Conceptual Overview of the Implementation of Web Services Specifications in FIRMS

Community Grids Laboratory, Indiana University, USA

In WS specifications the logic contained within the specifications and the corresponding protocol is achieved through the exchange of SOAP messages. The logic pertaining to the specification can be encapsulated in the header and/or the body of the SOAP messages. Individual SOAP messages are self-descriptive.

1. Function performed by Software

The FIRMS implementation will provide Grid and Web Service applications with ability to interact reliably with each other based on the implementation of the WS-ReliableMessaging (WSRM) and the WS-Reliability (WSR) specification.

2. Design of the WsProcessor

In our approach to implementing the Web Service specifications we considered SOAP messages as the focal point. All implementations of the WS specifications (in FIRMS and FINS) extend the WsProcessor class that is part of the **`cgl.narada.wsinfra`** package. This WsProcessor contains just one method viz. **`processExchange (SOAPContext, direction)`** where **`SOAPContext`** simply encapsulates the SOAPMessage. Processors (either WSRM or WSR) process these SOAP messages based on the processing outlined in the respective specifications. Furthermore, in instances where these specifications leverage other WS specifications, such as WS-Addressing, the rules outlined in those specifications are also enforced. For example, the construction of SOAP Envelope responses is based on the WS-Addressing message information headers (MIH) and also on the rules governing the mapping of WS-Addressing Endpoint References (EPR). Note that these processors provide complete implementations of the specification, i.e. they generate the appropriate responses, actions or faults in response to SOAP messages that have been received.

Included below is the definition of the **`processExchange()`** method. Using the **`SOAPContext`** it is possible for an entity to retrieve the javax.xml.SOAPMessage or the equivalent EnvelopeDocument (from XMLBeans). The **`direction`** specifies whether the message was received over the network or from the application. The logic related to the processing of messages is different depending on whether the message was received from the application or from the network. Exceptions thrown by this method are all checked exceptions and can be trapped using the appropriate `try-catch` blocks. Examples of when these exceptions are thrown are described in the section outlining how the WsProcessors can be deployed within handlers.

```
public void processExchange(SOAPContext soapContext,
                            int direction)
                      throws UnknownExchangeException,
                             IncorrectExchangeException,
                             MessageFlowException,
                             ProcessingException
```

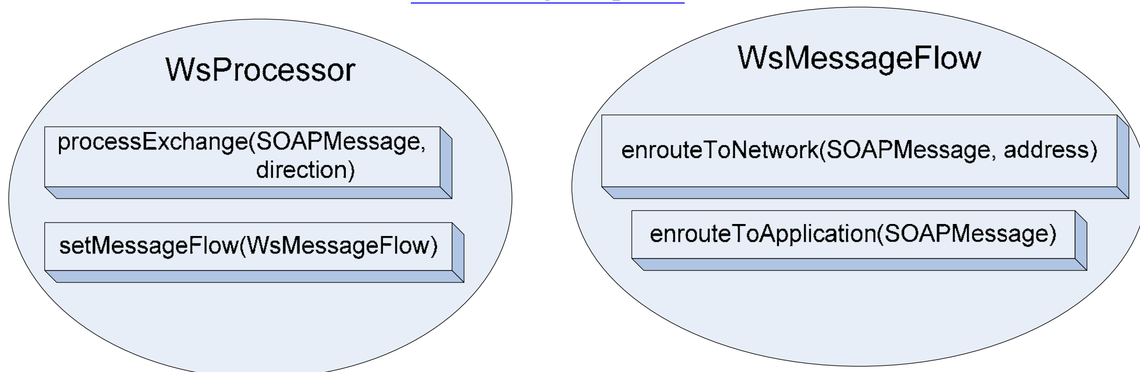

Figure 1: The WsProcessor and WsMessage flow components

Please note that in most WS specifications there are multiple roles. For example in WSRM the two distinct roles are that of a source and a sink. Similarly, in WS-Eventing the roles outlined include source (notification producer), subscription manager and sink (notification consumer). A processor decides on processing a SOAP based on three parameters

- The contents of the WS-Addressing `action` attribute contained within the SOAP Header.
- The presence of specific schema elements in either the `Body` or `Header` of the SOAP Message.
- If the message has been received from the application or if it was received over the network.

If the WS processor does not know how to process a certain message, it throws an **`UnknownMessageException`** an example of this scenario is a WS-Eventing source node receiving a WSRM CreateSequence response from over the network. An **`IncorrectExchangeException`** is thrown if the WsProcessor instance should not have received a specific exchange. For example if a WSRM sink node receives a wsrm:Acknowledgement it would throw this particular exception. Please note that these exceptions are NOT fatal errors, they simply indicate that the WsProcessor instance (whether it is a WsrmSourceNode, WsrmSinkNode, WseSourceProcessor, WseSinkProcessor or WseSubManager) cannot process the message in question. In most cases these WsProcessors will be cascaded together. If a user wishes to use both the WSRM Source and Sink capabilities the corresponding WsProcessor instances need to be instantiated.

`MessageFlowException` and **`ProcessingException`**s are errors caused due to problems with the networking and processing a message respectively. Typically, when these exceptions occur, unlike the previous exceptions, processing related to the message within the handler/filter chain needs to be terminated immediately. **`ProcessingException`**s occur due to processing errors related to inability to locate protocol elements in message, incorrect schemas and no values being supplied for some elements.

2.1 Role of the WsMessageFlow.

With regards to deployment, the other class of interest is the WsMessageFlow. This interface contains two methods **`enrouteToNetwork(SOAPMessage,address)`** & **`enrouteToApplication(SOAPMessage)`**. The WsProcessor uses these methods to route SOAP messages (requests, responses or faults) en route to applications and a network endpoint respectively. Since the WsProcessor delegates the actual transmission of messages to implementations of the WSMessageFlow, it can be deployed in a wide variety of settings, for example as a handler, proxy and within application programs. Note that, typically the job of transmitting messages is within the purview of the hosting environment viz. Axis, JWSDP etc.

2.2 Rationale for the choice of XMLBeans

While implementing these specifications we were faced with an important decision regarding the choice of tool to use in processing the XML schema that the aforementioned specifications conform to. In simple terms, we were looking for a system that allowed us to process the XML for these schemas from within the Java domain. There were three main choices. First, we could use the AXIS wsdl2java compiler. It is well-known that Axis' support for complex Schema Types is poor. In fact, the XML generated by the generated Java classes can sometimes not be compliant with the original schema specification. Support for the XML schema in Axis is both buggy and incomplete; the problems that we outlined for Axis also exist in Sun's JWSDP. We need true support for the XML Schema since at no point can we make the assumption that we will be dealing with Axis/JWSDP endpoints, which would conform to a subset of the XML schema.

The second approach was to use the JAXB specification (a specification from Sun to deal with XML and Java data-bindings). JAXB, though better than what is generated using Axis' *wsdl2java,* still does not provide complete support for the XML Schema. We looked at both the JAXB reference implementation from Sun and JaxMe from Apache (which is an open source implementation of JAXB). JAXB while significantly better than Axis' wsdl2java still does not provide complete support for the XML schema.

The final approach involves utilizing tools which focus on complete schema support. Here there were two candidates –- XMLBeans and Castor –- which provide good support for XML Schemas. We settled on XMLBeans because of two reasons. First, it is an open-source effort. Originally developed by BEA, it was contributed by BEA to the Apache Software Foundation. Second, in our opinion it provides the best and

most complete support for the XML schema of all the tools currently available. It allows us to validate instance documents and also facilitates simple but sophisticated navigation through XML documents. The XML generated by the corresponding Java classes is true XML which conforms to (and can be validated against) the original schema.

3. Specifications and schemas used

As most folks who have tracked WS specifications would agree, there are sometimes more than 1 version of the specification and the concomitant schemas. In some cases (such as SOAP) we were forced to use the older schema available at http://schemas.xmlsoap.org/soap/envelope/ since the javax.xml.SOAPMessage is based on this one. We do not however foresee problems in migrating to a newer version of the SOAP schema based on SOAP 1.2 should a need arise. We enumerate below the schemas corresponding to the various WS-Specifications and XML utilities that we used in our implementations.

FIRMS provide an implementation of the WSRM specification that was released in March 2004 and WSR specification that was released on August 2004. The WSRM protocol developed jointly by IBM, Microsoft and BEA can be found at
ftp://www6.software.ibm.com/software/developer/library/ws-reliablemessaging200403.pdf

WS-RM: http://schemas.xmlsoap.org/ws/2004/03/rm
SOAP: http://schemas.xmlsoap.org/soap/envelope/
WS-Addressing: http://schemas.xmlsoap.org/ws/2004/08/addressing
WS-Policy: http://schemas.xmlsoap.org/ws/2004/09/policy
XML utilities: http://schemas.xmlsoap.org/ws/2002/07/utility

The specification for WS-Reliability -- developed jointly by Fujitsu, Novell, Oracle, and Sun -- can be found at http://docs.oasis-open.org/wsrm/2004/06/WS-Reliability-CD1.086.pdf.
WS-Reliability: http://docs.oasis-open.org/wsrm/2004/06/ws-reliability-1.1.xsd
SOAP: http://schemas.xmlsoap.org/soap/envelope/
WS-Addressing: http://schemas.xmlsoap.org/ws/2004/08/addressing

4. Deployment and other Issues

This release includes two different deployments of the implementation of the WS specifications. The first deployment environment is the OMII Container. We have also included a prototype deployment container along with this release which is based on one-way SOAP messages and provide several capabilities. The prototype deployment container is based on Filter/Filter-Pipeline architecture. This prototype container addresses several problems currently present in the Axis environment (which the OMII Container leverages). Some of the features of this prototype container include

1. Dynamic configuration of the Filter Pipeline. Filters can be added, removed, updated or reorganized within the Filter-Pipeline.
2. Individual filters can configure themselves to be part of Input, Output or Input/Output processing.
3. Filters can inject messages either towards the application or towards the network. The Filter-Pipeline ensures that appropriate filters are traversed depending on the direction of traversal.
4. A filter can terminate processing related to a message within the Filter-Pipeline.
5. A filter can add, remove, modify or replace the contents of the SOAP Message or the equivalent EnvelopeDocument.
6. It is possible to configure different networking substrates with the Filter-Pipeline thus delegating the networking capabilities to it. Destinations are computed based on WS-Addressing's [wsa:To] element.
7. The filter model and its capabilities such as injecting messages, terminating processing and the ability to be cascaded provide a true representation of how WS specifications are intended to provide incremental addition of capabilities at an end-point.

The primary motivation for writing this prototype deployment container was to have a test environment that allows us to test every exchange as outlined in the implemented specifications and see if the intended

action corresponding to each exchange is taken and faults issued if necessary. The prototype deployment environment is based on one-way messaging between SOAP endpoints.

5. Package structure

The figure below provides a snapshot of the FIRMS distribution.