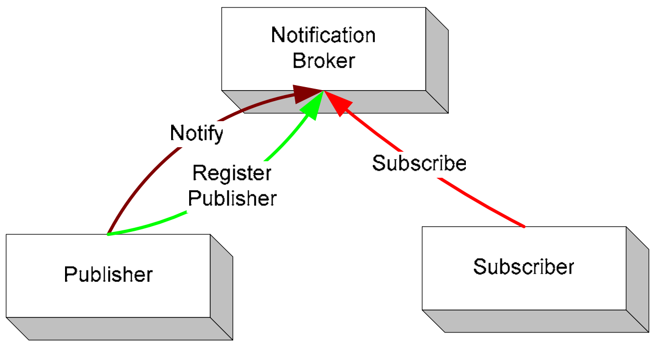

Figure 2: WS-BrokeredNotification - Chief components

Figure 2 depicts the chief components of the WS-BrokeredNotification specification. The notification broker interface performs the function of an intermediary between the producers and consumers of content. The broker is responsible for managing the subscriptions and also for routing the notifications to the subscriber. Furthermore, the broker also maintains a topic space (based on the WS-Topics specification) that allows consumers to review the list of topics to which publishers publish. It should be noted that each topic is also a resource and can be inspected for its properties such as *dialect* and *topic expressions*.

5.2. WS-Eventing

Figure 3 depicts the chief components in WS-Eventing. When the sink subscribes with the source, the source includes information regarding the subscription manager in its response. Subsequent operations –- such as getting the status of, renewing and unsubscribing –- pertaining to previously registered subscriptions are all directed to the subscription manager. The source sends both notifications and a message signifying the end of registered subscriptions to the sink.

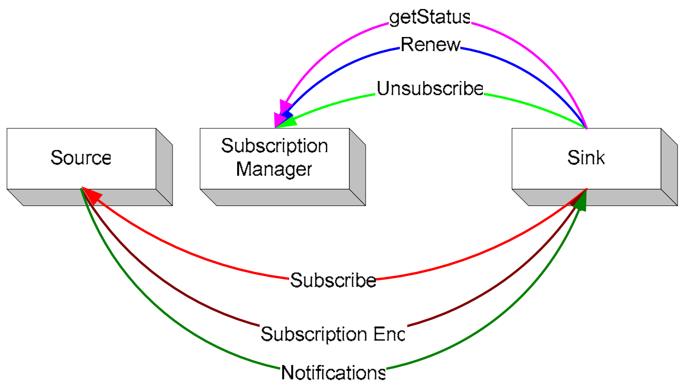

Figure 3: WS-Eventing - Chief components

6. Comparisons & differences in approaches

In this section we compare the difference in the approaches and philosophies towards some of the key concepts in these specifications. Our comparisons are based on the following key points.

- Complexity of specifications
- Notifications of messages
- Delivery Modes
- Subscription operations
- Topic Space management
- Publishing

Subsequent subsections elaborate these issues in detail. It should be noted that both these specifications recommend the security strategies outlined by the WS-Security suite of specifications. Table 1 in the appendix summarizes our comparisons.

6.1. Complexity of specifications

WS-Notification is a complex specification comprising three other specifications viz. WS-BaseNotification, WS-BrokeredNotification and WS-Topics. Furthermore several elements (such as subscriptions and topic spaces) are also resources (WS-Resource) as outlined in the WSRF suite of specifications. In their role as resources these aforementioned elements also need to support inspection and modification of the associated properties and lifetimes as outlined in the WS-ResourceProperties and the WS-ResourceLifetime specifications respectively. WS-Eventing on the other hand is a self-contained specification.

6.2. Notification of messages

WS-Notification provides support for both a *Notify* message as well as *raw* application-specific messages. A subscriber can specify either one these two types of messages that it is interested in receiving. The Notify message type also encapsulates topic information within them. This is especially useful in allowing a consumer to identify the sub-processes responsible for dealing with specific topics. For e.g. a consumer may be written in such a way that different modules handle processing related to different topics. The WS-BrokeredNotification specification also provides support for loosely-coupled interactions since a publisher need not keep track of all its consumers.

WS-Eventing on the other hand provides support only for raw application specific messages. The specification does not outline any specific element for encapsulating the notifications within the body of SOAP messages. WS-Eventing notifications do not encapsulate any topic information within them.

6.3. Delivery modes

WS-Notification currently only outlines the push delivery mode for notifications. The push model is one in which notifications are pushed to the consumer. An advantage of the push model is that notifications are routed to the consumer as soon as they are available. WS-Notification however incorporates support for delegated delivery of notifications. Here an intermediary, the broker, can push notifications to the consumer.

WS-Eventing also outlines the push model for notifications. A related specification from Microsoft and Intel, WS-Management [30] outlines three other modes for delivery: *batched*, *pull* and *trap*. The first mode, batched, allows an event source to batch multiple notifications into a single SOAP envelope. This is an way effective of reducing the number of notifications from a high volume notification source without sacrificing too much on timeliness. The second mode is the pull mode; here a sink is responsible for polling the source at regular intervals and pulling notifications if any are available. Though a sink may not receive notifications instantly, one advantage of the pull mode is that a sink is always in control of the rate at which it process notifications. One disadvantage of the pull mode is the need for continuous polling. The final mode, the trap mode, leverages the SOAP over UDP specification and indicates that the sink is interested in receiving notifications over UDP. It should be noted that in these extended modes, individual SOAP messages *are* expected to include information regarding the subscription that triggered the receipt of these notifications.

6.4. Subscription operations

Both specifications provide support for delegated management of subscriptions through the Subscription Manager interface. Furthermore, the specifications also allow the specification of XPath constraints to *filter* notifications. There are however a few differences in aspects related to subscriptions.

In WS-Notification the subscription related operations include *subscribe*, *pause* and *resume*. Pause and resume relate to the ability to suppress receipt of notifications in the intervening period between them. WS-Notification also includes support for retrieving the last message that was published by a publisher on a given topic. The specification also allows consumers to modify their termination times. It should be noted that there is no operation for unsubscribe. Instead, the WS-Notification specification expects consumers to adjust the time for expiration of the subscription resource as governed by the WS-ResourceLifetime specification. This is a problematic issue since an unsubscribe operation is semantically different from the expiry of a subscription.

In WS-Notification there is also no exchange which announces the end of a subscription to a consumer. This is especially important since the expiration times are based on the time at the publisher; there is thus no way for a consumer to know that it is not receiving notifications in case the clocks at the publisher and consumer are out of sync (which will most likely be the case). Finally, the filter expressions supported within the subscriptions include XPath.

In WS-Eventing the subscription related operations include *subscribe*, *renew*, *unsubscribe* and *subscription-end*. The renew operation relates to the ability to extend the lifetime of a subscription. There is also a separate unsubscribe method which allows a sink to unsubscribe its previously registered interests. A sink receives a Subscription End notification either as a result of the lifetime expiring or an unsubscribe operation. Though the WS-Eventing specification does not support the pause-renew set of operations, the WS-Management specification facilitates this operation. There is no separate message in WS-Eventing to retrieve the last message published by a source, though this is not really needed if one has the pause-resume feature from WS-Management. XPath is the filter expression used within subscriptions.

6.5. Topic Space Management

WS-Notification includes a separate specification, WS-Topics, which deals with the management of a topic space. A topic space is essentially a collection of topics. The topic space also supports inspection based on the exchanges that are supported in the WS-ResourceProperties specification. The topic space facilitates hierarchical organization of the topics within the topic space, though nothing precludes a topic space from comprising only of root topics. The topic space also supports two wildcard operators, ***** and //, for the selection of topics within a topic tree. In WS-Notification there is also support for advertisements where a publisher publishes information regarding the topics that it will publish to. Furthermore, a consumer can also inspect the topics available at a producer through the Notification Producer interface. This consumer can also retrieve information regarding the topic expression dialects available at a publisher.

In WS-Eventing there is no formal specification regarding the management of topics. There is thus no support for hierarchical topics or the ability to navigate a topic space to retrieve topics of interest.

6.6. Publishing

In WS-Notification a publisher need not keep track of all the subscriptions or the routing of events to consumers. This task is performed by the broker intermediary. WS-Notification, also supports an important feature known as *on-demand* publishing. Here a publisher will publish or issue notifications only if there is at least one consumer which is interested in the receipt of these notifications. This features ensures that bandwidth and computational resources are not wasted in the creation and publishing of notifications that no one is interested in. This feature is also referred to as *quenching* in distributed middleware systems.

In WS-Eventing the source keeps track of all sinks, and is responsible for routing notifications to the sinks. WS-Eventing does not support delegation of routing related operations. Since a source always keeps track of all its consumers, the default mode is on-demand publishing. It should be noted that the same holds true for WS-BaseNotification.

7. Issues in the specifications

Subscriptions, in both WS-Notification and WS-Eventing, do not have a unique identifier associated with them. This means that if a consumer has its subscription registered twice, it would be considered as two separate subscriptions. This situation can easily arise if the subscription response to the first subscription request was lost, in which case a subscriber may issue the same request again. This situation results in the following problems related to the receipt of messages.

- ♦ Duplicate receipt of messages: A consumer will receive duplicate (corresponding to the number of duplicate subscriptions) copies of notifications from the producer. Since the consumer has no way of recognizing these duplicates, it may process these as separate notifications. In some cases depending on the application, this may result in unpredictable behavior. These specification themselves do not have any information which can be used to identify such duplicates. It is possible for a producer to include an additional field, the message identifier to circumvent this problem. But this feature would then need to be implemented in a proprietary manner by each system.
- ♦ Bandwidth utilization problems: Since every notification is being received multiple times the bandwidth utilization is not optimal. This problem is further exacerbated under conditions where the rate and size of these notifications are quite high.
- ♦ Management of expiration times. Since a consumer is aware of only one subscription that is registered at a producer, even after it unsubscribes/terminates the subscription in question, it will continue to receive

notifications as a result of the duplicated subscription(s). Once again this may result in unpredictable behavior at a consumer.

This problem can be assuaged if there were a way for consumers/sinks to retrieve the list of subscriptions registered at a producer/source. However, this operation is not supported in either specification.

8. Federation between the specifications

We believe that it is possible that these specifications might be deployed concurrently. Federation between these specifications will allow endpoints in these specifications to interact with each other. This would involve mapping the semantics of operations involved in these specifications. These operations need to be managed by a *middleware*. Here we briefly review some of the key issues involved. First, the operations related to subscriptions need to be mapped. Here, the requests to unsubscribe and to renew subscriptions in WS-Eventing should be mapped into the appropriate calls using WS-ResourceLifetime if needed. Second, the middleware also needs to maintain a list of properties that are automatically generated. This would enable WS-Eventing components to behave as WS-Resources that facilitate inspection of properties. Delivery modes supported in either specifications need to be mapped appropriately. Issues pertaining to pausing and renewing of subscriptions need to be handled by the federation module by appropriately keeping track of issued notifications.

8.1. Deployment of the federation module

To facilitate incremental addition of capabilities to service endpoints one can also configure *filters* (examples include filters for encryption, compression, logging etc.) in the processing path between the service endpoints. Since the service endpoints communicate using SOAP messages these filters operate on SOAP messages. Several of these filters can be cascaded to constitute a *filter pipeline*. In Java these filters are referred to as JAX-RPC *handlers*, in gSOAP they are referred to as *plugins;* while in Microsoft's WSE these are referred to as *filters*. The federation module can be implemented as a filter and configured during the deployment phase of the service in question. Note that this filter strategy while not entail any changes to the service implementations and applications using either specifications. Another deployment strategy is to implement the federation as a proxy, which receives messages and routes mapped messages appropriately.

9. Conclusions

In this paper we have analyzed and contrasted the two dominant specifications in the area of Web Services notifications. Depending on the needs of the application deployments can choose to leverage either of these specifications. Table 1 in the appendix summarizes some of our comparisons.

10. References

1. Web Services Notification (WS-Notification). IBM, Globus, Akamai et al. http://www-106.ibm.com/developerworks/library/specification/ws-notification/
2. The Web Services Resource Framework. http://www.globus.org/wsrf/
3. I. Foster, C. Kesselman, J. Nick, S. Tuecke, "The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration." Open Grid Service Infrastructure WG, Global Grid Forum, June 22, 2002. Available from http://www.globus.org/research/papers/ogsa.pdf.
4. The Open Grid Services Infrastructure (OGSI). http://www.gridforum. org/Meetings/ggf7/drafts/draft-ggf-ogsi-gridservice-23_2003-02-17.pdf
5. D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard, "Web Services Architecture." W3C Working Group Note 11 February 2004. Available from http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.
6. Web Services Eventing. Microsoft, IBM & BEA. http://ftpna2.bea.com/pub/downloads/WS-Eventing.pdf
7. The Object Management Group (OMG). OMG's CORBA Event Service. Available from http://www.omg.org/
8. The Object Management Group (OMG). OMG's CORBA Notification Service. Available from http://www.omg.org/
9. T.H. Harrison, D.L. Levine and D.C. Schmidt. The design and performance of a real-time CORBA object event service. Proceedings of the OOPSLA'97. Atlanta, GA.
10. The IBM WebSphere MQ Family. http://www-3.ibm.com/software/integration/mqfamily/
11. Microsoft Message Queuing. http://www.microsoft.com/windows2000/technologies/communications/msmq/default.asp
12. The NaradaBrokering Project. http://www.naradabrokering.org
13. Shrideep Pallickara and Geoffrey Fox. NaradaBrokering: A Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids. Proceedings of ACM/IFIP/USENIX International Middleware Conference Middleware-2003.
14. G. Banavar et al. An Efficient Multicast Protocol for Content-Based Publish-Subscribe Systems. In Proceedings of the IEEE International Conference on Distributed Computing Systems, Austin, Texas, May 1999.
15. Bill Segall, David Arnold, Julian Boot, Michael Henderson, and Ted Phelps. Content based routing with elvin4. In Proceedings AUUG2K, Canberra, Australia, June 2000.

16. Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In Proceedings of the 19th ACM Symposium on Principles of Distributed Computing, pages 219–227, Portland OR, USA, 2000.
17. B. Scott Michel, Peter L. Reiher: Peer-through-Peer Communication for Information Logistics. GI Jahrestagung (1) 2001: 248-256.
18. Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. Proceedings of Middleware 2001.
19. C. Schmidt and M. Parashar. Enabling Flexible Queries with Guarantees in P2P Systems, IEEE Network Computing, Special Issue on Information Dissemination on the Web, IEEE Computer Society Press, Vol. 8, No. 3, pp. 19- 26, May/June 2004.
20. Sun Microsystems. The JXTA Project and Peer-to-Peer Technology http://www.jxta.org
21. Web Services Description Language (WSDL) 1.1 http://www.w3.org/TR/wsdl
22. M. Gudgin, et al, "SOAP Version 1.2 Part 1: Messaging Framework," June 2003. http://www.w3.org/TR/2003/REC-soap12-part1-20030624/
23. Web Services Addressing (WSAddressing) ftp://www6.software.ibm.com/software/developer/library/ wsadd200403.pdf
24. Web Services Base Notification (WS-BaseNotification). IBM, Globus, Akamai et al. ftp://www6.software.ibm.com/software/developer/library/ws-notification/WS-BaseN.pdf
25. Web Services Brokered Notification Notification (WS-BrokeredNotification). IBM, Globus, Akamai et al. ftp://www6.software.ibm.com/software/developer/library/ws-notification/WS-BrokeredN.pdf
26. Web Services Topics (WS-Topics). IBM, Globus, Akamai et al. ftp://www6.software.ibm.com/software/developer/library/ws-notification/WS-Topics.pdf
27. WS-Resource Properties. IBM, Globus, USC et al. http://www-106.ibm.com/developerworks/library/ws-resource/ws-resourceproperties.pdf
28. Web Services Resource Lifetime. IBM, Globus, USC et al. http://www-106.ibm.com/developerworks/library/ws-resource/ws-resourcelifetime.pdf
29. I. Foster (ed), J. Frey (ed), S. Graham (ed), S. Tuecke (ed), K. Czajkowski, D. Ferguson, F. Leymann, M. Nally, I. Sedukhin, D. Snelling, T. Storey, W. Vambenepe, S. Weerawarana, “Modeling Stateful Resources with Web Services v. 1.1.” March 5, 2004. Available from http://www-106.ibm.com/developerworks/library/ws-resource/ws-modelingresources.pdf.
30. Web Services Management. Microsoft, Intel et al. http://msdn.microsoft.com/library/en-us/dnglobspec/html/ws-management.pdf

Table 1: Comparison of WS-Notification and WS-Eventing

	WS-Notification	**WS-Eventing**
Related Specifications	SOAP, WS-Addressing, WS-BaseNotification, WS-Brokered Notification, WS-Topics, WS-Resource Properties and WS-ResourceLifetime	SOAP, WS-Addressing
Support for loosely coupled notifications. (Producers need not know consumers)	Yes. The intermediary called Notification Broker and the exchanges that need to be supported are defined in the WS-Brokered Notification specification.	No.
Delivery modes supported	Push	Push Batched, Pull, & Trap (udp) defined in WS-Management
Delegated Management of subscriptions	Yes. Through the subscription manager interface.	Yes. Through the subscription manager interface.
Support for replay like features	One can get last message to a topic. A sink can also retrieve message issued between the **pausing** and **resumption** of a subscription.	No. However WS-Management introduces notion of resume/pause subscriptions.
Subscription operations	**Subscribe**, **Pause** and **Resume**. (There is **NO** exchange to *unsubscribe*).	Subscribe, Renew, Unsubscribe and Subscription End.
Subscription termination notification	**NO**	Yes. There is a SubscriptionEnd notification that is sent out by the source to the sink anytime the subscription ends (either an unsubscribe or termination)
Support for filters on to narrow notifications	YES	YES
Subscription lifetimes	Defined using the WS-Resource Lifetime specification.	Contained within the Subscribe and Renew exchanges.
Notification filters and topic expressions supported	Topic Expressions supported: QName, "/" separated Strings, and XPath path expressions.	Filter supported is XPath.
Hierarchical topics and Wildcards support	Yes. Supports * and // wildcards for selection of topic descendants in a topic tree.	No.
Topic space management	Defined using WS-Topics. The topic space will also support exchanges as defined by the WS-ResourceProperties specification.	No formal recommendation regarding topic management.
Advertisement of supported topics	Yes. The NotificationProducer interface allows inspection of available topics.	No.
On demand publishing	YES. This is supported through the WS-Brokered Notification specification. This allows a publisher to publish ONLY if there is a consumer interested in receipt of notifications.	NA. A source always keeps information regarding the sinks, so on-demand publishing is the default mode.
Notification messages	Provides support for both a Notify message as well as "raw" application specific message,	Does not define any special Notification message type.
Retrieve information about Topics from producer	Yes. Also indicates if the set of topics is going to be dynamic.	NO.
Retrieve info about topic expression dialects	Yes.	NO
Suggested Security	WS-Security and assorted specifications.	WS-Security & assorted specifications.