

A Flexible and Efficient Approach to Reconcile Different Web Services-based Event Notification Specifications

Yi Huang and Dennis Gannon

Department of Computer Science, Indiana University - Bloomington
{yihuan, gannon}@cs.indiana.edu

ABSTRACT:

Web services-based event notification is an emerging technology that combines the asynchronous communication feature of event notification mechanisms and the interoperability feature of Web services technologies. Web services-based event notification systems are important components for service-oriented computing. *WS-Eventing* and *WS-Notification* are two major competing specifications on these systems. Our *WS-Messenger* project is designed to create a Web services-based event notification system for efficient, scalable and interoperable message delivery on the Internet-scale. This paper first identifies some unique challenges in Web services-based event notification systems by comparing them with previous event notification systems. One of these challenges is how to mediate the competing specifications so that notification systems from different vendors can interoperate with each other. A successful mediation requires balancing flexibility and efficiency. This paper presents the mediation approach in *WS-Messenger* that maximizes both efficiency and flexibility based on a flexible mediation model (NPC model). The performance results show our mediation approach accounts for only less than 0.4% of overall message processing time.

KEY WORDS:

Publish/Subscribe, Web services, Event Notifications, Mediation, WS-Eventing, WS-Notification

Introduction

Event notification mechanisms are widely used in distributed applications to achieve asynchronous communications. One application can notify other applications about its running status, computing results, errors or exceptions by sending events. In Publish/Subscribe event notification systems, event consumer applications (subscribers) can express their interest in receiving certain notification messages published by event producer applications (publishers) using a subscribe operation. In large-scale publish/subscribe systems, notification brokers are important components for decoupling the publishers and the subscribers and improving system scalability (Eugster, et al., 2003). Event-driven architecture (EDA) is referred as a set of design guidelines for loosely-coupled system architectures based on event notifications.

Web services technologies are promising technologies to integrate distributed heterogeneous applications that are developed using different programming languages and run on different platforms. Interoperability among web services is achieved by exchanging messages in well-defined formats following a set of specifications, such as XML (W3C, 2000), SOAP (W3C, 2003) and WSDL (W3C, 2001). Comparing with previous distributed technologies, such as CORBA (Gokhale, et al., 2002), Web services technologies are easier to use and better suited for integrating applications across wide area networks (WAN). Web services technologies fits well with the emerging Service-Oriented Architecture (SOA) which aims at flexible application constructions using a set of autonomous, reusable and adaptive services. EDA is getting more and

more attention in the context of SOA because its loosely-coupled nature matches well with the goals of SOA.

Web services-based (WS-based) event notification systems serve as a perfect connection point between EDA and SOA. They combine features of both the event notification mechanisms and the Web services technologies. In such systems, notification messages are XML messages. Operations, such as subscribe, unsubscribe and message delivery, are accomplished using Web services technologies. WS-based event notification systems play an important role in SOA. Unlike most of the current event notification systems, WS-based event notification systems do not require all entities in the publish/subscribe systems to use the implementation from the same vendor. This is a great advantage for the integration of applications across enterprise boundaries as it is very hard to mandate applications in different organizations to use the same event notification implementation or library.

There are several unique challenges in WS-based event notification systems that haven't been addressed in previous event notification research, including interoperability, performance in XML processing and XML filtering and security for message dissemination in wide area network. The WS-Messenger project is motivated by these unique challenges. It is a notification broker that we have designed and implemented to achieve efficient, scalable and interoperable WS-based event notification. In this paper, we will concentrate on how WS-Messenger addresses the interoperability problem among competing specifications for WS-based event notification systems.

A mediation approach is adapted in WS-Messenger to address this interoperability challenge. Besides the typical features of a notification broker in a publish/subscribe system, such as decoupling heterogenous clients, subscription managements and scalable message delivery, WS-Messenger implements the Web services interfaces of both WS-Notification and WS-Eventing specifications and can reconcile the differences between them. Interoperability between these two specifications is achieved through the transformation of message formats to make sure the message "on the wire" can be understood by message receivers. The research challenge is how to balance the flexibility and efficiency in the mediation. Our mediation approach is based on a flexible and scalable mediation model called "Normalization-Processing-Customization" (NPC) model. The performance evaluation shows this approach is also very efficient.

In this paper, we will first present unique features and challenges in WS-based event notification systems and the architecture of WS-Messenger. We will then present the efficiency vs. flexibility problem in Web services mediations and our solutions in WS-Messenger to maximize both efficiency and flexibility when reconciling competing specifications. WS-Messenger has been applied in real-world service-oriented computing projects. In this paper, we will also describe its application to the workflow orchestration in LEAD project.

WS-Messenger has been introduced in several previous publications (Huang & Gannon, 2006a; Huang & Gannon, 2006b; Huang, et al., 2006). For completeness of the presentation, here we have included an updated version of some of that material.

Web Services-based Event Notifications

The Publish/Subscribe paradigm (Eugster, et al., 2003) is a practical pattern for event dissemination in distributed systems. In this paradigm, a *subscriber* subscribes to specific kinds of events to be sent to one or more *event consumers* (also called *event sinks* or *notification*

consumers). An *event producer* (also called *event source*, *notification producer* or *publisher*) publishes events. Events are delivered to the event consumers based on the subscriptions. One or more notification brokers can be added to publish/subscribe systems to decouple event producers and event consumers, as shown in Fig.1. A notification broker takes the roles of both an event consumer and an event producer. It acts as an event producer to an event consumer/subscriber and acts as an event consumer to an event producer. By communicating through notification brokers, event consumers and event producer do not need to know each other. Event consumers do not need to be online when event producer publish events. Event producers do not need to manage subscriptions. Notification brokers decouple event producers and event consumers and add flexibility and scalability to event notification systems.

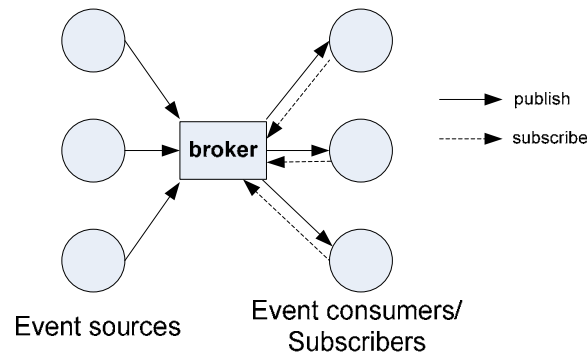


Fig. 1: Roles of notification brokers in Publish/Subscribe systems

Web services technologies are based on a set of open and widely adopted Internet standards. They are designed for integrating heterogeneous applications over the Internet. Features of Web services technologies include platform-independent, programming language-independent and transport-independent. Web service messages follow the XML standard (W3C, 2000) which defines a flexible and easy-to-extend data format and is supported on virtually every platform (Gisolfi, 2001). XML messages are encapsulated in SOAP (W3C, 2003) envelopes. Web Service Description Language (WSDL) (W3C, 2001) defines valid XML document structures for message exchanges to enable the interoperability feature of Web services. Web service messages can be transported using various transport mechanisms, such as HTTP, TCP and SMTP. SOAP messages are composable. Features like security, reliability and transaction can be added to the messages using respective specifications, such as WS-Security (Nadalin, et al., 2004).

WS-based event notification systems utilize the Web services technologies to deliver event notifications and manage subscriptions. A *subscriber* sends a SOAP-formatted subscription request message to an *event producer* Web service, requesting the delivery of certain kinds of XML-based notification messages to one or more *event consumer* Web services. When events are created in one service, other services can receive notification messages in the SOAP message format. The locations of the *event consumer* Web services are specified using the WS-Addressing specification (Box, Curbera, et al., 2004). Notification messages can be transported through intermediary and can use various transportation mechanisms supported by Web services.

Various Specifications on Web services-based Event Notification Systems

In order to achieve interoperability among different vendors of WS-based notifications, vendors need to agree on a specification that defines the message formats and Web service interfaces for notification delivery and for subscriptions creation and management. Ideally, we wish there was only one specification agreed by all vendors. However, three similar specifications have been

proposed in this area: WS-Events (Catania, et al., 2003), WS-Eventing (Box, Cabrera, et al., 2004) and WS-Notification (Chappell & Liu, 2004; Graham & Murray, 2004; Vambenepe, 2004). WS-Events was the earliest one. It was created by HP in mid-2003. HP joined others and proposed WS-Notification, which replaced WS-Events. Currently, WS-Eventing and WS-Notification are two major competing specifications in this area. Recently, another proposed specification, WS-EventNotification specification, is discussed in a white paper (Cline, et al., 2006) that aims at integrate functions from both specifications and co-exist with them. It is not available yet.

WS-Eventing

The Web Services Eventing (WS-Eventing) specification (Box, Cabrera, et al., 2004) has two released versions, the 1/2004 version and the 8/2004 version. The first version was released in January 7, 2004, led by Microsoft. The second version was released in August 2004. It received broader vendor supports. IBM, Sun and CA joined the supporters for this specification. WS-Eventing was submitted in March 2006 to the standard organization, World Wide Web Consortium (W3C) ("W3C website", 2006), for standardization.

WS-Notification

The WS-Notification family was released together with the WS-Resource framework (WSRF). They are both developed by the Grid computing community. WSRF is a framework designed to manage Grid computing resources through Web services. The Web Services Notification (WS-Notification) specification was first released in January 20, 2004, led by IBM and Globus Alliance. It was refactored into a family of three individual specifications and a whitepaper in March 2004. These three specifications are: the Web Services Base Notification (WS-BaseNotification) specification (Graham & Murray, 2004), the Web Services Brokered Notification (WS-BrokeredNotification) specification (Chappell & Liu, 2004) and the Web Services Topics (WS-Topics) specification (Vambenepe, 2004). WS-BaseNotification defines basic interactions between notification producers and notification consumers. WS-BrokeredNotification defines interfaces for notification brokers. WS-Topic defines a hierarchical topic space. The WS-Notification family was submitted to the standard organization, OASIS ("OASIS organization", 2006), in April 2004 and was approved as an OASIS standard in October 2006.

WS-BaseNotification is very similar to WS-Eventing in architectures and functions. It has 3 major versions so far, 1.0, 1.2 and 1.3. Version 1.0 was released in March, 2004 by refactoring the first WS-Notification specification. Version 1.2 is the version submitted to OASIS, it is very similar to version 1.0. Version 1.3 is the version that has been approved as OASIS standard. It has some major improvements over version 1.2, including making WSRF optional and adding "pull" delivery support.

Comparison of Web services-based Event Notification specifications with previous Event Notification Specifications

As first described in (Huang & Gannon, 2006a), prior to WS-based event notification specifications, several other specifications have attempted to define a standard way of sending event notifications in distributed systems. WS-Eventing and WS-Notification specifications have many similarities to previous specifications. However, they address some unique issues that are not covered by previous specifications, such as XML format and XPath filter. In this part, we will present a review of major event notification specifications prior to the announcements of these

two WS-based event notification specifications and compare them with these two new specifications. The comparisons are based on the latest released versions of each specification.

CORBA Event service specification and Notification service specification

The Common Object Request Broker Architecture (CORBA) (OMG, 2004a) specification was developed by the Object Management Group (OMG), which is a consortium of over 700 member companies. CORBA is designed to be programming language-, operating system-, and vendor-independent. It defines common interfaces for different programming languages and allows different programs to communicate through Object Request Broker (ORB). CORBA uses General Inter-ORB Protocol (GIOP) for intranet communications and Internet Inter-ORB Protocol (IIOP) for the Internet communications. IIOP maps requests and replies of GIOP to the Internet's TCP layer in each computer. The message payload is in a binary format known as Common Data Representation (CDR).

CORBA defines Event services and Notification services to support interactions among CORBA objects. The specifications of these services define both the interfaces and the underlying infrastructures for CORBA notification systems. They are based on the publish/subscribe paradigm. Event suppliers and event consumers communicate with each other through event channels.

The CORBA Event Service specification (OMG, 2004b) was first introduced in March, 1995. It is intended to decouple clients and servers so that the servers do not have to keep a list of client callback registrations. According to this specification, an event supplier publishes events to a CORBA event service channel. Event consumers get events from the channel. Both “Push” and “Pull” modes are supported. CORBA event service enables asynchronous communications between suppliers and consumers. They are location transparent. Although CORBA Event Service defines a simple mechanism for event propagation, it has noticeable drawbacks. It does not address event filtering and Quality of Service (QoS). A consumer receives all events on a channel.

The CORBA Notification Service specification (OMG, 2004c) is an enhancement to the CORBA Event Service specification. It adds supports for event filtering and Quality of Service (QoS). The CORBA notification service specification introduced “Structured Events” which provides a well-defined data structure to map a generic event to a well structured event. The structured event is useful for efficient filtering. The event filtering in the notification service is based on a filter object. The filter language is an expression whose syntax follows the Extended Trader Constraint Language. CORBA Notification specification defines 13 QoS properties that must be understood by all implementations even though they are not required to be implemented. Other QoS properties can be extended.

Although CORBA has implementations on different platforms and in different programming languages, the reality is that any solution built on CORBA will depend on a single-vendor's implementation. Vendors like to deploy their products on every nodes and using their own middleware to integrate these nodes. They do not have the incentive to achieve interoperability with others. Implementations from different vendors cannot interoperate well, especially when it comes to security, transaction management and performance optimization (Gisolfi, 2001). CORBA can only achieve interoperability on the intranet scale, where the distributed environment is well managed and has predictable latencies.

JMS Specification

Java Message service (JMS) (Hapner, et al., 2002) is a specification created by Sun Microsystems. It describes the APIs for Java programs to create, send, receive and read an enterprise messaging system's message. JMS is widely used in J2EE enterprise applications.

JMS defines two messaging styles: the point-to-point message queue style and the publish/subscribe style. It also defines five message types: `textMessage`, `byteMessage`, `mapMessage`, `streamMessage` and `objectMessage`.

JMS messages have well defined structure in the header field for efficient filtering. Subscribers can express their interests in JMS messages using queue names, topic names or message selectors. A message selector defines selecting criteria based on the header fields using an expression whose syntax is a subset of the SQL92 conditional expression. QoS criteria defined in JMS are priority, persistence, durability, transaction and message order.

The limitation of the JMS specification is that it only works on Java platforms.

Comparison with Web services-based event notification specifications

Table 1 compares the three previous specifications we discussed above with major WS-based event notification specifications. From this comparison, we can see how the event notification specifications have been evolving over time. These changes also reflect changes in the requirements on event notification systems over times. Several interesting observations on WS-based event notifications are found in this comparison:

- (1) The event delivery scope is extended to the Internet scale. The message delivery mechanism is moving towards transport-independent. WS-based event notification systems best suit for delivering event notifications among heterogeneous systems across the Internet where interoperability is a major concern. Although some of previous event notification systems enable Internet-scale delivery, they need well-managed environment. Event sources and event consumers usually need to be under the control of the same administrator in previous systems.
- (2) XML-based SOAP messages are used as message payloads.
- (3) The message filtering mechanism is moving from the simple subject-based topic filtering to the content-based XPath filtering.
- (4) The criteria of Quality of Service (QoS), such as reliability, transaction, are no longer defined in the specifications. Instead, they depend on composing with other WS-* specifications, such as WS-Reliability, WS-Transaction.
- (5) The soft-state management (timeout) of subscription terminations is used. The connections to event consumers do not always keep alive. Also, a subscriber and an event consumer can be two different entities. They are usually combined together in previous systems.
- (6) Interoperability concerns are shifted from the fine-grained API level to the more coarse-grained service interfaces and SOAP messages level. Event producers, event consumers and brokers can interoperate with each other using SOAP messages with standard formats. They do not need to use implementations from the same vendor.

j		CORBA Event Service	CORBA Notification Service	JMS	WS-Notification	WS-Eventing
Overview	First Release	3/1995	6/1997	1998	1/20/2004	1/7/2004
	Latest Release	10/2/2004	10/11/2004	4/12/2002	10/1/2006	3/15/2006
	Creator(s)	OMG	OMG	Sun Microsystems	OASIS (HP, IBM, TIBCO edited)	W3C submitted (BEA, CA, IBM, Microsoft, TIBCO edited)
Delivery	Message transport	RPC	RPC	RPC	Transport independent	Transport independent
	Intermediary	EventChannel object	EventChannel object	Message Queue, Pub/Sub broker	directly or through broker	directly or through broker
	Delivery Mode	Push, pull & both	Push, pull & both	Pull, Push	Push, Pull	Push by default, Can use Pull or other modes
Message	Message Structure	Generic (Anys), Typed	Generic (Anys), Typed, Structured, sequences of structured	TextMessage, ByteMessage, MapMessage, StreamMessage, ObjectMessage	SOAP (with Raw XML data or wrapped messages)	SOAP (with Raw XML data only). Can use wrapped mode.
	Filter	No	Channel, Filter Object.	Queue/topic name, message selector on header fields	Hierarchy Topic tree; Content Selector. Producer properties.	A "Filter" element for any filter. At most 1 filter.
	Filter language	No	Extended Trader Constraint Language	a subset of the SQL92 conditional expression syntax	Any expression (xsd:any) that evaluates to a Boolean. e.g. XPath	Default XPath. Can use any expression (xsd:any) that evaluates to a Boolean.
Management	QoS criteria	Not defined	Defined 13 QoS properties, can be extended to others	Priority; persistence; durable; transaction; message order	Depends on composition with other WS* specification	Depends on composition with other WS* specification
	Subscription Timeout	No	No	No	Absolute Time or duration	Absolute time or duration
	Demand-based	No	Defined	No	Defined	No
	Management operations	connect_*, obtain_(typed)_push/pull_supplier/consumer	connect_*, obtain_notification_push_supplier/consumer, suspend/resume_connection., get/set/validate_qos, add/remove/get/getAll/removeAll_filter, obtain_subscription/offered_types	createSubscriber, createDurableSubscriber, unsubscribe	Subscribe, Renew, unsubscribe, Pause/resume subscription, get/getMultiple/set/query ResourceProperties, TerminationNotification, Destroy, SetTerminationTime	Subscribe, Renew, GetStatus, Unsubscribe, SubscriptionEnd

Table 1: Comparison among specifications on event notifications

Unique Challenges in Web Services based Publish/Subscribe Systems

The changes in WS-based event notification systems bring are some unique challenges that have not been addressed in traditional publish/subscribe systems. In this Section, we will discuss some of the major challenges to Web services based publish/subscribe systems.

Mediation among different Web services based publish/subscribe specifications

WS-Notification and WS-Eventing are two major specifications for WS-based event notification systems. It is hard to connect an event source and an event consumer that are implemented either in different specifications or in different versions of the same specification. It is quite likely that

neither side wants to change the current specification implementation if they belong to different organizations. A mediation service is needed to connect them. A successful mediation requires balancing flexibility and efficiency. This challenge is the focus of this paper. We will explain the problem in more details and present our proposed solutions in WS-Messenger in the following sections.

Interoperability among different implementations of the same specification

Interoperability in the open Internet environment depends on service specifications. However, it is still not easy to achieve truly interoperable Web services even if they follow the same specification because Web services technology is still not mature enough and the specifications are not precise enough for defining the “messages on the wire” that are interoperable. Optional elements defined in the specification can also cause incompatibility problem when connecting two different implementations.

Mediation among different transport mechanisms

Web services are transport-agnostic. The services can communicate using different transport protocols, such as HTTP 1.0, HTTP 1.1, TCP, SMTP, etc. It is quite likely that an event source and an event consumer support two different transport protocols and cannot communicate with each other, in which case mediation is needed to transform the transport protocols.

Scalable and efficient XML processing

In traditional Publish/Subscribe messaging systems, the performance bottleneck is usually the message filtering and the destination matching. In Web services based publish/subscribe systems, the bottleneck is most likely to be the compute-intensive XML processing. In other words, the bottleneck is changed from “CPU bound” (internal processing) to “IO bound” (sending and receiving messages). More scalable XML processing capability is needed to improve the performance of Web services based publish/subscribe systems.

Efficient XML-based content filtering

In traditional publish/subscribe messaging systems, most messages are not in XML format. In Web services based publish/subscribe systems, however, most messages delivered are XML-based SOAP messages. There is increased interest in filtering the XML messages based on XML structure and content. For example, both WS-Eventing and WS-Notification define XPath (Clark & DeRose, 2006) filtering for message selection. WS-Eventing specification uses it as the default messaging filtering. How to create efficient XML-based message filters for the Web services based publish/subscribe systems is a research challenge. Some research has been conducted in this field to create an efficient XPath filtering for XML message stream, such as YFilter (Diao & Franklin, 2003).

Efficient XML-based content filtering on the Internet-scale is an even harder problem. When there are multiple notification brokers distributed across the Internet, the additional challenge is how to make them work together to achieve efficient message filtering and delivery on the Internet-scale.

Security

Traditional publish/subscribe systems usually operate in a controlled environment. They can be protected by firewalls from outside attacks. When the scope of message delivery is extended to the Internet and among un-trusted service entities, security is an important concern. Authentication, authorization, integrity, confidentiality and non-repudiation all need to be designed at the Internet scale. Web services based security also brings tremendous performance overhead (Shirasuna, et al., 2004) and makes interoperability even harder.

WS-Messenger Overview

As originally introduced in (Huang, et al., 2006), the research efforts for WS-Messenger focus on addressing the aforementioned unique challenges in Web services based publish/subscribe systems. In this section, we will discuss the architecture of WS-Messenger and explain the functionality of each layer in the architecture.

Figure 2 shows the architecture of WS-Messenger. It has four layers: a Web service I/O layer, a mediation layer, an application logic layer and a messaging system adapter layer.

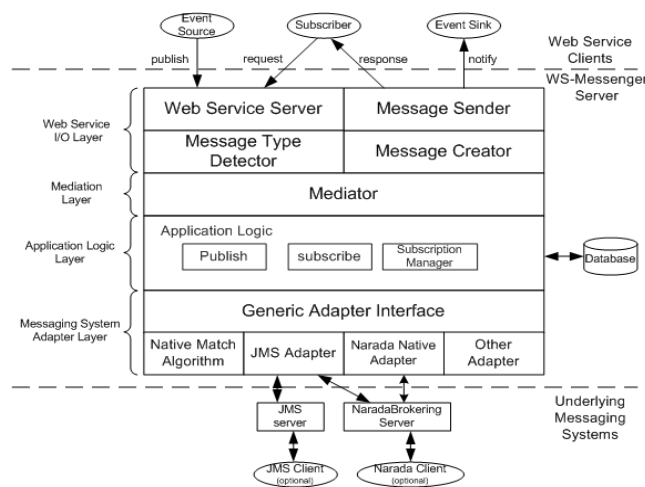


Fig. 2: Architecture of WS-Messenger

The “*Web service I/O layer*” interacts with both WS-Eventing clients and WS-Notification clients. Different types of transport mechanism can be used in the Web service I/O layer. Currently, we have implemented support for the HTTP protocol from two different implementations, HTTP implementation in the XSUL Web services toolkit (“XSUL web site”, 2005) and Apache Tomcat Web server (“Apache Tomcat Project”). They both have advantages and disadvantages. XSUL HTTP implementation is light-weighted and runs as regular java process. It is simple for debug tracing and embedding in other Java applications. Tomcat is more scalable and support latest HTTP 1.1 protocol. We also implemented support for SOAP.TCP protocol. SOAP.TCP protocol is used in the Web Service Enhancement (WSE) package for Microsoft .NET platform. It sends and receives DIME (Nielsen, et al., 2002) framed messages using TCP transport.

The “*message type detector*” inspects each received message to decide which specification the message follows. In our implementation, this is decided by checking the *wsa:action* element in the SOAP header, which is an element defined in the WS-Addressing (Box, Curbera, et al., 2004) specification and it is required to be specified in the SOAP header in both specifications.

The mediator in the “*mediation layer*” is the translator between the WS-Eventing message format and the WS-Notification message format based on the message type that is already determined by the “Message Type Detector”. It also translates the request messages, such as a *subscribe* request, to a java bean object that is used by the “application logic layer”. Mediation between WS-Eventing and WS-Notification is needed when handling the received notification messages and sending them to different kinds of event consumers. We will discuss the mediation approach we used in WS-Messenger in more detail in the following sections.

The “*application logic layer*” handles the business logic of subscribing, publishing and subscription management. A database is currently used in the application logic layer to save the subscription entries. If the WS-Messenger server accidentally crashes, previous subscriptions can be retrieved from the database to restore the WS-Messenger server to the state before the crash. The database is also designed to temporarily save undeliverable messages to support reliable message delivery.

The “*messaging system adapter layer*” is used to support various underlying publish/subscribe messaging systems. It is designed to leverage existing messaging systems. This layer contains a generic adapter interface to the “application logic” layer and individual adapters for different messaging systems. Currently two adapters are implemented: a JMS adapter and a NaradaBrokering (Fox & Pallickara, 2003) native interface adapter. The JMS adapter can be used to integrate with most Java-based Publish/Subscribe messaging systems, e.g. openJMS (OpenJMS), activeMQ (ActiveMQ). Other adapters can be created to accommodate other non-standard interfaces. For example, the NaradaBrokering native adapter can take advantage of the content-based subscription option offered by NaradaBrokering system.

By wrapping up these underlying messaging systems, WS-Messenger creates interoperable Web services based publish/subscribe systems based on existing messaging systems. It can take advantage of the features offered by the existing well-developed messaging system. Different messaging systems have different features. Some may emphasize on the scalability and reliability; while others may emphasize on fine-grained message filtering. By choosing different messaging systems, WS-Messenger can be applied to different environments to meet various requirements.

This “wrapping-up” approach also enable WS-Messenger to act as a communication bridge between traditional publish/subscribe clients and WS-based publish/subscribe clients. For example, if WS-Messenger uses a JMS broker as the underlying message system, a JMS client can publish a notification message to the JMS broker. The JMS broker then forwards the message to WS-Messenger through a JMS adapter if there are interested WS-based consumers for this message on the topic. WS-Messenger can then deliver the message to those WS-based consumers. Fig 3 shows this scenario.

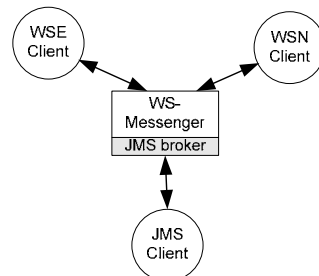


Fig. 3: WS-Messenger acts as a communication bridge between JMS clients and Web services clients

The limitation of this “wrapping-up” approach is that the available subscription options (dialects) of WS-Messenger depend on underlying messaging systems. If the underlying messaging system

Interoperability Problems in WS-based Event Notification Systems

Interoperability among distributed systems has been a challenge for years. Although Web services technology make it easier to integrate heterogonous systems, achieving interoperable WS-based event notifications across organization boundaries is still a challenge. Interoperability among applications from different vendors requires agreements on communication protocols, SOAP message formats and semantics in XML message contents. Out of these three factors, communication protocol is the easiest to get agreement since the HTTP protocol is widely used for communication among web services. The semantics in XML message contents can be agreed by using application level XML schemas. Notification brokers do not understand the content of published messages.

We concentrate our work on addressing the interoperability problem in different message formats defined in competing specifications and assume the involved parties have agreed on communication protocols, e.g., HTTP 1.0, and XML schema for notification messages. It is much harder to get agreement on event notification specifications used in applications from different parties since it depends on vendors of their notification infrastructure.

As discussed earlier, there is no universally supported specification for the Web services interfaces of event notification systems. Different specifications have been proposed in this area. Specifically, *WS-Notification (WSN)* (OASIS, 2004) and *WS-Eventing (WSE)* (Box, Cabrera, et al., 2004) are two major initiatives. They are incompatible with each other and both have their own supporters and implementations. For example, in the Grid computing (Foster, et al., 2003) community, the WS-BaseNotification specification (version 1.2) is implemented in the Globus toolkit 4 (Foster, et al., 2002) which is the most popular toolkit for building Grid systems. The WS-Eventing specification is also used for building several Grid systems (Gannon, et al., 2005; Humphrey, et al., 2005) because it is simpler and supported by more vendors. Since these two specifications are incompatible with each other, it is hard to make the event notification systems in different Grid computing systems interoperable with each other even though they all use Web services technologies. Even when the proposed WS-EventNotification specification comes out in the future that merges these two specifications, we expect that specification difference will still be a problem for a long time due to different vendor supports for existing specifications, backward compatibility and version updates in the new specification.

In many cases, it is not feasible to change the specification implementations used in existing systems, especially if the change involves two or more organizations. First, specification change may break other existing systems that depend on the current specification to interoperate. Second, changing specification also involves various costs, such as the development cost and the license cost. Third, even for the same specification, implementations of different versions of that specification are still incompatible with each other. How to keep existing WS-based event notification systems unchanged and achieve interoperability among various implementations of competing specifications is the problem that we will address in this paper. A mediation approach is adapted in WS-Messenger to reconcile specification differences transparently to the clients.

Flexibility vs. Efficiency Problem in Mediation

Competing Web services is defined as two or more Web services that address the same application area but have different Web services interfaces for the same function. A mediation approach is needed to reconcile their differences so that they can interoperate with each other. *Flexibility* of a mediation approach is defined as how easy it is to add other Web services to an existing set of mediated Web services. The mediation approach should be as flexible as possible

for adding new mediation scenario. *Efficiency* of a mediation approach is defined as how fast a mediation process can be done. The mediation overhead should be as small as possible. Adding flexibility to mediation usually requires adding intermediary processing steps, which adds more complexity and overhead in the mediation process than direct transformation. Achieving both high efficiency and high flexibility in mediation is a challenge.

NPC Mediation Model

We propose a Normalization-Processing-Customization model (NPC model) for mediation among competing Web services. The premises of using NPC model are that the Web services have *similar* functionalities but *different* message formats. Two Web services are considered having *similar functionalities* if their core functionalities can be achieved using the same or almost the same business logic. It is the foundation for mediation. On the other hand, if two Web services have the same message format, no mediation is necessary.

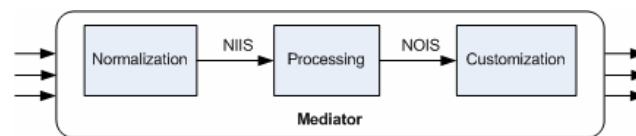


Fig. 6: NPC mediation model

Fig. 6 shows the NPC model. Three major processes are involved in this model: normalization, processing and customization. Incoming SOAP messages following different formats/specifications are transformed to the normalized incoming information set (NIIS) through a normalization process. NIIS is then processed by a business logic program and generates the normalized outgoing information set (NOIS). The processing unit contains business logics of all core functionalities defined in competing Web services. The customization process transforms the NOIS to customized outgoing SOAP messages that follow the formats/specifications required by the message receivers. In the customization process, it is important to make sure that the messages going “on the wire” can be understood by the message receivers.

NIIS and NOIS are determined by analyzing mediation requirements. The incoming/outgoing SOAP messages in competing Web services have a great portion of overlap in the information set, even though these pieces of information may appear in different locations in the SOAP messages or in different namespaces. The **Normalized Incoming Information Set (NIIS)** contains the union set of all possible information in the incoming SOAP messages that are needed for processing the message. Depending on business logics in the processing units, some NIIS entries are required, some are optional. Optional entries can have no value (null). The **Normalized Outgoing Information Set (NOIS)** contains the union set of all possible information needed to create the outgoing messages. Each entry in a NOIS may have two statuses: either value available or value unavailable.

Entries (elements and attributes) in incoming/outgoing SOAP messages are defined by Web services interface definition languages, e.g. WSDL (W3C, 2001) or SSDL (Parastatidis, et al., 2006). When transforming between an entry in an incoming/outgoing SOAP message and an entry in NIIS/NOIS, there are three scenarios: (1) The entry is required in the SOAP message, (2) The entry is optional in the SOAP message, or (3) The entry is not defined in the SOAP message by a web services interface definition language. Optional values may or may not appear in SOAP messages.

Mediation actions are used in the normalization and customization processes to transform between entries in incoming/outgoing SOAP messages and entries in NIIS/NOIS. A *mediation rule* consists of a set of mediation actions. In general, there are four types of mediation actions in the normalization or customization process.

Transform: Transform the values directly between entries in NIIS/NOIS and entries in incoming/outgoing SOAP messages.

Add default value: Create a default value and use it in mediation. The default values need to be analyzed case by case since they involve business logic.

Ignore: Discard the values in the information set since the values are not needed.

No action: No mediation is needed.

From \ To		NIIS entry	
		Required	Optional
Incoming Message Entry	Required	<i>Transform</i>	<i>Transform</i>
	Optional value available	<i>Transform</i>	<i>Transform</i>
	Optional value Unavailable	<i>Add default value</i>	<i>No action</i>
	Not defined	<i>Add default value</i>	<i>No action</i>

Table 3: Mediation actions selection matrix in *normalization* processes.

From \ To		Outgoing Message Entry		
		Required	Optional	Not defined
NOIS Entry	Value Available	<i>Transform</i>	<i>Transform</i>	<i>Ignore</i>
	Value Unavailable	<i>Add default value</i>	<i>No action</i>	<i>No action</i>

Table 4: Mediation actions selection matrix in *customization* processes.

Table 3 and Table 4 summarize guidelines in selecting mediation actions to transform from an entry in an incoming message/NOIS to an entry in a NIIS/outgoing SOAP message in the normalization/customization process in different scenarios.

The NPC model is flexible and scalable. By creating a normalized data representation, we can easily adapt to future changes. NPC model eliminates pair-wise transformation rules among SOAP message formats. To mediate among N formats, only maximum 2N mediation rules are needed: N normalization rules and N customization rules. This approach reduces the maximum possible transformations from N² to 2N. Adding a new SOAP message format to an existing mediation set of size N only requires adding 2 more mediation rules, instead of 2N mediation rules.

Analysis of WS-Eventing and WS-BaseNotification

The major goals of both specifications are to publish and subscribe event notification messages using standard Web services interfaces. However, they are incompatible with each other. At the SOAP message level, the message formats of these two specifications are different.

In this section, we will analyze the architecture, functionalities and SOAP messages of the latest versions of the WS-BaseNotification specification (version 1.3, the OASIS standard version) and the WS-Eventing specification (3/2006 version, the W3C submission version).

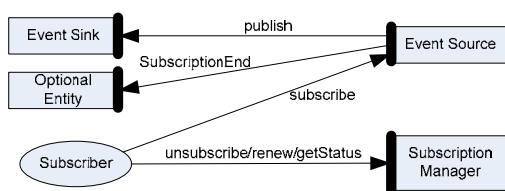


Fig. 7: WS-Eventing Architecture and Operations

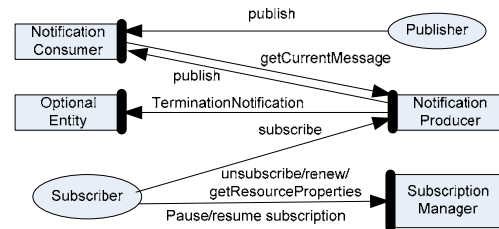


Fig. 8: WS-BaseNotification Architecture and Operations

Architecture Similarity

WS-Eventing and WS-BaseNotification have almost identical Publish/Subscribe architectures. They both define the *subscriber* and *subscription manager* entities. The *event sink* defined in WS-Eventing is comparable to the *notification consumer* defined in WS-BaseNotification. In both specifications, subscribers are separated from notification consumers so that notification consumers only need to handle received messages. They do not need to know broker locations and create subscriptions.

WS-Eventing does not separate the *publisher* from the *event source*. The Event source in WS-Eventing has the functions of both the *notification producer* and the *publisher* defined in WS-BaseNotification. The implication of this architecture in WS-Eventing is that the publisher needs to manage subscriptions and deliver notification to the consumers. This is a tightly coupled publish/subscribe model and not scalable. In order to add supports for intermediary message brokers to decouple event sources and event sinks, we follow the architecture of WS-BaseNotification specification in WS-Messenger, i.e. enabling an entity to just publish messages to a notification consumer without having to provide Web services interfaces to handle subscriptions. We expect WS-Eventing will separate these two entities (or make web services interfaces optional) in future versions. Fig. 7 and Fig. 8 show the entities defined in WS-Eventing and WS-BaseNotification and their interactions. The bold lines indicate Web services interfaces.

Functionalities Similarity

WS-BaseNotification and WS-Eventing have great overlap in their functionalities. Most of the functionalities defined in one specification can be mapped to another specification. Both specifications define key publish/subscribe related functions and are composable with other WS-* specifications to provide features like security, reliability and transaction management. For example, WS-Security (Nadalin, et al., 2004) can be used to add security supports and WS-Reliability (Iwasa, et al., 2004) can be used to achieve “once and only once” delivery.

Fig 9 shows the functionalities comparisons. WS-Eventing defines five Web service operations: *Subscribe*, *Renew*, *GetStatus*, *Unsubscribe* and *SubscriptionEnd*. The “*Subscribe*” message is used to create a subscription for an event sink. The “*Renew*”, “*GetStatus*” and “*Unsubscribe*” messages are sent from subscribers to subscription managers to manage existing subscriptions. The “*SubscriptionEnd*” message is generated when an event source terminates a subscription unexpectedly. It is sent to an address specified in the subscription request. If this address is not presented in the subscription request, this “*SubscriptionEnd*” message is not generated.

WS-BaseNotification has comparable operations for the above five operations. Although it does not define *GetStatus* and *SubscriptionEnd* operations, these operations can be achieved with the optional WS-ResourceFramework (WSRF) since WS-Notification can treat subscriptions as WS-

Resources in WSRF. Besides these five operations, WS-BaseNotification defines three more operations than WS-Eventing. It defines how to *pause* and *resume* a subscription and how to get the current message (*getCurrentMessage*). These three operations are optional operations.

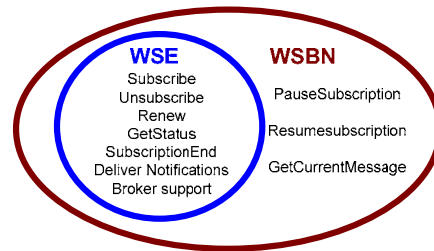


Fig.9. Functionalities Comparisons

Message Formats and Message Contents Differences

Web services specifications define SOAP message formats to encapsulate request and response messages. Since WS-Eventing and WS-Notifications are two different specifications, their message formats are different. When comparing the request and response SOAP messages in corresponding operations, such as the *subscribe* operation, many differences exist. The differences can be summarized in the following categories:

- (1) **Operation name difference:** Operation names for the same operation are different. For example, the “GetStatus” operation defined in WS-Eventing is similar to the “GetResourceProperty” operation (in WS-ResourceProperties) in WS-Notification.
- (2) **Element name difference:** The element names for the same content are different. These differences are not as apparent as the operation name differences. For example, the *subscriptionID* values in the subscription response messages are enclosed in the *ReferenceParameters* elements in WS-Eventing, while they are enclosed in the *ReferenceProperties* elements in WS-BaseNotification.
- (3) **Namespace difference:** The specifications’ namespaces and some other related namespaces used in the specifications are different, such as SOAP envelope namespaces and WS-Addressing namespaces.
- (4) **WS-Addressing version difference:** WS-Eventing uses the 2004/08 version of WS-Addressing, while WS-Notification uses the 2005/08 version.
- (5) **Message content difference:** The specifications define different content for certain XML elements in the SOAP messages. For example, the required values of the “*action*” elements in the WS-Addressing are different.
- (6) **SOAP message structure difference:** The XML message structures defined in both specifications are different. For example, the message payload of a wrapped notification message in the WS-Notification format is enclosed in a *NotificationMessage* element which is again enclosed in a *Notify* element. WS-Eventing notification messages do not need such structures.
- (7) **Content location difference:** The same XML elements may appear in different locations in SOAP messages. For example, a *topic* element is required in the SOAP body of a wrapped WS-Notification notification message, while it needs to be put in the SOAP header in a WS-Eventing notification if needed.

Flexible Mediation Approaches in WS-Messenger

A WS-Messenger broker can automatically convert SOAP message formats according to predefined mediation rules. Mediation rules are determined by analyzing the competing specifications, finding the normalized information sets for all specifications and mapping the

incoming/outgoing SOAP messages to the normalized information sets. The subscription request type for an event consumer determines the message format that the event consumer will receive. If a WS-Notification subscription request is received by the broker, it will send WS-Notification messages to the event consumer. Similarly, if a WS-Eventing subscription is received by the broker, the broker will produce WS-Eventing messages for that event consumer. An event source can publish messages in either format to WS-Messenger. It makes no difference to the event consumers. One assumption we have in our mediation approach is that the subscribers and the event consumers follow the same specification. This is a valid assumption for most cases since the subscribers need to know the location of the event consumers. They are tightly-coupled.

WS-Messenger discovers the specification a SOAP message follows by checking the “*action*” element in the WS-Addressing in the SOAP headers (see top of Fig. 11). Both WS-Eventing and WS-Notification specifications define the content of this element for operation messages. The WS-Eventing specification requires the action elements to start with WS-Eventing’s namespace, e.g. <http://schemas.xmlsoap.org/ws/2004/08/eventing/Subscribe>. The WS-Notification specification requires the action elements to start with the namespace of WS-Notification. After getting the content of the action element, simple string comparisons are used to determine the specification types.

Maximizing both the *efficiency* (lowering mediation overhead) and the *flexibility* (supporting future specification updates) is the concentration of our mediation efforts in WS-Messenger because of the nature of the messaging systems. First, as a notification broker, WS-Messenger is expected to handle high volume of messages. If mediation takes a large portion of message processing time, the system performance is deteriorated by the mediation. Second, both WS-Eventing and WS-Notification specifications are not finalized, we expect them to change in the future. Updating for future versions should be as simple as possible.

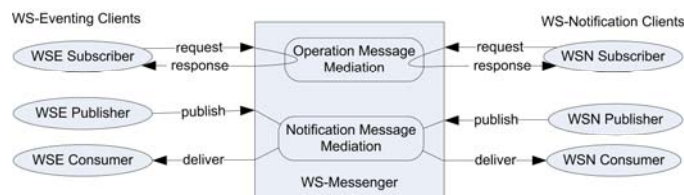


Fig. 10: Mediations in WS-Messenger

WS-Messenger adapts NPC model to achieve maximum flexibility. Future changes in specifications can be easily adapted and newer clients can interoperate with older clients which use older version of the specifications. This approach ensures a smooth and gradual update of the Web service clients in messaging systems. Two different types of mediations are needed in WS-Messenger: mediation for the request-response style operation messages and mediation for the one-way style notification messages (see Fig. 10). NPC model has been successfully applied to both mediation types.

Mediation of Operation Messages

Operation messages are messages for subscription management Web services, such as “subscribe” messages, “unsubscribe” messages, etc. These messages follow the request-response message exchange pattern. The WS-Messenger services acts as synchronous Web services. It receives a SOAP request message from a client (e.g. a subscriber), processes it and then sends a

SOAP response message back to the client. The incoming messages (requests) and outgoing messages (responses) always follow the same specification. The business processes for operation messages mainly deal with the subscription management.

The flowchart of processing and mediating operating messages is shown in the left branch in Fig. 11. Java bean objects are used as NIIS and NOIS in the normalization processes and processing processes. Since both specifications define the same core operations, e.g. subscribe and unsubscribe, the information set required for processing them are the same. For example, message filter, consumer endpoint location and expiration time are the key NIIS information to create a subscription for an event consumer. In addition to the information needed for processing the request, an additional field indicating the original specification used in the request messages is also included in NIIS and NOIS. In the customization process, the format of a response message is determined by which specification the request message uses.

Since the WS-Notification specification defines three more operations than the WS-Eventing specification (see Fig. 9), the processing business logic part of WS-Messenger needs additional data structures and codes for the internal management to accommodate these operations.

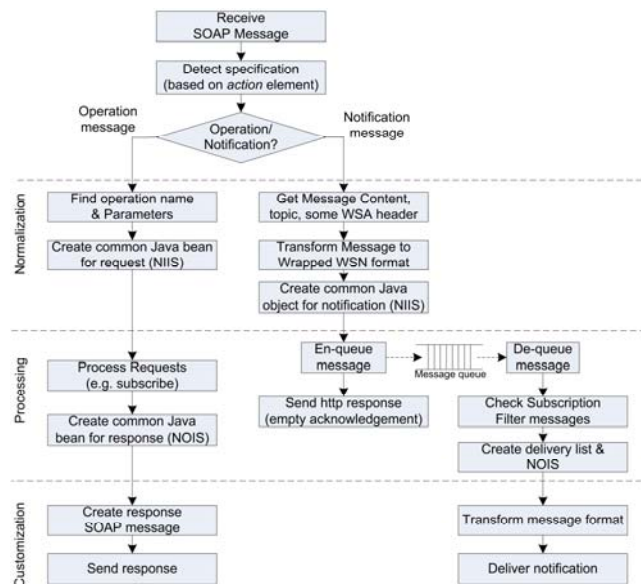


Fig. 11: The mediation processes in WS-Messenger based on the NPC model

Mediation of Notification Messages

Notification messages are published by publishers and received by event consumers. These messages follow the *one-way* message exchange pattern. In this case, WS-Messenger acts as a router relaying notification messages from the publishers to the event consumers. Unlike operation messages, one incoming notification message may create *multiple* outgoing notification messages following different specifications. The processing of notification messages mainly deal with message filtering and forwarding.

Event consumers need notification messages in their required formats. If an event consumer requires notifications following specification X while a publisher publishes notifications in the format of specification Y, mediations in notification brokers are used to fill in the gap between them. The notification messages are transformed by the notification brokers to make sure that the

event consumers get the notification messages following their respective specifications. This mediation process is transparent to both event consumers and event sources. Event consumers do not even know that the messages were originally published in different formats. Event sources do not need to know that there are event consumers expecting different message formats.

Mediation for notification messages involves transformation among different notification message formats. It is more complicated than mediation for operation messages. There are three possible notification message formats: WS-Eventing format, wrapped WS-Notification format and raw WS-Notification message format. WS-Eventing format is the simplest since WS-Eventing does not have any constraints on notification message formats. Any SOAP message could be a WS-Eventing notification message. Wrapped WS-Notification format has the most constraints. It is well defined in the WS-Notification specification, including what namespace is used for the SOAP message, what elements to put the notification in and how to specify the topic. Raw WS-Notification message has no constraint on the body of SOAP messages, but the action element in the WS-Addressing header uses the same action as wrapped WS-Notification message.

The flowchart of processing and mediating notification messages is shown in the right branch in Fig. 11. Java bean objects are used as NIIS and NOIS in the *normalization* and *processing* processes in the notification messages mediation. For example, the java bean for NIIS has fields on message content, messageId in WS-Addressing header, topic, etc. Wrapped WS-Notification format is used as the internal format in the java bean objects as the common notification format in NIIS and NOIS since it has the richest information set. If a WS-Eventing formatted notification message is received, it will be transformed to the wrapped WS-Notification format. If a wrapped WS-Notification formatted message is received, it will keep that message format.

Message queues are used in notification brokers to enable immediate replies to event producers when notification message are received without having to wait until the message delivery is complete. This can help decouple event producers and event consumers and improve scalability at the cost of some performance overhead in notification brokers.

Mediation Actions

Mediation action types in the NPC model are determined using mediation action selection matrix in Table 3 and Table 4. In this part, we will show some examples of mediation actions used in WS-Messenger.

“Transform” Actions

Creating adequate transformation rules is a critical part for a successful mediation. Transformation rules are determined by analyzing individual specifications and map respective message entries to normalized information sets. The mapping process requires thorough understanding of the semantics and message formats of both WSE and WSN specifications and related underlying specifications, such as WS-Addressing. This kind of expert knowledge is a common requirement for semantic mediation approaches.

Based on the analysis of the two specifications presented in the previous section, we created a set of transformation rules. Each rule includes the message specification version, message type, SOAP message entry XPath and normalized information set. Table 5 shows some sample transformation rules in the normalization process. Table 6 shows some sample transformation rules in the customization process. For example, the first row in table 5 shows that when

receiving a subscribe request message following WS-Eventing 1.0 specification, the content in the XPath of “/Envelope/body/Subscribe/Delivery/NotifyTo/Address” needs to be transformed to the “consumerAddress” property in the SubscriptionInfo.java.

Request Version	Message Type	Incoming Message Entry XPath	Normalized Incoming Information Set (NIIS)
WSE 1.0	subscribe request	/Envelope/body/Subscribe/Delivery/NotifyTo/Address	“consumerAddress” in SubscriptionInfo.java
WSE 1.0	notification	/Envelope/body	“notificationMessage” in ReceivedMessage.java
WSBN 1.2	subscribe request	/Envelope/body/SubscribeRequest/ConsumerReference/Address	“consumerAddress” in SubscriptionInfo.java
WSBN 1.2	wrapped notification	/Envelope/body/Notify/NotificationMessage/Message	“notificationMessage” in ReceivedMessage.java

Table 5: Sample transformation rules in *normalization* process

Response Version	Message Type	Normalized Outgoing Information Set (NOIS)	Outgoing Message Entry XPath
WSE 1.0	subscribe response	“subId” in SubscriptionManager.java	/Envelope/body/SubscribeResponse/SubscriptionManager/ReferenceParameters
WSE 1.0	notification	“textMessage” in OutgoingMessage.java	/Envelope/body
WSBN 1.2	subscribe response	“subId” in SubscriptionManager.java	/Envelope/body/SubscribeResponse/SubscriptionReference/ReferenceProperties
WSBN 1.2	wrapped notification	“textMessage” in OutgoingMessage.java	/Envelope/body/Notify/NotificationMessage/Message

Table 6: Sample transformation rules in *customization* process

“Adding Default value” Actions

The WS-Eventing (WSE) specification and the WS-Notification (WSN) specification do not always have one-to-one match. If an element is required by one specification, but it is not required by the other one, adding default value technique is usually needed. In general, WSE specification is simpler than WSN specification. When converting a WSE notification message to a WSN notification message, default values are added when necessary. If a default/required value can be found in the specification, that value is used in the transformation.

If a default/required value cannot be found in the specification, we need to create our own default value. Topic content mediation is such a case. A wrapped WSN notification message requires a topic destination, while a WSE notification message does not require a topic. How to create WSN notification messages when receiving WSE notification messages that do not have topic is a problem. WS-Messenger uses the “adding default value” technique to solve this problem.

Firstly, WS-Messenger tries to find the topic in the SOAP header of a WSE notification message. Although WSE specification does not require a topic in a notification message, it is possible to create an element in SOAP header to specify a topic if needed. For example, there is a non-normative example notification message in the WS-Eventing specification that has a topic in the header. By examining the SOAP header, WS-Messenger tries to find a topic element in the SOAP header. If there is one, it uses that as the topic element in the new WSN notification message.

Secondly, if WS-Messenger cannot find a topic in a WSE notification message, a reserved special topic, “*wseTopic*”, is used in the transformation to a wrapped WSN notification message. This addition is transparent to all the event consumers. (1) For WSE or WSN consumers who are

interested in specific topics, they will not receive the message since this reserved topic does not match to those topics. (2) For WSE or WSN consumers who showed interest in all messages by subscribing to all the topics using a wildcard subscription, they will not miss the message and can receive it in the correct format. (3) For WSE consumers who subscribed using filtering criteria other than topic, like XPath (Clark & DeRose, 2006) filtering, they will receive the message if it matches the filtering criteria since WS-Messenger can detect that no topics are specified in the subscriptions and subscribe automatically to this special topic for them.

“Ignore” Actions

WS-Messenger does not explicitly handle “Ignore” mediation actions. It just does not fetch the ignored values in NOIS in customization processes. These values are automatically ignored.

Client APIs for both specifications

WS-Messenger provides a client library that can be used by any java application to create subscriptions, manage subscriptions, send and receive event notifications following either WS-Eventing or WS-Notification. Individual API class for each of these two specifications is available. They both implement the same interface: *WsmgClientAPI*. (See Fig. 12). Depending on whether the user wants to use WS-Eventing or WS-Notification, the user use either *WseClientAPI* object or *WsntClientAPI* object in the application code. For example, *WSEClient publisher=new WseClientAPI()* creates a publisher following the WS-Eventing specification.

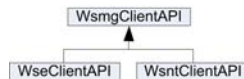


Fig. 12: Client APIs that support both WS-Notification and WS-Eventing

Calling *publisher.publish(consumer, producer, topic, message)* can publish the message to the consumer or notification broker.

To create a notification consumer, a class that implements a *NotificationHandler* interface is needed. The interface has only one method: *handleNotification(String soapMessage)*. The user can put the business logic to handle the received messages in this method. Then the user needs to call *startConsumerService(port, handler)* to start a listener at the specified port and use the specified handler to handle received notification messages.

Achieving Efficient Mediation in WS-Messenger

NPC model provides great flexibility in the mediation. To make the mediation approach feasible in a heavy-loaded message broker, an efficient mediation implementation is also very important. WS-Messenger pays special attention to improve the XML performance and minimize mediation overheads in both the normalization processes and the customization processes. Several factors contribute to the efficiency of our mediation approach.

First, WS-Messenger stores normalized information sets in java bean objects and transforms SOAP messages directly to/from java beans. Since WS-Messenger is implemented in Java, this approach minimizes the overhead in message transformation. Java bean objects can be used directly by message processing modules. The other option we considered is to create XML schemas for normalized XML representations and perform transformation using XSLT (W3C, 1999). The advantage of this approach is that we can modify the mediation rules and add new

specification for mediation without changing the java code. However, XSLT transformation takes about 10's to a few 100's milliseconds depending message size and complexity (Bittner, 2004). This is slow for our purpose. Operations on Java objects are much more efficient. Also, XSLT style sheet is hard to read and maintain by human. Further, we don't expect the specifications on WS-based event notifications change very frequently, so updating or adding mediation rules is not a regular task.

Second, WS-Messenger manipulates lower level XML messages directly instead of processing a set of java-binding objects generated by a SOAP toolkit. This approach minimizes XML document processing overhead. By treating a SOAP message as an XML document, mediations are performed directly on the XML documents based on both WSE and WSN specifications. When transforming among different notification message formats, we also work on XML document directly. In this way we can perform arbitrary transformations easily and also achieve high performance required by the notification broker. The disadvantage of this approach is that it involves some lower level XML programming. However, we found that working with XML document directly is not harder than working with java objects generated by java-XML binding tools. It brings the flexibility we need for mediating among different XML documents. Our experience is in line with arguments in (Loughran & Smith, 2005) which challenges the java SOAP stack for building Web services and proposes an XML document centric approach.

Third, WS-Messenger uses an efficient XML pull parser called XPP (Sosnoski, 2001) to improve XML processing performance. XML parsing is expensive and should be avoided if possible. The pull mode of XPP can postpone parsing XML document until an element/attribute in the XML document is needed instead of parsing the whole XML document all at once. This feature allows fast scanning a XML document to find only the value of certain components. It is useful for notification brokers since only certain elements/attributes in the XML documents need to be parsed, such as the WS-Addressing part in the SOAP header. The message payload in a SOAP message does not need to be parsed. This can save much processing time if the message payload is large.

Fourth, WS-Messenger pre-builds some of notification message fragments at the subscription creation time for each consumer so that it doesn't have to create those fragments repetitively in the mediation process. For example, it pre-builds part of WS-Addressing headers for each event consumer based on their subscriptions so that it only needs to perform necessary XML operations unique to each notification message.

Fifth, WS-Messenger uses WS-Addressing implementation in our XSUL toolkit ("XSUL web site", 2005). It supports multiple WS-Addressing versions and switching/transforming among them is very easy. We found that WS-Addressing transformations are difficult in some other WS-Addressing implementations, such as Apache Addressing ("Apache Addressing website", 2006), since they do not expose interfaces to change the WS-Addressing versions. It is possible to modify the source code to change the default WS-Addressing version, but to transform between them is a challenge since no API is exposed for this purpose.

We will show that our mediation approach is very efficient from the data we gathered in the performance tests in the next section.

Mediation Performance Evaluations

Notification brokers handle high volume of notification messages. High performance is crucial. In order to quantitatively evaluate the mediation overhead, performance tests were conducted using a computer with dual Intel Pentium 4 3.20GHz CPU and 1GB of RAM, running Linux gentoo, JDK5.

Since we are interested in the mediation overheads, we used the simplest one publisher, one event consumer scenario. Four test cases are conducted: WSN to WSE (publishing a wrapped WS-Notification formatted notification message to a WS-Eventing consumer), WSN to WSN, WSE to WSE and WSE to WSN. To eliminate time synchronization problem and minimize the network delay, all entities were running on the same machine. We first measured the average processing times in all four test cases. A timestamp (t_0) was taken and embedded in each published notification message. Another timestamp (t_1) was taken when the event consumer received them. The processing time t was measured as $t=t_1-t_0$. The average processing time was the average of all t values in each test round.

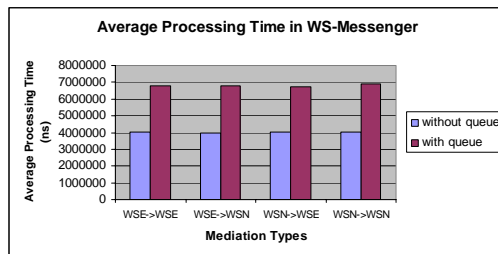


Fig. 13: Average processing time for different mediation types on notification messages

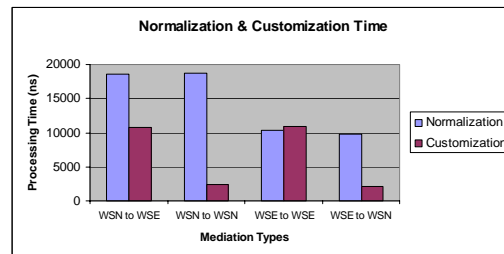


Fig. 14: Normalization and customization time for different mediation types on notification messages.

We compared the average processing time in two different situations (with queue and without queue) for mediating between the WS-Eventing notification message format and the wrapped WS-Notification message format. Message queues are important in notification brokers in a publish/subscribe systems. It enables a notification broker to acknowledge a publisher immediately when it receives a notification message without having to wait for message delivery. However, it introduced some overhead in the notification processing. In order to get the idealized shortest possible processing time as a baseline, we also bypassed the queue in the broker (See Fig. 4).

From this result graph in Fig 13, we can see that the average processing time is about 4 ms without message queue and about 6.8 ms using the activeMQ message queue. Surprisingly, the processing time for all mediations is almost the same. We could not tell any differences among the different cases in this performance test.

We investigated into this question and measured the processing time of just the normalization and customization processes. The new `System.nanoTime()` method in JDK5 is used in this measurement to get the most precise possible system timer. The difference between WS-Eventing and WS-Notification stands out. Fig. 14 shows the results:

(1) Even though NIIS uses WS-Notification as the internal message format, WS-Eventing incoming messages takes shorter time in the normalization process. This is because wrapped WS-Notification messages have more complicated XML structure than WS-Eventing messages. Fetching values in these extra XML structures takes more time than wrapping a WS-Eventing notification message into the WS-Notification format.

(2) In the customization process, it takes less time to create a wrapped WS-Notification formatted notification message since no transformation is needed.

(3) The total mediation overhead (Normalization time and Customization time) of the “WSN to WSE” case is the largest (29 μ s), while that of the “WSE to WSN” case is the smallest (12 μ s).

(4) This explains why we cannot see the difference in our previous tests as the difference (17 μ s) only accounts for about 0.4% of the average processing time (4.0 ms), which is already much faster than other implementation as described in (Huang, et al., 2006).

(5) The largest overhead (29 μ s) is only 0.7% of the average processing time (4.0 ms). This shows our implementation of the mediation is very efficient and adds almost no overhead to the message broker.

Interoperability tests with GT4

Currently the Globus Toolkit 4 (GT4) implementation only supports the WS-BaseNotification specification (version 1.2) (Sotomayor, 2006). It does not have a notification broker implementation and uses the “tightly coupled” publish/subscribe model. WS-Messenger can be used as a notification broker for GT4 to alleviate the burden on an event source and to decouple the event source and event consumers.

We have confirmed interoperability with GT4 for creating subscriptions and delivering notification messages. The interoperability test is based on the auction sample program discussed in (Sundaram, 2006). The tests we conducted include (1) using a WS-Messenger client to subscribe to the GT4’s notification service and receive the notifications triggered by the *ResourcePropertyValueChangeNotification* in GT4 and (2) using a GT4 client to subscribe to the WS-Messenger broker and receive notifications created by a WS-Messenger publisher. WS-Messenger acts as a mediator between these two clients, as shown in Fig. 15. Test results show that GT4 and WS-Messenger can successfully interoperate with each other in creating subscriptions and delivering event notifications.



Fig. 15: Interoperability tests with GT4

Applications of WS-Messenger

An application we first described in (Huang, et al.) is the LEAD (Linked Environments for Atmospheric Discovery) project ("LEAD project website", 2006). LEAD project “addresses the limitations of current weather forecast frameworks through a new, service-oriented architecture capable of responding to unpredicted weather events and response patterns in real time” (Plale, et al., 2005).

Event notification system plays an important role in the communications among various Web services involved in the LEAD project. A Grid workflow engine orchestrates these services. The workflow engine sequences the workflow tasks based on the notification messages. WS-Messenger is applied in the LEAD project to create all the entities in the notification system, including the event consumers, the event sources and the notification broker. Figure 16 shows the architecture of the notification system in the LEAD project.

Some services are event consumers. They need information about workflow execution status, output data location, etc. Specifically, the services include:

Metadata catalog services that manage personal metadata on previous and current experiments, including data files, workflow configuration and execution logs and so on.

Data provenance services that keep track of the derivation history of scientific data.

Workflow engines that orchestrate the execution of workflow.

Real-time workflow monitors that display the status of a running workflow in a graphic interface (Gannon, et al., 2005).

Some services are event sources. They publish notification messages on status updates, e.g. workflow execution status, output data location. We developed a generic web services factory to covert executable application programs to web services and generate notification messages about the execution status (Gannon, et al., 2005). Many services in the LEAD project are long running processes that may take hours or even days to finish. Notification services, working together with the workflow engine, can monitor the status of long-running processes and automate the process of workflow execution. Some sample services that are event sources include:

Decoder services that decode raw data from instrument to well-formatted data for further software processing.

Visualization services that convert the simulation output to a movie or images.

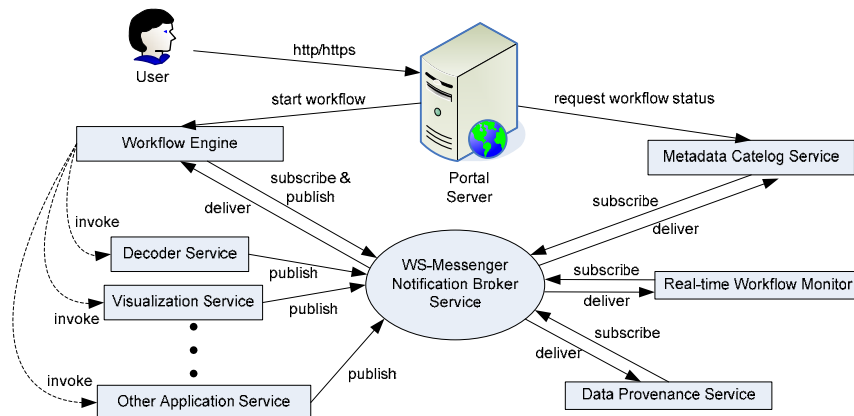


Fig. 16: Notification system architecture in the LEAD project

Topic-based subscription is currently used in our model because of its simplicity and scalability. Before invoking any services, the workflow engine sends subscription requests to the notification broker. It also subscribes itself to the broker to receive state-change notifications about a workflow. Other event consumer services, such as monitoring services, logging services, also send subscription requests to the broker about the workflow. When the workflow engine invokes a service, in addition to passing the application parameters, it also passes the location of the notification broker and specifies the topic for the notification service. The invoked services publish event notifications to the notification broker, which then sends the messages to the subscribed parties. The workflow engine orchestrates different services based on the status notifications of each service. When a workflow finishes or has errors, the workflow engine sends either “workflow_finished” or “workflow_failed” notifications message and "unsubscribe" from the notification broker. Other service components also "unsubscribe" themselves when they receive either “workflow_finished” or “workflow_failed” notifications.

The users access the Grid workflow from a web portal interface. They can compose workflows using a graphic interface, select the data for the workflows, execute the workflows and monitor the workflows (Gannon, et al., 2005). The publish/subscribe notification mechanism and topic

creation are transparent to the end users. They only need to care about the simulation workflow and selecting the appropriated data for the workflow. The topics are generated automatically and uniquely by the portal server based on a user's information, such as userID, workflow name, etc. The portal server passes the topic to the workflow engine when an end user invokes the workflow through the Web portal.

Related Works

There are many notification broker implementations available. Examples of such systems are Gryphon (Banavar, et al., 1999), Siena (Carzaniga, et al., 2000) , JEDI (Cugola, et al., 2001), Hermes (Pietzuch & Bacon, 2002), ActiveMQ (ActiveMQ), and NaradaBrokering (Fox & Pallickara, 2003). They have proposed different ways of managing subscriptions and delivering notification messages in a scalable and efficient way. Some of them have implemented or are working on creating Web services interfaces based on either WS-Eventing or WS-Notification. WS-Messenger project is based on previous research efforts and concentrates its research efforts on WS-based event notification scenario.

Mediation has been a practical approach in solving the incompatibility issues caused by having different standards in our daily life. For example, voltage converters mediate among different power supply standards among different countries, translators mediate among different languages among different people. Mediation also has many research activities in the database field.

Mediation among Web services is getting more and more research attention since it is an important component to enable interoperability among different Web services. Most of the available approaches are based on ontology, such as (Cabral & Domingue, 2005; Valle & Cerizza, 2005). However, ontology technologies are very complicated and have not been adapted widely in the Web services community (McCool, 2005). Schmidt et al. (M.-T. Schmidt, et al., 2005) summarized ten basic mediation patterns in Enterprise Services Buses (ESB). Hérault et al. proposes a mediator middleware in the ESB in a position paper (Hérault, et al., 2005). However, none of the previous works addressed the mediation of competing Web services specifications.

To the best of our knowledge, WS-Messenger is the first open source implementation that supports two competing Web services specifications at the same time and provides mediation between them. Competing Web services specifications have well defined SOAP message formats and certain message contents, but no ontology description is available. Our mediation approach takes advantage of these well defined message formats and contents in each specification and provides flexible and efficient meditations between them. Some unique characters of our mediation approach are:

(1) It takes advantage of both message formats and message contents defined in each specification and automatically detects the semantic information based on SOAP messages received. WS-Messenger supports both specifications even though each specification defines SOAP message formats in a different way. It can automatically detect the specification a client follows based on incoming messages. No additional semantic information or registration is needed to perform the mediation. From a client's perspective, it is a notification broker service that follows its own specification. Mediations are transparent to the client.

(2) It is flexible. Our mediation approach is based on a generic NPC model that can easily adapt to multiple competing Web services. It concentrates on the core functionalities of both specifications. The formats of Web Services messages defined in the specifications are treated as information set representation to achieve the core functionalities. Since both WS-Eventing and

WS-Notification specifications may change in the future, WS-Messenger is designed with future changes in mind.

(3) It is efficient and has high performance. The performance overhead is very small. Our approach treats SOAP messages as XML documents and manipulates them directly. Special attentions are paid to increase the performance of our mediation implementation.

(4) The approach we used is extensible. Although our implementation concentrates on reconciling the differences between WS-Eventing and WS-Notification, it is possible to extend our approach to reconcile the differences among other competing Web services specifications or competing Web services and achieve effective and flexible mediations.

Conclusions

In this article, we identified several challenges in exploring the full potential of Web services technology to create a global interoperable message delivery network. WS-Messenger addresses some of these challenges by leveraging existing notification messaging systems to provide the interoperability of WS-based publish/subscribe systems. It supports both WS-Eventing and WS-Notification at the same time through a mediation approach.

One key challenge in the mediation is how to maximize both flexibility and efficiency. A Normalization-Processing-Customization (NPC) model is proposed to mediate among competing Web services. This model is applied in WS-Messenger to provide mediation between the WS-Notification specification and the WS-Eventing specification. Several mediation techniques used in WS-Messenger are discussed. The NPC model is flexible and scalable. Our mediation design and implementation is very efficient and the mediation overhead is negligible in the total processing time of notification messages.

We have applied WS-Messenger to the Grid workflow orchestration in the LEAD project and found that it is a practical approach to decouple the service components in the service-oriented Grid computing systems and to orchestrate Grid workflows.

ACKNOWLEDGMENT

This research is supported by NSF ITR Grant ATM-0331480 and NMI ANI-0330613 and the Department of Energy Office of Science DE-FC02-01ER25492.

REFERENCES

- ActiveMQ. from <http://activemq.codehaus.org/>
- Apache Addressing website. (2006). from <http://ws.apache.org/addressing/>
- Apache Tomcat Project. (2006). from <http://tomcat.apache.org/>
- Banavar, G., et al. (1999). An efficient multicast protocol for content-based publish-subscribe systems. *Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS'99)*.
- Bittner, T. (2004). Performance Evaluation for XSLT Processing *Technical Report, University of Rostock*.
- Box, D., et al. (2004). Web Services Eventing.
- Box, D., et al. (2004). Web Services Addressing (WS-Addressing). from <http://www.w3.org/Submission/ws-addressing/>
- Cabral, L. & Domingue, J. (2005). Mediation of Semantic Web Services in IRS-III. *Proceedings of the First International Workshop on Mediation in Semantic Web Services (MEDIATE 2005)*.
- Carzaniga, A., et al. (2000). Achieving scalability and expressiveness in an internet-scale event notification service. *Proceeding of Nineteenth ACM Symposium on Principles of Distributed Computing (PODC 2000)*.

- Catania, N., et al. (2003). Web Services Events (WS-Events) Version 2.0. from <http://devresource.hp.com/drc/specifications/wsmf/WS-Events.pdf>
- Chappell, D. & Liu, L. (2004). Web Services Brokered Notification (v1.2). from <http://docs.oasis-open.org/wsn/2004/06/wsn-WS-BrokeredNotification-1.2-draft-01.pdf>
- Clark, J. & DeRose, S. (2006). XML Path Language (XPath) Version 1.0. from <http://www.w3.org/TR/xpath>
- Cline, K., et al. (2006). Toward Converging Web Service Standards for Resources, Events, and Management.
- Cugola, G., et al. (2001). The Jedi event-based infrastructure and its application to the development of the opss wfms. *IEEE Transactions on Software Engineering*.
- Diao, Y. & Franklin, M. J. (2003). High-Performance XML Filtering: An Overview of YFilter. *IEEE Data Engineering Bulletin*(March, 2003).
- Eugster, P. T., et al. (2003). The Many Faces of Publish/Subscribe. *ACM Computing Surveys*, 35(2).
- Foster, I., et al. (2002). Grid Services for Distributed System Integration. *Computer*, 35(6).
- Foster, I., et al. (2003). The Physiology of the Grid. In F. Berman, G. Fox & T. Hey (Eds.), *Grid Computing: Making the Global Infrastructure a Reality*.
- Fox, G. & Pallickara, S. (2003). NaradaBrokering: An Event-based Infrastructure for Building Scalable Durable Peer-to-Peer Grids. In *Grid Computing: Making the Global Infrastructure a Reality*.
- Gannon, D., et al. (2005). Service Oriented Architectures for Science Gateways on Grid Systems. *International Conference on Service Oriented Computing 2005*.
- Gisolfi, D. (2001). Is Web services the reincarnation of CORBA? , from <http://www-106.ibm.com/developerworks/webservices/library/ws-arc3/#resources>
- Gokhale, A., et al. (2002). *Proceedings International WWW Conference 2002*.
- Graham, S. & Murray, B. (2004). Web Services Base Notification(v1.2). from <http://docs.oasis-open.org/wsn/2004/06/wsn-WS-BaseNotification-1.2-draft-03.pdf>
- Hapner, M., et al. (2002). Java Message Service (Version 1.1). from <http://java.sun.com/products/jms/>
- Hérault, C., et al. (2005). Mediation and Enterprise Service Bus: A position paper *Proceedings of the First International Workshop on Mediation in Semantic Web Services (MEDIATE 2005)*.
- Huang, Y. & Gannon, D. (2006a). A Comparative Study of Web Services-based Event Notification Specifications. *ICPP Workshop on Web Services-based Grid Applications*
- Huang, Y. & Gannon, D. (2006b). A Flexible and Efficient Approach to Reconcile Different Web Services-based Event Notification Specifications. *Proceedings of International Conference on Web Services (ICWS)*.
- Huang, Y., et al. (2006). WS-Messenger: A Web Services based Messaging System for Service-Oriented Grid Computing. *International Symposium on Cluster Computing and the Grid (CCGrid06)*.
- Humphrey, M., et al. (2005). Alternative Software Stacks for OGSA-based Grids. *Proceedings of Supercomputing 2005*.
- Iwasa, K., et al. (2004). WS-Reliability 1.1. from http://docs.oasis-open.org/wsrn/ws-reliability/v1.1/wsrn-ws_reliability-1.1-spec-os.pdf
- LEAD project website. (2006). from <http://www.leadproject.org>
- Loughran, S. & Smith, E. (2005). Rethinking the Java SOAP Stack. *IEEE International Conference on Web Services (ICWS) 2005*.
- M.-T. Schmidt, et al. (2005). The Enterprise Service Bus: Making service-oriented architecture real. *IBM Systems Journal*, 44(4).
- McCool, R. (2005). Rethinking the Semantic Web, Part 1. *IEEE Internet Computing (November/December, 2005)*, 9(6).
- Nadalin, A., et al. (2004). WS-Security 1.0. from <http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-soap-message-security-1.0.pdf>
- Nielsen, H., et al. (2002). Direct Internet Message Encapsulation from <http://msdn.microsoft.com/library/en-us/dnglobspec/html/draft-nielsen-dime-02.txt>
- OASIS. (2004). WS-Notification (v1.2). from <http://docs.oasis-open.org/wsn/2004/06/>
- OASIS organization. (2006). from <http://www.oasis-open.org/>
- OMG. (2004a). Common Object Request Broker Architecture: Core Specification. from <http://www.omg.org/docs/formal/04-03-01.pdf>
- OMG. (2004b). CORBA Event Service Specification. from http://www.omg.org/technology/documents/formal/event_service.htm

- OMG. (2004c). CORBA Notification Service Specification. from http://www.omg.org/technology/documents/formal/notification_service.htm
- OpenJMS. from <http://openjms.sourceforge.net/index.html>
- Parastatidis, S., et al. (2006). Asynchronous Messaging between Web Services Using SSDL. *IEEE Internet Computing (January/February, 2006)*, 10(1), 26-39.
- Pietzuch, P. & Bacon, J. (2002). *Hermes: A Distributed Event-Based Middleware Architecture*. Paper presented at the Workshop on Distributed Event-Based Systems (DEBS).
- Plale, B., et al. (2005). Cooperating Services for Data-Driven Computational Experimentation *Computing in Science & Engineering*, 7(5).
- Shirasuna, S., et al. (2004). *Performance Comparison of Security Mechanisms for Grid Services*. Paper presented at the 5th IEEE/ACM International Workshop on Grid Computing.
- Sosnoski, D. (2001). XML and Java technologies: Document models. from <http://www-128.ibm.com/developerworks/xml/library/x-injava/index.html>
- Sotomayor, B. (2006). The Globus Toolkit 4 Programmer's Tutorial. from <http://gdp.globus.org/gt4-tutorial/multiplehtml/index.html>
- Sundaram, B. (2006). WS-Notification and the Globus Toolkit 4 WS-Java Core. from <http://www-128.ibm.com/developerworks/grid/library/gr-wsngt4/>
- Valle, E. D. & Cerizza, D. (2005). The mediators centric approach to Automatic Web Service Discovery of COCOON Glue. *Proceedings of the First International Workshop on Mediation in Semantic Web Services (MEDIATE 2005)*.
- Vambenepe, W. (2004). Web Services Topics (v1.2). from <http://docs.oasis-open.org/wsn/2004/06/wsn-WS-Topics-1.2-draft-01.pdf>
- W3C. (1999). XSLT specification. from <http://www.w3.org/TR/xslt>
- W3C. (2000). Extensible Markup Language (XML) 1.0 (Second Edition). from <http://www.w3.org/TR/2000/REC-xml-20001006>
- W3C. (2001). Web Services Description Language (WSDL) 1.1. from <http://www.w3.org/TR/wsdl>
- W3C. (2003). SOAP Version 1.2. from <http://www.w3.org/TR/soap12-part1/>
- W3C website. (2006).
- XSUL web site. (2005). from <http://www.extreme.indiana.edu/xgws/xsul/index.html>

ABOUT THE AUTHOR

Yi Huang Computer Science Department, Indiana University, 150 S. Woodlawn Avenue, Bloomington, Indiana 47405 (yihuan@cs.indiana.edu). Mr. Huang is a doctoral candidate in the Computer Science Department at Indiana University, under the guidance of Dr. Dennis Gannon. He holds a B.E. degree from Beijing University of Technology, China, and an M.S. degree from Florida State University. Mr. Huang's research interests include web services, distributed messaging systems, grid computing, system integration, and analysis of message pattern and workflow.

Dennis Gannon Computer Science Department, Indiana University, 150 S. Woodlawn Avenue, Bloomington, Indiana 47405 (gannon@cs.indiana.edu). Dr. Gannon is a professor of computer science at Indiana University. He received his Ph.D. degree in computer science from the University of Illinois and his Ph.D. degree in mathematics from the University of California. He was on the faculty at Purdue University from 1980 to 1985. Dr. Gannon's research interests include software tools for high-performance distributed systems and problem-solving environments for scientific computation.