Enabling Web Service Extensions for Scientific Workflows

Srinath Perera, Dennis Gannon
Computer Science Department
Indiana University
Bloomington IN 47405
{hperera, gannon}@cs.indiana.edu

Abstract

Web Services have become an integral part of workflow orchestration in scientific applications and many tools used by scientists including Kepler, Taverna and Triana incorporate Web Services. However orchestration of complex workflows that involve services that exploit standards like WS-Addressing or WS-Security are not easily managed by these tools. Most complex use cases that involve "add on" service standards use implementation specific assumptions, and as a result, the services and workflow composers become tightly coupled. This leads to stove-piped, non-interoperable implementations. This paper describes an effort to implement complex use cases that include asynchronous messaging and WS-Security, for a large application project called LEAD, while maintaining standard conformance and composer simplicity. The primary contribution of the paper is design of a Mediator and a generic Web Service actor that allow the addition of new Web Service standards to services in a workflow, without the need to make the workflow composer or enactor explicitly aware that these standards are being used. These concepts are demonstrated by an implementation that allows large workflows to be constructed using two different scientific workflow systems that were not designed with these extensions in mind.

1. Introduction

Web Services are increasingly being used in workflow management for large scientific applications. This is due in part because of the transformation of Grid computing standards to a service oriented architecture model and because Web Services based workflow models like BPEL [8] are becoming de-facto standards in the commercial sector. Many e-science workflow tools including Kepler [6], Taverna [29], and Triana [30] use Web Services as components. This growing pervasiveness of Web Service components in scientific applications is drawing attention to the need for interoperability between the tools used to manage workflows. Unfortunately, the Web Service specification stack is growing at a rate that makes it difficult for some workflow tools to use the latest Web Service concepts. In this paper we explore this problem and describe several approaches to simplifying the burden on the workflow systems.

The Web Services specifications could be grouped in to two categories. The basic specifications, SOAP [13], WSDL [16] and UDDI [17], define underline infrastructure for Web Services. The second category consists of so called Web Service Extensions (WS-Extensions), which define add-on services like messaging, security, and transactions. For the sake of clarity we label Web Services based on first category as *simple Web Services* and Web Services based on both categories as *complex Web Services*. Even though research on scientific workflows has focused on support for the first category of specifications, support for the second category is limited.

Clearly, certain features such as security and messaging are essential for most real world workflows. However owing to lack of support for Web Service Extensions, complex use cases have been implemented using non-standard ad-hoc approaches. The defining characteristic of those approaches are the existence of mutual understanding between service providers and workflows composers. Even though they solved the problem at hand, the composer and the services become tightly coupled, and they are not interoperable when complex Web Services are involved.

In this paper we present our effort to realize use cases form the NSF Linked Environment for Atmospheric Discovery (LEAD) Project [20] using a variety of different tools, while preserving a loosely coupled link between composer and services. Our goal is to allow the user to pick any workflow composer and enactor they feel most comfortable with, without having to deploy a unique set of services for each tool. We present over experiences and observations, and discuss their applicability in enabling WS-Extensions for scientific Workflows.

The Remainder of the paper is organized as follows. In the next section we present a discussion on orchestration of complex Web Services. Sections 3 discuss realizations of LEAD use cases, and next two sections (4,5) discuss implementation of use cases in two workflow composers, Kepler and Taverna. Section 6 presents related work and section 7 concludes the discussion with conclusions and future work.

2. Web Service and Orchestration

Simple Web Services support in scientific workflows has been explored thoroughly by previous research [6], [29], [30], [19]. Among scientific workflow composers Kepler, Taverna, Triana [30] and Pegasus [19] provide support for generic Web Services as well as their respective application domain specific Services. Enabling Web Services by providing a generic Web Service actor is the universal approach used by those efforts. Even though the implementation details very, each provides a generic Web Service actor or equivalent component that handles Web Services. These components can be configured by providing a WSDL description with related meta-data and the component become a proxy for the service represented by the given WSDL description. The resulting model supports any simple Web Service and scientists can orchestrate a Web Service with limited knowledge about underline technologies.

We will discuss the generic Web Service actor in greater detail in the sections 3, 4, and 5 of the paper. In this section we present Web Service orchestration in general. Even though we do not claim a comprehensive discussion on the subject, we will present few important observations and discuss specific detail regarding WS-Addressing [11] and WS-Security [26] (authentication and authorization).

To simplify the discussion we break Web Service orchestration in to two topics, orchestration of Simple Web Services and Complex Web Services. Typically a workflow is represented by a graph of independent components and a workflow execution is a conditional traversal on the graph. Each component accepts a set of inputs and produces set of outputs. Workflow composers define an abstract representation for components of workflow and provide specific component implementations for specific tasks. Bowers and Ludascher [10] provide a detailed discussion related to component base scientific workflow modeling. From this point onward we use their term *Actor* to mean a component that act as a building block for composition. Actors may provide local as well as distributed tasks. In the distributed category Web Service based Actors are important because they allow services from remote systems to be present in a one workflow.

2.1 Orchestrating Simple Web Services

The Web Services standards define WSDL, a machine readable and interoperable Service description. Service providers write WSDL descriptions for their services and make them available to service requesters. Owing to the machine readable nature of descriptions there are tools which can simplify Web Service invocation to a great extent. Using this architecture it is possible to develop a generic Web Service actor that would accept a WSDL document together with information to locate a specific operation and configure itself to invoke a Web Service in scientist's behalf. Such an Actor acts as a proxy that accepts inputs, performs a Web Services invocation and produces outputs. Better still such actor could handle any Web Service, thus expanding the reach of scientists to the entire Web.

For an example let us consider a meteorologist who need to perform an experiment that can only be done at university A. Let us assume university A has provided a Web Service to perform the experiment and publishes a description of the Service. To use this service the meteorologist could add the generic Web Service Actor to his workflow, and configure the Actor by filling a pop up menu with the values published by university A. One of the values will be a WSDL URL and another will be an operation name. The configuration adds input and output ports to the actor according to the WSDL descriptions and, once the configuration is done, the Meteorologist may use it as any other actor. Once the workflow is executed, the Web Service actor is invoked by the workflow composer and in turn the actor does a real Web Service invocation and returns the results. With this architecture the distributed nature of the Actor's operation is opaque to the meteorologist.

Our example brings in to light a very important characteristic of scientific workflow orchestration. Even though the scientist is invoking Web Services, there is no need for him to understand the details of how it works. Scientist's composer can process the machine readable service descriptions and handles the complexities of Web Services in his behalf.

As far as simple Web Services are concerned, the primary challenge is the need for dynamic Web Service invocations at run time. This is because the WSDL description is made available only at the runtime and all the processing should be done dynamically. Typically Web Service middleware uses code generation to generate stubs from WSDL descriptions. Therefore Web Service actors often use code generation, runtime compilation, followed by dynamic invocation or Dynamic Invocation Interface (DII) of Web Services. The former method is used by Triana and latter is used by Kepler and Taverna.

2.2 Orchestrating Complex Web Services

With the techniques to orchestrate simple Web Services established, to understand their basic limitations and the need for complex Web Services, let us return to our meteorologist example.

Assuming the service provide by University A performs a huge simulation that takes two days to complete, this scenario require asynchronous interactions to avoid HTTP connection timeouts. Asynchrony requires a service requester to provide a return address to the service provider, which will be used to send the response Message. This address can be transferred as the first argument of the request. However with this approach both parties are operating via a private agreement and we have lost interoperability at the Web Services level. Similarly University A can achieve security using a private protocol which works as long as both sides have a private agreement.

On the other hand Web Services can achieve both these capabilities and many others while maintaining interoperability. For an example the return address can be conveyed with WS-Addressing and security can be implemented using WS-Security. In General to preserve loosely coupled nature between the service provider and requester all the information should be shared using corresponding Web Service standards.

However, following a standard solves only part of the problem at hand. For an instance we did not discuss how both parties know that security is required. If that information is conveyed without using a Web Service standards that implies there is a private understanding, which is undesirable. This can be avoided using WS-Policy [12]. University A could use WS-Policy assertions inside the Service's WSDL description to convey that the security is needed and the nature of the expected request. The properly configured composer, without intervention from the scientist, would conclude from WSDL description that the service provider expects security and adds security information to the message.

To understand how Web Service architecture can be used in scientific workflows, we should look at the Web Service extensions in detail. Based on functionalities provided by each Web Service extensions we observe three categories of Web Service extensions, service description, non-functional and functional extensions.

Two extensions to Service descriptions, WS-Policy and WS-Metadata Exchange [9] define the mechanism by which both parties can express and negotiate the policies that govern the interactions.

Non-functional extensions like WS-Reliable Messaging [22], WS-Transaction [18], WS-Security, WS-SecureConversation [23] and WS-Trust [24] provide add-on services to the communication channel between Service requester and provider. They address a property of the communication channel and, when middleware on the both side can handle extensions, neither requester nor provider need to be aware of the fact that the extensions are enabled.

Functional extensions like WSRF [31] provide features for the user. WSRF provide support for stateful Web Services and it is a feature for the Web Service user and not a property of the channel. As a result specifications at this level can not be made transparent to the user.

The difference between second and third set of specifications is crucial as they need to be handled differently within the workflow composer. In the most abstract view, handling a complex Web Services can be broken down to following tasks.

1. Administrator of the scientist's computing environment will configure the environment with the suitable parameters. This is a one time process and the need for knowledgeable person can be justified.

2. Scientist will configure the Web Service actor by providing WSDL, service, port and an operation.

3. From WS-policy descriptions embedded in the WSDL document the Web Service actor deduces features that should be enabled in order to invoke the Web Service.

4. If the feature is a non-functional extension, it will be handled transparently using the configuration from the scientist's environment.

5. If the feature is a functional extension, it becomes part of the users programming model (e.g. Stateful nature when WSRF is available). The composer must make it explicit to the user and give user a opportunity to fine tune it.

6. The actor invokes the Web Service and gets the results.

7. The actor provides the results back to the workflow composer.

Web Service middleware supports WS-Extensions as modules that can be associated with a given Service. By implementing a Web Service actor that can bind and configure those modules we could provide support for those extensions. We recognize two challenges that should be addressed to enable a given Web Service Extension.

1. Define the behavior of the extension inside a scientific workflow.

2. Configure the WS-Extension and enforce it in service invocation

Firstly the behavior, or how extension can be used inside a scientific workflow, can be usually derived from use cases in Web Services. However in some instances scientific workflows may impose special considerations.

Enabling the WS-Policy and WS-Metadata Exchange means the actor can configure other extensions according to the policies of the service provider. The policies can be specified in the WSDL description, WS-Addressing Endpoint Reference, or negotiated at the invocation time using WS-Metadata Exchange. The actor should process the policies and decide on the policy alternative for the current interaction and configure other extensions accordingly.

Web Services Security consists of three main specifications WS-Security, WS-SecureConversation and WS-Trust. The security can be enabled by enabling and configuring the WS-Security implementation based on the Service policy description. Depend on the policy description the actor may need to contact WS-SecureConversation or WS-Trust endpoints to obtain derived keys or security tokens that are used to configure WS-Security implementation. The security policy descriptions are defined by the WS-Security Policy Specification [27].

To provide reliable messaging support the actor should enable and configure a reliable messaging implementation based on the policy description. The configuration of the module is available from the policy description and, in their absence, default values can be used.

Secondly it is duty of the Web Service actor to configure WS-Extensions and enforce them in service invocations. The configuration of extensions requires different information and the most pragmatic solution to this problem is to load the information form the environment of the scientist. Some of these, such as identity as defined by X.509 credentials, are already available as part of the scientist's environment, and for others we have to define representations for that information in the environment. The problem of locating this information should be discussed in case by case basis.

Typically to store a configured actor, workflow composer need only to store the end point URL of the service and the operation the actor represents. Consequently to store a workflow, a composer stores the actors in the workflow and their interactions. Web Service middleware is configured dynamically when the workflow is loaded and as a result if the service policies are changed after the workflow is composed, the composer can apply the changes by reloading the workflow from the composer, which would re-configure Web Service middleware according to the new service polices.

As far as our work is concerned we have chosen asynchronous messaging, authentication and authorization as the extensions we need to support based on LEAD use cases and left definition of other extensions as future work. The next section presents implemented extensions, WS-Addressing and WS-Security in detail.

2.3 Synchronous and Asynchronous interactions

The following discussion is based on the assumption that WS-Addressing is supported by the service provider. According to the Web Service architecture extensions like security can be optional and a service provider may choose not to support it. However Addressing holds together most of the other specifications, therefore we do not believe that it is practical to implement support for complex Web Services without addressing.

According to the Web Service architecture, the nature of an interaction is decided by service requester. We can visualize this scenario using a lawyer who provides legal advice though the phone. The seeker of advice may wait on the phone for a response or provide a number to call back when the response is available, and usually the lawyer supports both protocols. Similarly, a service requester can uses WS-Addressing to inform the server that he is waiting for the response or to instruct the server to send the response to a given location.

The workflow composers always play the role of service requester and, as a result, they are free to decide on the nature of the interaction. This decision shall be based on a number of factors. For an example, asynchronous invocations are usually used with long running services. If most of services in a workflow are long running, the workflow composer may treat every service as asynchronous. Otherwise the decision can be left to the user as a configuration parameter.

Synchronous and asynchronous behavior is enforced using the WS-Addressing parameters Reply-To and Fault-To. If the Reply-To value is absent (or an anonymous URL), the response is sent back through the return path of the request transport connection, or else the response will be sent to the Reply-To URL. In case of an error, the error message should be sent to the Fault-To property. Typically a service requester initiating an asynchronous interaction starts a receiver for incoming SOAP messages and directs the Reply-To and Fault-To properties to that receiver.

In a synchronous interaction the response and request messages are correlated by fact that they travel in the same transport connection. However message correlation in a asynchronous interaction requires additional information, and the WS-Addressing properties Message-ID and Relates-To provide that information.

2.4 Security

When the security Extension is enabled, the composer should provide authentication, authorization and confidentiality based on policies of the service provider. WS-Security defines basic constructs to achieve them with Web Services.

The Web Service security mechanisms are based on general security use cases and operate with the XML based Web Service representations of the usual security entities. As a result there is a one to one correspondence between parameters in WS-Security and parameters in the security model of the scientist's environment. Using this correspondence it is straightforward to parameterize the WS-Security module in a workflow composer using the scientist's environment.

For an example GRID environment, where most of the scientific workflows will be running, provides a standard way to find security credentials from scientist's environment. Those security credentials provide sufficient information to configure WS-Security to operate within scientist's workflow composer.

For authentication, messages are signed with user's private key and service provider can verify the authenticity using user's public key. Similarly both the authorization and confidentiality can work with the information included in GRID credentials. There are additional parameters to be shared, such as the supported security algorithms and the parts of a SOAP message that need to signed or encrypted. Technically that information should be shared through WS-Policy but in absence of policy information, it can be provided as part of the configuration. The composer could pick default values while user can override them in need. On the whole typical scientist's environment contains sufficient information to parameterize WS-Security to operate in workflow composer.

3. Implementing Enhancements

As far as simple Web Services are concerned, interactions between composer and Web Service middleware are limited to converting input and output parameters from one domain to the other. But once the complex Web Services enter the picture, a substantial amount of configuration information should be transferred between domains and their mappings are no longer straight forward. The security credential transfer from the user's environment to the Web Service middleware framework is a good example of such complex mapping.

Our architecture introduces mediation tools that manage information flow and communication between two domains. The Web Services side is implemented using Axis2 [4] considering support for WS-Security, WS-Addressing and ongoing efforts to support other WS-Extension. We believe our architecture can be extended to capture other WS-Extensions as they become available. We have extended two widely used workflow composers, Kepler and Taverna to orchestrate complex Web Service workflows that uses asynchronous interactions, authentication and authorization. We have made use of extensibility of each composer to develop generic Web Service Actors for each.

Implementation of the generic Web Service actor could be broken down to two parts, the mediation tools and the composer specific actor implementation. The rest of this section will discuss the mediation tools and the composer specific details are discussed in sections 4, 5.

3.1 LEAD Usecases

Before we present the architecture, let us discuss a typical LEAD use case. LEAD services operate in GRID environment and wrap scientific tools as Web Services. The LEAD environment provides special services, and among them a Capability Manager Service and a Message Box service are used by the scenario we present.

In a typical use case, a scientist will log in to the Web Portal provided by LEAD, and search for services in the LEAD Resource catalog. He could identify services of interest and obtain WSDL description for each service. The scientist will configure a generic Web Service Actor per each service and compose a workflow using the actors.

When the workflow is executed, the composer should obtain capability tokens from the capability manager and include then in the request Message. The capability tokens are signed by the service provider and they authorize users to use services. Most of the services have long execution times and the asynchronous Messaging is used. As we wrap scientific workflows we only need support for simple type parameters in services orchestrated in the composer.

3.2 Generic Web Service Actor

LEAD use cases are supported by implementing a Web Service Actor that support asynchronous interactions, authentication and authorization. The execution and communication between the actors are managed by the composer they reside in. The heart of each actor is above mentioned Mediation tools, which perform mediation between composer and Web Services middleware.

This section presents basic concepts behind the actor and composer specific details will be presented in sections 4 and 5. The Web Service Actor is initialized at composer start up and become available through actor libraries of the composer. While composing a workflow, the scientist adds one or more actors to the workflow, and each addition will create a new actor instance. Each Actor is configured by providing a WSDL URL and an operation name, where they are handed over to mediation tools which configure both the actor and Web Service middleware.

A Web Service actor in a workflow is directly mapped to a WSDL operation binding. The input and output messages of the WSDL operation correspond to input and output ports of the Actor. The Actor expands first level of the XML element referenced by each message, and uses first level children to derive input and out put ports. Our present implementation only supports simple types as parameters.

The configured Actor can be used in workflow composition. Once executed the actor performs a Web Service invocation using the inputs made available via input ports of the Actor. The Actor waits for results of the invocation and broadcast the results to the output ports.

3.3 Asynchrony

The Actor can be configured to do both synchronous and asynchronous invocations. However since the LEAD services are typically long running by nature, the actor is configured for asynchronous mode by default. We use WS-Addressing to convey addressing and correlations information between two parties.

Asynchronous mode requires a listener inside the composer to receive messages. We could have used the Axis2 listener that automatically starts when Axis2 asynchronous mode is enabled. But that listener is managed by Axis2, and in our scenario this could cause a number of listeners to be started and shut down during the lifetime of a one workflow.

To remedy this, we use the Message Box service provided by LEAD environment. The Message Box is similar to a typical postal box, where each user can create his own personal Message Box and all the messages that are received by the Message Box are stored and reproduced when the owner of the Message Box wants to collect messages.

When a workflow composer is started the composer creates a Message Box which is a shared among all Actor instances. The Message box creates a new Message Puller, a separate thread that keeps polling the Message Box for new Messages. We use one-way API of Axis2 for Web Service invocation. With the receiving mechanism in place, asynchronous interaction is achieved using following steps.

1. Add a Message-ID property to the request Message.

2. Add address of the Message Box as the Reply-To and Fault-To properties of the request message.

3. Register a callback that includes a reference to the output ports of the current actor with the message puller. The Message-ID value is used as the key for the callback.

When a response Message is received, according to the WS-Addressing it must have a Relates-To property equal to the Message-ID of the request message. The Message Puller looks up the callback using the Relates-To value and invokes the callback providing response message as a parameter. The callback will parse the message and broadcast the results to the output ports.

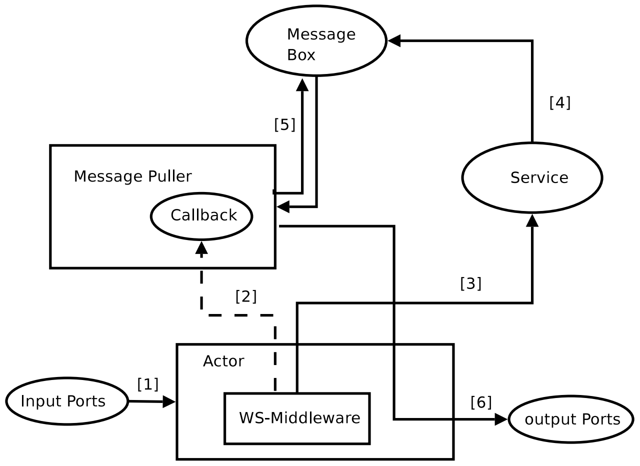


Figure 1. Mediation Architecture

Figure 1 shows the message path between different components of the model. The superscripts on the lines correspond to the bracketed numbers in the figure. When the Actor is executed ^{1}, it registers a callback with the message puller ^{2} and sends a SOAP message to the service ^{3}. According to WS-Addressing parameters service responds to the Message Box Service ^{4} and the response is stored in the Message Box. When message puller queries the Message box the next time it will collect the message ^{5} and invoke the callback which in turn send the results to the output channel of the workflow ^{6}.

3.4 Authentication and Authorization

Authentication and authorization is built on top of WS-Security and the XPOLA [21] Authorization infrastructure. Since the composer operates inside a GRID environment and GRID credentials are used to configure WS-Security.

Authentication is achieved using public key infrastructure. The Composer signs the SOAP messages with the private key of the scientist who uses the workflow composer and service provider verifies the signature of the SOAP message and establishes the authenticity of the Service Requester (The workflow composer). The WS-Security standard enforces the standards and ensures interoperability.

Authorization is implemented using XPOLA authorization framework. The scenario includes four parties, and the trust is represented with SAML capability tokens issued by service provider. As shown by the Figure 2, for each le-

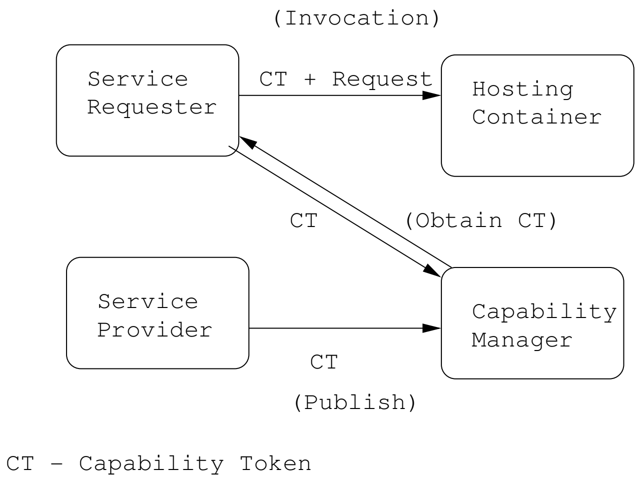


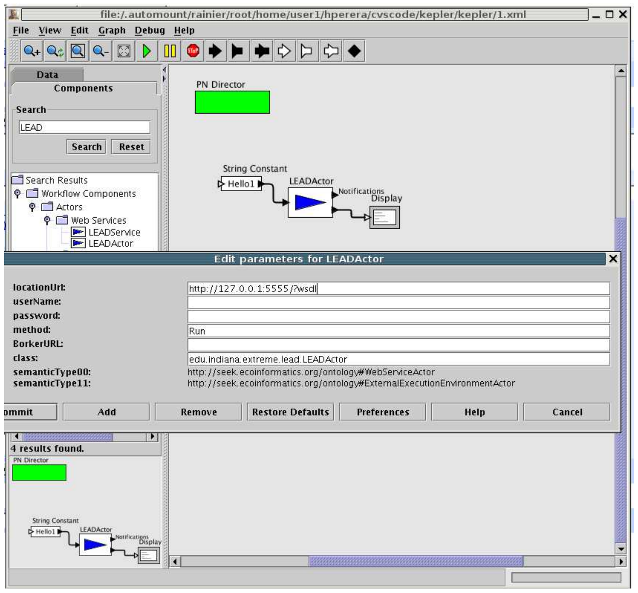
Figure 2. Authorization Model

gitimate user, service provider publishes a capability token to capability manager service provided by XPOLA. The capability manager service is a registry of capability tokens for each user. A Capability token includes the distinguished name of the authorized scientist and the operations he is allowed to access and is signed by the service provider's private key to establish the trust.

For each request, workflow composer signs the message with scientist's private key, obtains his capability tokens from capability manager and adds the capability token as a security Header. The Service's container (which may or may not be the service provider itself) compares the identity given in the capability token with the identity of the Service Requester (obtained from the request signature) and if both are equal authorizes the Service requester.

Our security implementation is greatly eased by the GRID Security infrastructure (GSI) [28]. We use Cog JGlobus [1] Toolkit to extract Globus credentials from scientist's environment and the Apache WS-Security implementation, WSS4J [5] is used to provide WS-Security support for Web Service middleware. WSS4J obtain the credentials from a Java key store by default and can not be used out of the box. We use the WSS4J extension mechanisms to override the configuration to load the credentials from the GRID environment. We enable WSS4J by adding WSS4J based security module to Axis2. Using WSS4J the actor signs the request message, obtain the capability tokens and add them as a security header.

To summarize, we have enabled asynchrony, authentication and authorization support for the workflow composer while preserving a loose coupling between the Services and composer. The service provider and composer need not have any understanding beyond the principles of Web Services standards conformance. We use standards WSDL and


Edit parameters for LEADActor

locationUrl:	http://127.0.0.1:5555/?wsdl
userName:	
password:	
method:	Run
BorkerURL:	
class:	edu.indiana.extreme.lead.LEADActor
semanticType00:	http://seek.ecoinformatics.org/ontology#WebServiceActor
semanticType11:	http://seek.ecoinformatics.org/ontology#ExternalExecutionEnvironmentActor

Figure 3. Kepler LEAD Actor

SOAP for Web Service support, WS-Addressing to impose asynchronous interactions, WS-Security for message signatures and SAML [3] for capability tokens.

4. Extending Kepler

Kepler is an extension to Ptolemy II project [2] which was developed initially for signal processing. Ptolemy is a Java based component assembly framework builds on top of generic components called actors. Ptolemy defines an actor, the input and output ports, their life cycles and interactions. The actors can operate in different execution domains and executions behave accordingly. Kepler includes actors for RPC Web Services and message base Web Services. However they can not be used out of the box to orchestrate LEAD services as the latter depend on Web Services Extensions. To circumvent this issue we extended Kepler by introducing a new generic Web Service actor, which we named LEAD Actor.

The LEAD actor is an enhanced generic Web Service Actor that acts as a proxy for any Web Service. We develop our actor for the Process Networks (PN) execution domain [14], where actors are executed in parallel threads and are able to responds to events. We do not present the implementation details about the LEAD actor, but Hylands et al [25] provide comprehensive details about how to extend Ptolemy using actors.

User who wants to compose a workflow that includes a LEAD Actor can search actor libraries and add the actor to the workflow by dragging and dropping the actor to the workflow. This process will create a new instance of the LEAD actor and add it to the workflow. This initial actor does not have any input or our put ports, and can be configured though the menu available by right clicking the actor. User should provide WSDL URL, operation name, and other optional parameters. Once the information is provided the actor configures itself by adding input and output ports according to the WSDL description. Thus configured, the actor can be used to compose workflows.

A workflow may have more than one instance of the LEAD Actor. The composition is done by connecting the input and output components of the various actors with each other. When the workflow is executed each actor is started in a separate thread and each blocks their execution while reading inputs from the input ports. Once the inputs are available, each LEAD actor will use the mediation tools described above to construct a SOAP message and invoke the Web Services. When the results are available, they are processed and broadcast to the output ports.

Our actor depends only on the services provided by Ptolemy and do not have any adverse effort on the architecture of Kepler. Web Service invocations use the streaming capabilities of Axis2 and therefore we expect a slight performance improvement on simple Web Service invocations. On the other hand enabling the extensions, specifically Security has introduced overheads. But with WS-Security it is unavoidable and we have a trade off between functionality and performance. However, considering the long running nature of LEAD use cases, the impact from the overhead is minimal.

5. Extending Taverna

Taverna workflow consists of components called Processors. A Processor consists of a name, a set of inputs, a set of outputs. Its function is defined as a transformation between the inputs and outputs. The Scufl workbench [29] provides a view for composition and execution of the Processors.

Taverna has a WSDL Processor which is a generic Web Service Processor implementation based on Apache Axis. In order to extend the composer for Web Service extensions we develop a new Generic Web Service Processor based on Axis2. As we discussed above, both Kepler and Taverna share the Mediation tools that provide a generic Web Service interface for workflow composers to handle the complexities of both Web Services and Web Service extensions.

The new processor can be installed by adding it in taverna.properties file and it will add new scavenger to the Taverna ”Available Services Tab”, which act as a factory for our Processor. When the scientist right click on the Available processors node of the ”Available Services” window, the menu shown by the diagram 4 will appear. To orchestrate a LEAD Web Service the user should select the ”Add

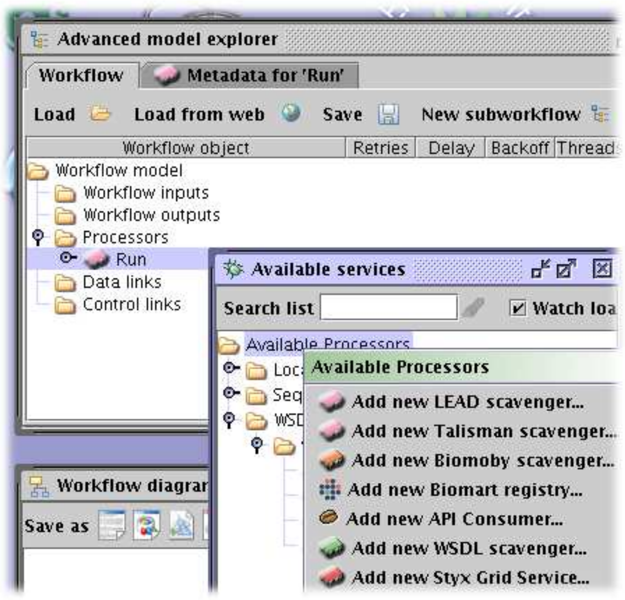


Figure 4. Taverna with LEAD Processor Added

new LEAD scavenger” option. This will create a scavenger which will fetch the WSDL and create a processor for each operation described in the WSDL file. Each Processor will have an input output ports as described by the corresponding WSDL Operation. The composition is done by adding processor(s) corresponds to the operations of interests to the Advanced Model Explorer and connecting them with data and control links.

When workflow is executed and execution reaches a LEAD Processor, an associated invocation task (LEAD execution Task) is called. The Execution task uses the Mediation tools to construct request SOAP message and invoke Web Services. The response to the Web Service invocation is processed and broadcast to the output ports.

As far as the Taverna is concerned the LEAD Processor is yet another processor, and enjoys the availability of all services available to a Processor inside Taverna. As it is based on the Taverna extension mechanisms, it does not have any adverse effect on the Taverna architecture. The Performance of the new processor is expected to have the same concerns as the Kepler actor.

6. Related Work

There are number of workflow systems orchestrating scientific workflows [32] and Grid workflows [33]. In section 2 we discussed the generic Web Service orchestration support provided by scientific workflow composers like Kepler, Taverna. To best of our knowledge none of them provide WS-Addressing and WS-Security support for generic Web Services.

Business workflow composition is an important area of related work to the topic. Charfi and Mezini [15] provide a discussion on integrating WS-Security and related WS-Policy consideration to Business workflow. In contrast to the actor based approach, where we handle WS-Extensions at the actor level, their implementation is more tightly integrated with BPEL container.

In the area of GRID workflows, integrating GSI security infrastructure with the GRID service orchestration is of high importance. Amnuaykanjanasin and Nupairoj [7] discuss an approach to handle secure GRID services with BPEL. Their approach generated a security enabled proxy service for each secure Web Service and BPEL engine orchestrate the proxy service instead of the original service, which would in turn do the real invocation. The Actor based approach and this have some similarity, especially if we think about the actor as a proxy. However explicit proxy service generation could add overhead and complexity.

In contrast to the above stated approaches, in this paper we discuss an actor based approach that uses architecture and the patterns provided by Web Services to handle additional considerations (e.g.security,asynchrony) in scientific workflow orchestration, and make an effort to make them transparent to the scientist.

7. Conclusions and Future work

In this paper, we have discussed enabling Web Service extensions in scientific workflows. We discussed the relationship between orchestration and Web Service Extensions and the need to address two basic concerns with respected to each extension. First, how to define the behavior of each extension inside scientific workflows and second, how to parameterize the Web Service Extensions based on the workflow environment. We discussed mediation tools that bridge the gap between current Web Service middleware and workflows, and we discussed the architectural details of mediation tools.

Finally we discussed extending two popular scientific workflow composers to enable support for orchestration with asynchrony, authentication and authorization. We believe that our effort to extend existing composers is a proof of presented concepts. Furthermore by building on top of proven implementations, we have make it easier to use them in real world applications.

As future work we plan to expand our work to the other Web Services Extensions as middleware for them becomes available. Among them support for confidentiality in WS-Security and support for WS-Policy are of highest interest. Also WS-Reliable Messaging and WS-Transactions are another interesting and challenging directions. Enabling WS-Policy will be an important accomplishment as that will provide fundamental constructs to include information about WS-Extensions with WSDL description, thus detaching the user from the complexities of WS-Extensions.

Finally we want to restate our motivation to enable the WS-Extensions for scientific workflows while preserving standard conformance and simplicity of the composer operation, thus making services and the composers interoperable on next level. We believe this paper laid ground work for truly interoperable Web Services and workflow tools for scientific applications.

References

[1] Java COG kit. http://www-unix.globus.org/cog/java/.

[2] Ptolemy II. http://ptolemy.berkeley.edu/ptolemyII/.

[3] Security Assertion Markup Language (SAML) v1.1. http://www.oasis-open.org/specs/index.php#samlv1.1.

[4] Web Services - Axis2. http://ws.apache.org/axis2.

[5] Web Services - WSS4J. http://ws.apache.org/wss4j.

[6] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock. Kepler: An Extensible System for Design and Execution of Scientific Workflows, 2004.

[7] P. Amnuaykanjanasin and N. Nupairoj. The bpel orchestrating framework for secured grid services. In *ITCC (1)*, pages 348–353, 2005.

[8] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weerawarana. Business Process Execution Language for Web Services Version 1.1, May 2003. ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf.

[9] K. Ballinger, F. Curbera, J. Schlimmer, et al. Web Services Metadata Exchange (WS-MetadataExchange), September 2004.

[10] S. Bowers and B. Ludascher. Actor-Oriented Design of Scientific Workflows. *Lecture Notes in Computer Science*, 3716:369 – 384, November 2005.

[11] D. Box, E. Christensen, F. Curbera, D. F. andJ. Frey, M. Hadley, C. Kaler, D. Langworthy, F.Leymann, B. Lovering, S. Lucco, S. M. andN.Mukhi, M. Nottingham, D. Orchard, J. S. andE. Sindambiwe, T. Storey, S. Weerawarana, and S. Winkler. Web Services Addressing (WS-Addressing),W3C Member Submission, August.

[12] D. Box, F. Curbera, D. Langworty, A. Nadalin, N. Nagaratnam, M. Nottingham, C. von Riegen, , and J. Shewchuk. Web Services Policy Framework (WS-Policy Framework), 2002.

[13] D. Box, D. Ehnebuske, G. Kakivaya, A. Layman, N. Mendelsohn, H. F. Nielsen, S. Thatte, and D.Winer. Simple Object Access Protocol (SOAP)1.1, May 2000. http://www.w3.org/TR/SOAP.

[14] C. Brooks and E. A. Lee and X. Liu and S. Neuendorffer and Y. Zhao and H. Zheng. *Heterogeneous Concurrent Modeling and Design in Java (Volume 3: Ptolemy II Domains)*. Technical Memorandum UCB/ERL, M04/17, University of California, Berkeley, CA USA 94720, 2004. Chapter 8.

[15] A. Charfi and M. Mezini. Using aspects for security engineering of web service compositions. In *ICWS*, pages 59–66, 2005.

[16] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL), Version 1.1, March 2000. http://www.w3.org/TR/wsdl.

[17] L. Clement, A. Hately, C. von Riegen, and T. Rogers. UDDI Spec Technical Committee Draft, October 2004. http://uddi.org/pubs/uddi-v3.0.2-20041019.htm.

[18] F. Curbera et al. Web Services Atomic Transaction (WS-AtomicTransaction), Version 1.0, August 2005.

[19] E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M.-H. Su, K. Vahi, and M. Livny. Pegasus: Mapping Scientific Workflows onto the Grid. *Lecture Notes in Computer Science*, 3165:11 – 20, Jan 2004.

[20] K. K. Droegemeier and others. Linked environments for atmospheric discovery (LEAD): A cyberinfrastructure for mesoscale meteorology research and education. In *20th Conf. on Interactive Info. Processing Systems for Meteorology, Oceanography, and Hydrology*, January 2004.

[21] L. Fang, D. Gannon, and F. Siebenlist. XPOLA: An Extensible Capability-Based Authorization Infrastructure for Grids. In *Proceedings of the 4th Annual PKI R&D Workshop: Multiple Paths to Trust*, 2005.

[22] C. Ferris et al. Web Services Reliable Messaging Protocol, (WS-ReliableMessaging), February 2005.

[23] M. Gudgin, A. Nadalin, et al. Web Services Secure Conversation Language (WS-SecureConversation), 2005.

[24] M. Gudgin, A. Nadalin, et al. Web Services Trust Language (WS-Trust), February 2005.

[25] C. Hylands, E. A. Lee, J. Liu, X. Liu, S. Neuendorffer, Y. Xiong, and H. Zheng. *Heterogeneous Concurrent Modeling and Design in Java (Volume 1: Introduction to Ptolemy II)*. Technical Memorandum UCB/ERL, M04/17, University of California, Berkeley, CA USA 94720, 2004. Chapter 8.

[26] C. Kaler et al. Web Services Security Specification(WS-Security), April 2002.

[27] C. Kaler, A. Nadalin, et al. Web Services Security Policy Language (WS-SecurityPolicy), July 2005.

[28] N. V. Kanaskar, U. Topaloglu, and C. Bayrak. Globus security model for grid environment. *SIGSOFT Softw. Eng. Notes*, 30(6):1–9, 2005.

[29] T. Oinn, M. Addis, J. Ferris, D. Marvin, M. Senger, M. Greenwood, T. Carver, K. Glover, M. R. Pocock, A. Wipat, and P. Li. Taverna: a tool for the composition and enactment of bioinformatics workflows. *Bioinformatics*, 20(17):3045–3054, 2004.

[30] Shalil Majithia and Matthew Shields and Ian Taylor and Ian Wang. Triana: A Graphical Web Service Composition and Execution Toolkit. In *IEEE International Conference on Web Services (ICWS'04)*, 2004.

[31] D. Snelling, I. Robinson, T. Banks, et al. OASIS Web Services Resource Framework (WSRF). http://www.oasis-open.org/committees/wsrf/.

[32] J. Yu and R. Buyya. A taxonomy of scientific workflow systems for grid computing. *SIGMOD Rec.*, 34(3):44–49, 2005.

[33] J. Yu and R. Buyya. A taxonomy of workflow management systems for grid computing, 2005.