WS-Messenger: A Web Services-based Messaging System for Service-Oriented Grid Computing

Yi Huang, Aleksander Slominski, Chathura Herath, and Dennis Gannon
Department of Computer Science, Indiana University,
Bloomington, Indiana, 47405, USA
Email:{yihuan, aslom, cherath, gannon}@cs.indiana.edu

Abstract—**A Web services-based publish/subscribe system has the potential to create an Internet scale interoperable event notification system which is important for Grid computing as it evolves a service-oriented architecture. WS-Messenger is designed to be a Web services-based message broker that can decouple event producers and event consumers and achieve scalable, reliable and efficient message delivery. In this paper, we discuss some challenges that are unique to Web services-based publish/subscribe systems and the key features that distinguish WS-Messenger from other existing message brokers. We then present the architecture and the technology used in WS-Messenger. Performance tests indicate WS-Messenger performs better than the WS-Notification implementation in Globus Toolkit 4 (GT4) and it can be used as a complement to GT4 to improve its scalability. We lastly describe its application to Grid workflow orchestration in the LEAD project.**

I. INTRODUCTION

Grid computing is a promising technology to connect heterogeneous resources across the world and coordinate them for scientific computing. Logging what happened to these resources, auditing the usage of these resources and monitoring the state changes of these resources are important tasks for Grid applications. Updates of resources status need to be disseminated among multiple parties in a Grid system in the form of *"Event messages"* or *"Notification messages"*. Some sample events in a Grid application include: a resource starts or finishes a job, a resource creates an output file, or an error or exception happens to a resource.

The Publish/Subscribe paradigm is a practical pattern for disseminating these events to multiple entities in a Grid system. In this paradigm, an *event consumer* registers its interest for some specific kind of event using a "*subscribe*" operation and an *event source* "*publishes*" events to one or more event consumers based on their registered interests.

A Web services-based publish/subscribe system is an important component for service-oriented Grid computing as Grid computing is converging with Web services technologies. WS-Eventing specification [1] and WS-Notification specification [2] are two major specifications for Web services-based publish/subscribe systems. WS-Notification is adopted by Globus Toolkit 4 (GT4), which is the most widely used toolkit for building Grid computing applications. WS-Eventing is also a promising specification for building Grid applications because of its simplicity and the fact it being advocated by many software vendors, including Microsoft. The University of Virginia [3] implemented Open Grid Service Architecture (OGSA) [4] using WS-Transfer [5] and WS-Eventing [1] specifications. WS-Eventing is also used in the LEAD project [6] to communicate among different Grid services due to its simplicity. Since both WS-Eventing [1] and WS-Notification [2] specifications are used in different Grid computing systems, it is important to reconcile the differences between them and make them work together.

WS-Messenger is designed to create a scalable, reliable and efficient Web services-based message broker that sends Web services-based event notification messages among heterogeneous applications, platforms and Grid computing environments for logging, auditing and monitoring purposes. Higher level services, such as workflow orchestration services, grid resource management services, can use these event notification messages to keep track of the status of different components in distributed systems.

The key features that distinguish *WS-Messenger* from other existing message brokers are:

(1) It is based on Web services specifications and provides mediation between WS-Notification specification and WS-Eventing specification.

(2) It provides an extensible framework to leverage different existing underlying messaging systems so that it can adapt to different environments. The WS-Messenger project is concentrated on the Web services interface of a publish/subscribe system. It can utilize existing messaging systems to provide scalable subscription management and message delivery.

(3) It is light-weighted and has simple-to-use APIs to integrate with existing Java applications. A Java programmer only needs to add a couple lines of code to an existing java application to publish and receive Web services-based notifications. No deployment to any Web container is needed for a notification publisher or consumer since WS-Messenger implements the HTTP protocol and includes a mini HTTP server.

(4) It provides graphic interfaces for subscription management and debugging Web services-based publish/subscribe systems.

The rest of this paper is as follows. Section II provides an overview of the functionality of a notification broker in a traditional publish/subscribe system. Some unique challenges in Web services-based publish/subscribe systems are discussed in section III. Section IV contains the description of the architecture and the technology used in WS-Messenger. Some user-friendly graphic tools for WS-Messenger are demonstrated in section V. Interoperability test and performance test results are presented in section VI and section VII. We describe our experience in applying WS-Messenger to the Grid workflow engine in the LEAD project in Section VIII, and we conclude in Section IX.

II. NOTIFICATION BROKERS IN THE PUBLISH/SUBSCRIBE SYSTEMS AND RELATED WORKS

A notification broker is a key component for a scalable, loosely-coupled publish/subscribe system. It provides an abstraction layer between an event source and an event consumer so that they can communicate without knowing the location of each other. The event consumer can be offline when the event source publishes an event. The event source is relieved from the burden of handling subscription registrations and delivering events to all the event consumers.

There are many notification broker implementations available. Examples of such systems are Gryphon [7], Siena [8] , JEDI [9], Hermes [10], ActiveMQ [11], and NaradaBrokering [12]. 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.

III. CHALLENGES IN WEB SERVICES-BASED PUBLISH/SUBSCRIBE SYSTEMS

There are some unique challenges to Web services-based publish/subscribe systems that have not been addressed in traditional publish/subscribe systems. One major difference is that the scope of information disseminating is changed from closed intranets to the open Internet. Event sources and event consumers are less likely to be under the control of the same administrator. Also, notification messages formats are XML-based SOAP messages. This is uncommon in traditional Publish/Subscribe systems. In this Section, we will discuss some of the major challenges to Web services-based publish/subscribe systems.

A. 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 "message on the wire" that is interoperable. Optional elements defined in the specifications can also cause incompatibility when connecting two different implementations.

B. Mediation among different Web services-based publish/subscribe specifications

WS-Notification and WS-Eventing are two major specifications for Web services-based publish/subscribe 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.

C. Mediation among different transport mechanisms

Web services are transport-agnostic. 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 these transport protocols.

D. Scalable and efficient XML processing

In the traditional Publish/Subscribe messaging systems, the performance bottlenecks are usually message filtering and destinations 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.

E. Scalable and efficient XML-based content filtering

Most messages in traditional publish/subscribe messaging systems 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, WS-Eventing specification uses XPath [13] filtering as the default messaging filtering. How to create scalable and 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 Y-Filter [14].

F. Security

The 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 [15] and makes interoperability even harder.

IV. WS-MESSENGER ARCHITECTURE

WS-Messenger is an open source project developed at Indiana University. 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.

A. Architecture

Figure 1 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.

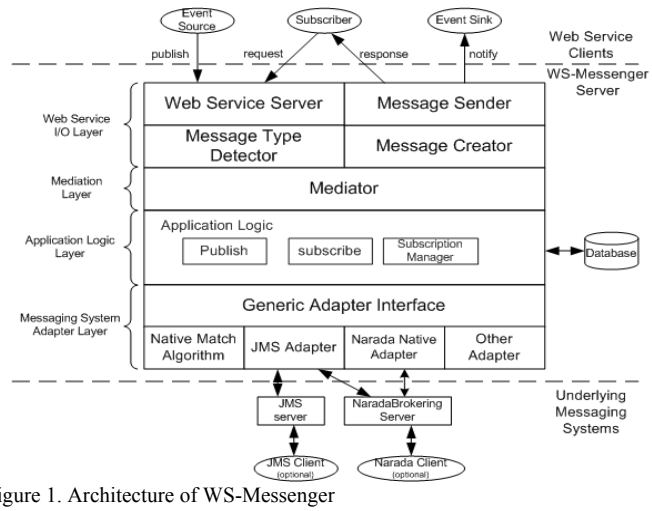

Figure 1. Architecture of WS-Messenger

The "*Web service I/O layer*" interacts with both WS-Eventing clients and WS-Notification clients. Different types of transport mechanisms can be used in the Web service I/O layer. Currently, we have implemented support for HTTP protocol and SOAP.TCP protocol. SOAP.TCP protocol is used in the Web Service Enhancement (WSE) package for Microsoft .NET platform. It sends and receives DIME [16] 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 WS-Addressing [17] 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 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 next section.

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 status 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 [12] native interface adapter. The JMS adapter can be used to integrate with most Java-based Publish/Subscribe messaging systems, e.g. openJMS [18], activeMQ [11]. 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 the 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.

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 does not have the desired subscription option, WS-Messenger cannot offer it. For example, few messaging systems have XML-based filtering capability while it is important for Web services-based publish/subscribe systems.

This problem can be solved by extending the generic adapter interface and providing a native matching algorithm in WS-Messenger. To use other message filtering packages that do not come with a publish/subscribe system, such as the XPath-based Y-Filter [14], one must create an adapter.

B. Mediation Technique in WS-Messenger

WS-Eventing and WS-Notification specifications are two competing specifications defining the interfaces for Web services-based publish/subscribe systems. The Web services interfaces and the notification message formats defined in these two specifications are different. Both specifications have been used in different Grid projects. The fact that both specifications are being used in different Grid projects creates obstacles in integrating and reusing their Web services. For example, the Web services in one project may generate WS-Eventing formatted notification messages. These messages cannot be consumed by a Web service developed using GT4 toolkit since that service expects WS-Notification formatted notification messages.

WS-Messenger helps to solve this problem through a mediation approach. As a notification broker, WS-Messenger implements both specifications and can accept SOAP messages following either specification. Two kinds of messages are expected by WS-Messenger: *Operation messages*, such as subscription requests, and *notification messages*. If an *operation message* is received, WS-Messenger will process it and send response messages using the same specification of the request message. When a subscription request is received from a client, the specification used in the request is stored together with the subscription information.

If a *notification message* is received, WS-Messenger needs to forward it to the interested event consumers. Depending on the event consumer type, either WS-Eventing or WS-Notification, WS-Messenger can automatically convert the notification message formats to makes sure that the message "on the wire" can be understood by the event consumers. For example, a *WS-Notification* consumer can receive notification messages published by a *WS-Eventing* publisher as long as it subscribes to that kind of messages, and vice versa. The event consumer type is determined by the subscription request message type. Here we assume that the subscribers and the event consumers use the same specification. This is a valid assumption for most cases since the subscribers needs to know the location of the event consumers. They are tightly coupled. A publisher can publish messages in any format to the WS-Messenger server. It makes no difference to the event consumers.

The mediation for notification messages is achieved through extracting the content of the notification messages received from the publisher and performing transformation according to the predefined mediation rules. Transformations performed include *namespace mediation*, *WS-Addressing mediation*, *message format mediation* and *adding default values*. Unlike many systems, WS-Messenger does not use a WSDL to java object tool to generate a SOAP skeleton and stub. Instead it manipulates SOAP messages directly as XML documents and performs message transformations efficiently.

1) Namespace Mediation

The namespaces used in the two specifications are different. Wrapped WS-Notification messages require the WS-BaseNotification namespace for the "*notify*" element, while WS-Eventing does not need to use the WS-Eventing namespace in the notification messages.

2) WS-Addressing Mediation

The WS-Addressing versions used in these two specifications are also different. WS-Notification uses the *2003/03* version, while WS-Eventing uses the *2004/08* version. Our own implementation of the WS-addressing specification is used in WS-Messenger.

3) Message format Mediation

There are three kinds of notification message formats: *wrapped WS-Notification format*, *raw WS-Notification format* and *WS-Eventing format*. The wrapped WS-Notification format wraps a notification message into a "Notify" element and add additional WS-Notification defined information (such as Topic) in it. The raw WS-Notification message format is similar to the WS-Eventing format except the WS-Addressing versions used in them are different. The internal message format in WS-Messenger is the wrapped WS-Notification format since that contains more information than the other two formats and can be transformed to the other format if needed. Transformation is needed when receiving notification messages in the WS-Eventing format or the raw WS-Notification format and when delivering notifications to the consumers requiring those two formats.

4) Using default values in the mediations

WS-Eventing specification does not require specifying a topic in a notification message, while a wrapped WS-Notification formatted message requires a topic. How to mediate the "*topic"* element is a challenge. WS-Messenger uses the following strategy to deal with this problem.

First, WS-Messenger tries to find the topic in the SOAP header of a WS-Eventing message. Although WS-Eventing specification does not require specifying a topic in a notification message, it is possible to specify a topic using SOAP header if needed. One scenario of specifying a topic in the WS-Eventing message is to use the *referenceProperties* or *referenceParameters* of the event consumer's EndPointReference in the subscription message. According to the WS-Addressing specification, when sending messages to the event consumer, each reference property and reference parameter element becomes a header block in the SOAP message. WS-Messenger can examine the SOAP header and see if it can find a topic element. If so, it will map that to the topic element in the WS-Notification message.

Second, if WS-Messenger cannot find a topic in the WS-Eventing message, a reserved default topic, *wseTopic*, is used in the transformation to a wrapped WS-Notification message for its internal processing. In this way, a WS-Notification subscriber for a specific topic will not get this message since it does not match the subscribed topic. However, a WS-Notification subscriber can get the message in the correct WS-Notification format if it subscribes to all the topics using a wildcard subscription. Similar case applies to a WS-Eventing subscriber that uses topic-based subscription. If a WS-Eventing subscriber subscribes using other kind of filtering criteria, like content-based filtering, WS-Messenger can detect that no topic is specified in the subscription and subscribe automatically to this special topic. In this way, it can still receive the message that contains no topic if it matches the filtering criteria.

C. API to Integrate WS-Messenger in Java Applications

The WS-Messenger package provides an easy-to-use java API for users to integrate the messaging system with their applications so that any java application can publish and receive Web services-based event notification messages. The java applications do not need to be deployed to a Web container for creating a web service or an event consumer since WS-Messenger has a light-weighted http server.

Switching the specification used in the java application is very easy. Depending on whether the user wants to use WS-Eventing or WS-Notification, she can create either a *WseClientAPI* object or a *WsntClientAPI* object in the application code. For example, *WseClient publisher=new WseClientAPI()* creates a publisher following the WS-Eventing specification. Calling *publisher.publish(consumer, producer, topic, message)* can publish the message to a consumer or a notification broker.

D. Delivering Messages to Event Consumers behind the Firewalls

Firewalls can block the communications between event sources and event consumers in the Internet scale messaging systems. It is hard to send event notifications to an event consumer behind a firewall. Opening firewalls to allow communications for some applications is not practical in many cases. To solve this problem, we created a *MessageBox* web service [19] that runs outside of the firewall and can store notification messages for the event consumers behind the firewalls.

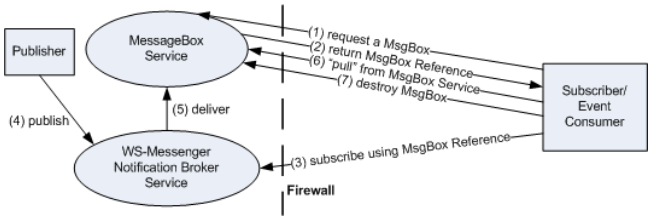

Figure 2. Using *MessageBox* service to deliver notification messages to event consumers behind firewalls

Figure 2 illustrates the steps to use the message box service. First, an event consumer requests a MessageBox web service to create a message box for it and receives a reply containing the reference to the created message box. The reference of the message box created is expressed using EndpointReferenceProperties in the WS-Addressing specification. Then it sends a subscription request to the WS-Messenger service using the message box reference as the event consumer location. WS-Messenger will then deliver notification messages to the specified message box. The event consumer can then periodically *pull* notification messages from the specified message box in the MessageBox service. When the message box is no longer needed, the subscriber can send another web service request to destroy the message box.

V. TOOLS FOR MANAGING AND DEBUGGING WEB SERVICES-BASED PUBLISH/SUBSCRIBE SYSTEMS

Monitoring and debugging in asynchronous Web services-based publish/subscribe systems can be challenging without appropriate tools. We created some tools to simplify the management of notification brokers and debugging processes in Web services-based publish/subscribe systems.

A. Subscription Manager Web Interface

The Subscription Manager Web Interface is a servlet that can access any WS-Messenger broker using standard Web service invocations. It can display all the subscriptions on that broker and can delete unnecessary subscriptions. The administrator does not need to enter the subscriptionId to delete a subscription. She only needs to check the checkbox of the subscriptions to be deleted and click the "delete" button. Multiple brokers can be managed through this simple Web interface by switching between different broker

URLs. This interface can be embedded in an administration portal as shown in Fig.3 below.

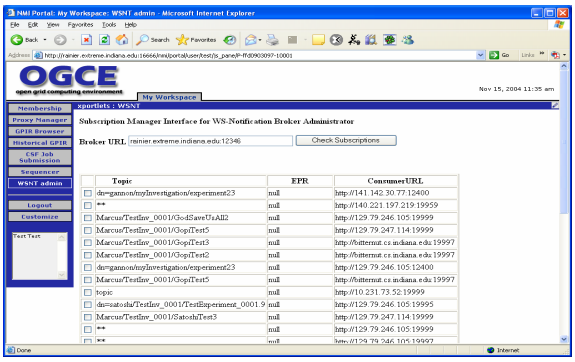

Figure 3. Subscription Manager Interface

B. *Event Notification Message Viewer*

The "*Event Notification Message Viewer*" (Fig. 4) can act as either a WS-Notification client or a WS-Eventing client that subscribes to a topic and it starts listening to that topic when the user clicks the "start" button. It can be used to check the message flow on an event channel, check the availability of the notification broker, check messages on the wire in the publish/subscribe system and check firewall problems that may block the delivery of the notification messages. The received notification can be displayed in "brief message" form or "whole message" form. The "brief message" form displays the message content only. The "whole message" form displays whole SOAP messages, including SOAP headers. To help checking the firewall problem that may block the delivery of messages, the "Notification Message Viewer" can switch to "Pull" mode and periodically pull messages from a message box service outside of the firewall.

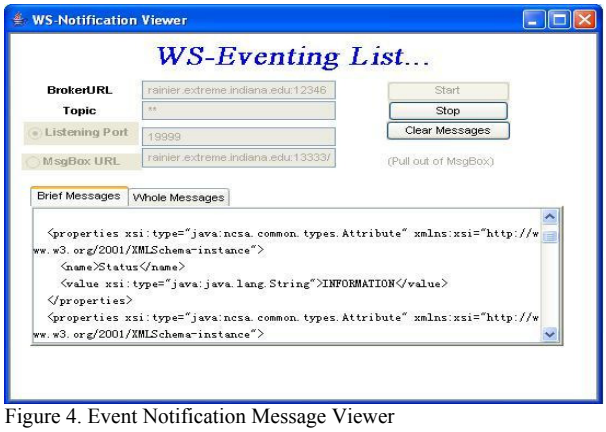

Figure 4. Event Notification Message Viewer

VI. INTEROPERABILITY WITH GT4

Currently the Globus Toolkit 4 (GT4) implementation only supports the WS-BaseNotification specification (version 1.2) [20]. 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 [21]. 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. Our results show that GT4 and WS-Messenger can successfully interoperate with each other in creating subscriptions and delivering event notifications subject to one small problem.

An issue we encountered in our interoperation test is a transport layer problem. The default transport protocol used by GT4 is HTTP 1.1. Our implementation uses HTTP 1.0 protocol which does not understand the "chunked encoding" in HTTP 1.1. To get around with this problem, we need to configure GT4 to use HTTP 1.0 protocol by modifying a line in the *client-config.wsdd* configuration file in GT4. In the future version of WS-Messenger, we will add HTTP 1.1 support to our Web server so that we do not need to change the default transport protocol in GT4 to interoperate with it. We will also carry out more interoperability tests on other operations defined in the WS-Notification specification.

VII. PERFORMANCE EVALUATION

We conducted performance tests on WS-Messenger (version 1.43.0) and compared its performance with the WS-Notification implementation in GT4 (version 4.0.1). The performance tests were performed using a computer with dual Intel Pentium 4 2.80GHz CPU and 1GB of RAM, running Linux gentoo. Performance data were taken after each event consumer had received 10 messages to ensure all the components had adequate starting times. Publishing rate was kept sufficiently low to make sure that one notification message was delivered to all the event consumer(s) before publishing the next message to the notification broker/GT4 Server. A timestamp (*t0*) was taken and embedded in each published notification message. Another timestamp (*t1*) was taken when an event consumer received the notification messages. The *processing time t* by WS-Messenger or GT4 service was measured as *t=t1-t0*. The *average processing time* was the average of all *t* values reported by each consumer and in each test round. To eliminate time synchronization problem and minimize the network delay, all entities were running in the same machine.

Fig.5 shows the *average processing time* to send a notification message (587 Bytes in size) to varying number

of event consumers. The *average processing time* for sending the message in both WS-Messenger and the GT4 increases linearly as the number of event consumers increases. However, there is noticeable performance difference. WS-Messenger performs much better than GT4.

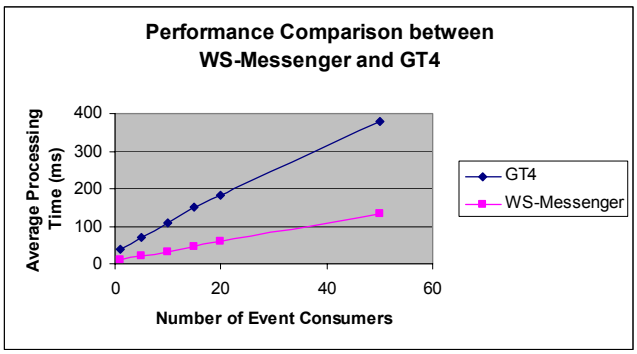

Figure 5. Performance Evaluation

We think the performance advantage of WS-Messenger is mainly attributed to two factors:

(1) The underlying SOAP toolkits have performance difference. WS-Messenger uses XSUL [22] to process the received SOAP messages, which performs better than the Apache Axis used by GT4 as described in the performance comparison in [23].

(2) WS-Messenger eliminates databinding overhead. WS-Messenger processes SOAP messages directly at the XML message level without creating databinding between XML elements and java objects. This can reduce some overhead in SOAP message processing.

VIII. Application in Grid Systems

The LEAD (Linked Environments for Atmospheric Discovery) project [24] is a Grid project that "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" [6].

Notification system plays an important role in the communications between 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 6 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 configurations and execution logs and so on,

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

A Workflow engine that orchestrates the execution of workflow,

Real-time workflow monitors that display the status of a running workflow in a graphic interface [25].

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 [25]. 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.

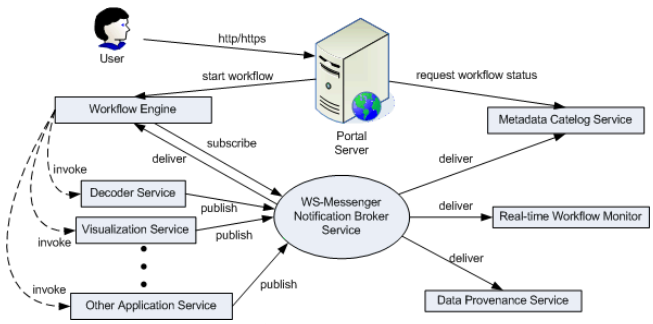

Figure 6. Notification system architecture in the LEAD project

Subscription creation and destruction are controlled by the workflow engine. Topic-based subscription is currently used in our model because of its simplicity and scalability. Before invoking any services, the workflow engine creates an event consumer and sends subscription requests to the notification broker. It also subscribes itself to the broker to receive state-change notifications about a workflow. It may also send other subscription requests for other services, such as the monitoring services, logging services, etc. 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, the workflow engine shuts down the event consumers and sends "unsubscribe" requests to the notification broker to

cancel the subscriptions for itself and other service components.

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 [25]. 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.

IX. Conclusions and Future Works

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 Web services-based publish/subscribe systems. It supports both WS-Eventing and WS-Notification at the same time through a mediation approach.

WS-Messenger can be complementary to GT4 to decouple event producers and event consumers. Interoperability test and performance comparison test have been conducted between WS-Messenger and GT4, and the results show that they can interoperate with each other successfully and there is significant performance lead of WS-Messenger over GT4.

We have applied WS-Messenger to the Grid workflow engine 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. We also developed GUI tools that can be used for debugging and monitoring Web services-based publish/subscribe systems.

The work of WS-Messenger is ongoing. We are applying it to real world service-oriented Grid projects and continuously improving it and adding new functionalities. As part of the future works, we would like to add XPath-based filter support, integrate with WS-ReliableMessage, add security support, and improve service reliability and scalability, etc.

Acknowledgements

The authors are grateful for Yong Zhao of the University of Chicago for his valuable comments to improve this paper. We would also like to thank the anonymous referees for helpful comments.

References:

[1] D. Box and et.al, Web Services Eventing, Available: http://ftpna2.bea.com/pub/downloads/WS-Eventing.pdf

[2] OASIS, WS-Notification (v1.2), Available: http://docs.oasis-open.org/wsn/2004/06/

[3] M. Humphrey, G. Wasson, *et al.*, "Alternative Software Stacks for OGSA-based Grids," *Proceedings of Supercomputing 2005*, 2005.

[4] I. Foster, C. Kesselman, *et al.*, "Grid Services for Distributed System Integration," *Computer*, vol. 35, 2002.

[5] J. Alexander and et.al., Web Service Transfer, Available: http://msdn.microsoft.com/library/en-us/dnglobspec/html/ws-transfer.pdf

[6] B. Plale, D. Gannon, *et al.*, "Cooperating Services for Data-Driven Computational Experimentation," *Computing in Science & Engineering*, vol. 7, 2005.

[7] G. Banavar, T. Chandra, *et al.*, "An efficient multicast protocol for content-based publish-subscribe systems," *Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS'99)*, 1999.

[8] A. Carzaniga, D. S. Rosenblum, *et al.*, "Achieving scalability and expressiveness in an internet-scale event notification service," *Proceeding of Nineteenth ACM Symposium on Principles of Distributed Computing (PODC 2000)*, 2000.

[9] G. Cugola, E. D. Nitto, *et al.*, "The jedi event-based infrastructure and its application to the development of the opss wfms," *IEEE Transactions on Software Engineering*, 2001.

[10] P. Pietzuch and J. Bacon, "Hermes: A Distributed Event-Based Middleware Architecture," presented at Workshop on Distributed Event-Based Systems (DEBS), 2002.

[11] ActiveMQ, Available: http://activemq.codehaus.org/

[12] G. Fox and S. Pallickara, "NaradaBrokering: An Event-based Infrastructure for Building Scalable Durable Peer-to-Peer Grids," in *Grid Computing: Making the Global Infrastructure a Reality*, 2003.

[13] J. Clark and S. DeRose, XML Path Language (XPath) Version 1.0, Available: http://www.w3.org/TR/xpath

[14] Y. Diao and M. J. Franklin, "High-Performance XML Filtering: An Overview of YFilter," *IEEE Data Engineering Bulletin*, 2003.

[15] S. Shirasuna, A. Slominski, *et al.*, "Performance Comparison of Security Mechanisms for Grid Services," presented at 5th IEEE/ACM International Workshop on Grid Computing, 2004.

[16] H. Nielsen, H. Sanders, *et al.*, Direct Internet Message Encapsulation, Available: http://msdn.microsoft.com/library/en-us/dnglobspec/html/draft-nielsen-dime-02.txt

[17] D. Box, F. Curbera, *et al.*, Web Services Addressing (WS-Addressing), Available: http://www.w3.org/Submission/ws-addressing/

[18] OpenJMS, Available: http://openjms.sourceforge.net/index.html

[19] D. Caromel, A. d. Costanzo, *et al.*, "Asynchronous Peer-to-PeerWeb Services and Firewalls," *7th International Workshop on Java for Parallel and Distributed Programming (IPDPS 2005)*.

[20] B. Sotomayor, The Globus Toolkit 4 Programmer's Tutorial, Available: http://gdp.globus.org/gt4-tutorial/multiplehtml/index.html

[21] B. Sundaram, WS-Notification and the Globus Toolkit 4 WS-Java Core, Available: http://www-128.ibm.com/developerworks/grid/library/gr-wsngt4/

[22] XSUL web site, Available: http://www.extreme.indiana.edu/xgws/xsul/index.html

[23] M. R. Head, M. Govindaraju, *et al.*, "A Benchmark Suite for SOAP-based Communication in Grid Web Services," *Proceedings of Supercomputing 2005 (SC 2005)*, 2005.

[24] LEAD project website, Available: http://lead.ou.edu

[25] D. Gannon, B. Plale, *et al.*, "Service Oriented Architectures for Science Gateways on Grid Systems," *International Conference on Service Oriented Computing 2005*, 2005.