FINS User Guide

Community Grids Lab, Indiana University

This document is intended to provide users with information on developing applications based on WS-Eventing. This specification -- developed jointly by Microsoft, BEA, IBM and Sun -- can be found at http://www-128.ibm.com/developerworks/webservices/library/specification/ws-eventing/. This document first provides the reader with a brief overview of messaging and notification based systems. Readers who are familiar with these concepts might want to skip these introductory sections and proceed directly to the sections related to developing applications.

1. Brief Overview of Messaging Systems and Notifications

In messaging systems distributed entities communicate through the exchange of messages. These messages are self-describing and contain the semantic intent of the message. Depending on the application these messages can be made to encapsulate system conditions, method invocations, resource sharing, data interchange among others. Messages may also describe their correlation, dependency and causal relationships to other messages. Messaging based interactions engender greater modularity in the development of systems. For example, one could have separate classes or modules to deal with different types of messages.

Notification based systems are an instance of messaging-based systems where entities have two distinct roles viz. source and sink. A notification is a message encapsulating an *occurrence of interest* to the entities. There are two main entities involved in a notification: the source which is the generator of notifications and the sink which is interested in these notifications. A sink first needs to register its interest in a situation, this operation is generally referred to as a subscribe operation. The source first wraps occurrences into notification messages. Next, the source checks to see if the message satisfies the constraints specified in the previously registered subscriptions. If so, the source routes the message to the sink. This routing of the message from the source to the sink is referred to as a notification. The scenario is depicted in Figure 1.

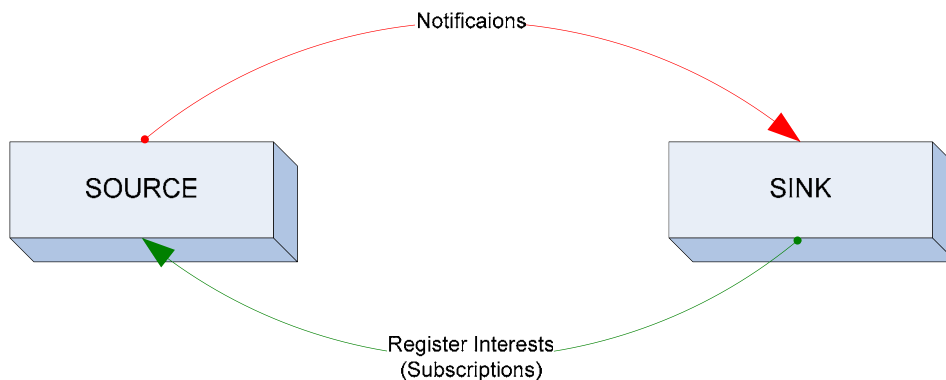

Figure 1: Example notification scenario

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

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

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

1.1 Brief overview of WS-Eventing

WS-Eventing is an instance of a tightly-coupled notification system. Here there is no intermediary between the source and sink. The source is responsible for the routing of notifications to the registered consumers. WS-Eventing, however introduces another entity — the subscription manager — within the system. This subscription manager is responsible for operations related to the management of subscriptions. Subscriptions within WS-Eventing have an identifier and expiration times associated with them. The identifier uniquely identifies a specific subscription, and is a UUID. The expiration time corresponds to the time after which the source will stop routing notifications corresponding to the expired subscription. Every source has a subscription manager associated with it. The specification does not either prescribe or prescribe the collocation of the source and the subscription manager on the same machine. The subscription manager performs the following operations

- It is responsible for enabling sinks retrieve the status of their subscriptions. These subscriptions are the ones that the sinks had previously registered with the source.
- It manages the renewals of the managed subscriptions.
- It is responsible for processing unsubscribe requests from the sinks.

Please note that sinks include their subscription identifiers in ALL their interactions with the subscription manager.

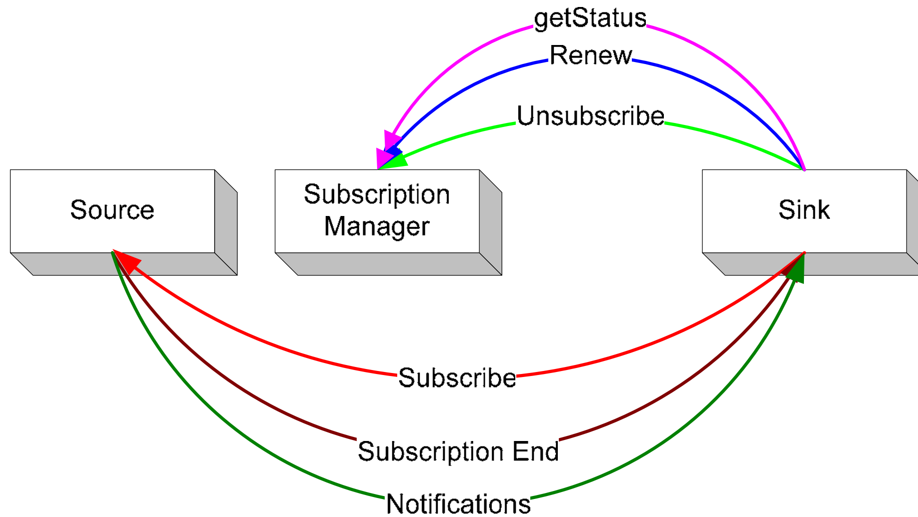

Figure 2: WS-Eventing - Chief components

Figure 2 depicts the chief components in WS-Eventing. When the sink subscribes with the source, the source includes information regarding the subscription manager in its response. Subsequent operations –- such as getting the status of, renewing and unsubscribing –- pertaining to previously registered subscriptions are all directed to the subscription manager. The source sends both notifications and a message signifying the end of registered subscriptions to the sink.

2. Building applications

The FINS software provides a complete implementation of the WS-Eventing specification. All functionality related to the various roles — source, sink and subscription manager – have been implemented. The subscription manager role is fixed and is not within the purview of application developers. Application developers will mostly be focused on the source and sink roles.

2.1 WS-Addressing and the Creation of Endpoint references

The WS-Eventing specification leverages WS-Addressing. WS-Addressing provides two very important constructs: endpoint references and message information headers. Endpoint references are a transport neutral way to identify and describe service instances and endpoints. The EPRs are constructed and specified in the SOAP message by the entity that is initiating the communications. EPRs are what facilitate support for dynamic usage patterns.

An EPR comprises the following

- An address element which is a URI
- A reference properties element which is a set of properties required to identify a resource
- A reference parameters element which are a set of parameters associated with the endpoint that are necessary for facilitating specific interactions.

The WS-Addressing specification has rules governing the construction of SOAP headers when trying to communicate with a service endpoint that has specified an EPR for communications. Basically, the message is targeted to the address element using the To header in WSA, the set of properties and parameters are moved directory as child elements of the SOAP header element.

We have simplified the creation of endpoint references. The complexity of the generation of the appropriate EPR is managed by the cgl.narada.wsinfra.wsa.WsaEprCreator class. To get a reference to this class please see the code snippet below.

```
WsaEprCreator wsaEprCreator = WsaEprCreator.getInstance();
```

A simple EPR reference can be easily created by the code snippet below.

```
String address = "http://www.other.example.com/OnStormWarning";
EndpointReferenceType eprType = wsaEprCreator.createEpr(address);
```

If one wishes to add ReferenceProperties to the EPR that we just created, the snippet below outlines how it is done.

```
QName qName = new QName("http://www.example.com/warnings",
                        "MySubscription");
wsaEprCreator.addElementToReferenceProperties(eprType, qName, "25");
```

It might also be useful for readers to be aware of elements with WS-Addressing. Since all exchanges leverage these elements heavily we have included a brief description of these elements for the reader's perusal. For additional details we suggest that the reader should take a closer look at the WS-Addressing specification. WS-Addressing also includes several message information

headers (hereafter MIH) which enable the identification and location of endpoints pertaining to an interaction (request, reply, and fault). The MIH elements comprise the following

To (mandatory element): This specifies the intended receiver of message. If there are EPRs contained in the SOAP header element, this identifies the node which would be responsible to route the message to final destination.

From: This identifies the originator of a message.

ReplyTo: Specifies where replies to a message will be sent to.

FaultTo: Specifies where faults as a results of processing the message should be sent to. If this element is not present, faults will be routed to the element identified in the replyTo element. If both the replyTo and faultTo elements are missing the faults are issued back to the source of the message.

Action: This is a URI that identifies the semantics associated with the message. WS-Addressing also specifies rules on the generation of Action elements from the WSDL definition of a service. In the WSDL case this is generally a combination of **[target namespace]/[port type name]/[input/output name]** . For e.g. http://schemas.xmlsoap.org/ws/2004/08/eventing/Subscribe is a valid action MIH element.

MessageId: This is typically a UUID which uniquely identifies a message. This is sometimes also used correlate previous messages. For e.g. in WSRM if you have requested the creation of sequence, the response to the creation of the sequence would include the messageId of the request in the relatesTo element.

RelatesTo: This identifies how a message relates to a previous message. This field typically contains the messageId of a previously issued message.

2.2 Dealing with the source side

The source is responsible for generating notifications. No restrictions have been imposed on the type or the content of these notifications.

Source functionality such as managing subscription requests from the subscriber, coping with the expiry of subscriptions, and dissemination of notifications to registered sinks are all accessible through the WsSourceProcessor class. Care must be taken to ensure that all incoming and outgoing messages from a node funnel through this class. Depending on the SOAPMessage (and the information encapsulate therein) and the whether it was received from the application or over the network, this processor deals with exchanges as outlined by the specification.

Before we proceed further, it might be useful to understand the functionality encapsulated within this class.

1. It manages subscription requests received over the network. It can also check this subscription request to see if it is well-formed and conforms to the constraints/rules within the WS-Eventing specification.
2. Matching Engines: This class automatically loads matching engines related to various subscription dialects. Matching engines related to XPath, String Topics, Regular Expression and XQuery are loaded by this class.

3. Management of disseminations: This class is responsible for ensuring the dissemination of notifications to the right entities.

By ensuring that all messages from the network and from the application are funneled through this class, the WS-Eventing source functionality available within this class is accessible to the application.

2.2.1 Generating Notifications

A source thus needs to be only concerned with its primary role — generation of notifications. The source-processor described in the earlier section deals with issues related to the matching of subscriptions and computation of sinks that the notification needs to be disseminated to. The source needs to generate appropriate SOAP messages.

The source-processor will inspect the received SOAP message and proceed to match it with the stored subscriptions. The subscription formats received from interested entities could be based on XPath, String topics, Regular Expressions or XQuery queries.

In the case of Topics, the source has to make sure that it adds Topic information to the published SOAPMessage. This is because unlike XPath, Regular expressions and XQuery which operate on the entire SOAP message, in the case of Topics the associated TopicMatching Engine is looking for the topic information to be encapsulated with a specified QName.

For example, if the Topic information that one wishes to add is "Literature/Shakespeare" all one needs to do is include the following code snippet:

```
String topicName = "/Literature/Shakespere";
QName qName = wseQNames.getTopicQNameNBX();
boolean added = soapMessageAlteration.addToSoapHeader(envelopeDocument,
                                                       qName, topicName);
```

Please note that a given SOAP message could contain information in it such that it can be matched against valid XPath, Topic, Regular Expression and XQuery subscriptions.

2.2.2 Adding your own matching engine

The WS-Eventing specification mandates support only for XPath subscriptions. We have included additional support for String Topics, Regular Expressions and XQuery based subscriptions. It is entirely possible that a given application may need to support additional subscription formats. We have built in a very easy extensibility mechanism into the system so that users can their own matching engines so that they can support additional filter/subscription dialects.

For example if one is interested in incorporating support for an SQL based subscription dialect. In this case, there is only one class to be implemented — SQLMatchingCapability — which extends a base class that all matching engines extend: cgl.narada.wsinfra.wse.matching.MatchingCapability.

Here, we need to implement only one method — performMatching (The signature of this method has been included for the reader's perusal — which is related to the actual matching operation.

```
public abstract boolean
  performMatching(EnvelopeDocument envelopeDocument, FilterType filter)
    throws ProcessingException;
```

Once, this method has been implemented the availability of this SQL matching engine needs to be made known to the source-processor which performs the matching operations for the SOAP messages received from the application. This can be done by leveraging the cgl.narada.wsinfra.wse.matching. MatchingCapabilityFactory class. The code snippet below demonstrates how this is done.

```
SQLMatchingCapability sqlMatchingCapability =
                                          new SQLMatchingCapability();
MatchingCapabilityFactory matchingCapabilityFactory =
MatchingCapabilityFactory.getInstance();
matchingCapabilityFactory
.registerMatchingCapability(sqlMatchingCapability)
```

2.3 Dealing with the sink side

Development of the application sink is a little more involved than on the source side. This is because the sink is responsible for the generation of several different request types. We however have a class — WseSinkProcessor — which encapsulates the sink's capabilities (as defined in the WS-Eventing specification) and simplifies the generation of requests. An application sink thus need not worry about the generation of well-formed requests since all processing related to this is handled by the SinkProcessor.

We now briefly enumerate the different types of requests that are issued by the sink

1. Subscribe : This register's a sink's interest with the source.
2. GetStatus: This allows a sink to check for the status of a previously registered subscription. Specifically, this allows a sink to know hen exactly its subscription is scheduled to expire.
3. Renew: This exchange allows a sink to renew previously registered subscriptions, so that they expire at a later duration.
4. Unsubscribe: This indicates that a sink is no longer interested in the receipt of notifications corresponding to a previously registered subscription.

The first step is to get a reference to the WseSinkProcessor. The code snippet below outlines how this is done.
WseSinkProcessor wseSinkProcessor = WseSinkProcessor.getInstance();

2.3.1 Creating a subscribe request

A valid subscribe request in WS-Eventing requires specifying several parameters. We enumerate these below

1. Source EPR: This is the EPR of the source that sink wishes to receive notifications from.

2. Sink EPR: This is the EPR of the source. This enables the source to route any notifications that match the subscription constraint specified by the source.
3. EndTo EPR: In case of problems at the source which might result in a source shutting down, a source is expected to send SubscriptionEnd notifications to the registered sink. This parameter enables a sink to register where exactly the aforementioned SubscriptionEnd notification should be sent.
4. Delivery Mode: A sink is also allowed to specify how it wishes these subscriptions to be routed to them. The source can either push notifications to the sink OR the sink can pull notifications from the source. Currently, WS-Eventing requires support ONLY for the push mode and this is what we have provided. The WS-Management API identifies 3 additional modes — pull, trap and batch — we have not yet decided if we will be implementing WS-Management. If we do so, we will include support for the aforementioned delivery modes.
5. Filter Dialect: This specifies the dialect in which your subscription query has been issued. All subscriptions are specified as Strings, however the identification of this String as XPath query or a regular expression query makes a big difference. The current spec requires support only for XPath. However, we have included support for several other dialects.
6. Filter Constraint: This is the actual constraint specified by the sink. All notifications must satisfy this constraint before being routed to the sink that registered the subscription.
7. Expires At: This specifies the Date and Time at which the subscription is set to expire.

The code snippet below demonstrates the use of the SinkProcessor to create the subscription request.

```
EndpointReferenceType sourceEpr = wsaEprCreator.createEpr(sourceAddress);
EndpointReferenceType sinkEpr = wsaEprCreator.createEpr(sinkAddress);
EndpointReferenceType endTo = wsaEprCreator.createEpr(sinkAddress);
String deliveryMode =
"http://schemas.xmlsoap.org/ws/2004/08/eventing/DeliveryModes/Push";

String xpathFilterDialect =
                  "http://www.w3.org/TR/1999/REC-xpath-19991116";
String xpathFilterConstraint = "declare namespace
c1='http://www.naradabrokering.org/catalog'" +
            "$this//c1:catalog/c1:cd/c1:price > 20";

Calendar expiresAt = Calendar.getInstance();
expiresAt.add(Calendar.MONTH, 1);

EnvelopeDocument envelopeDocument =
   wseSinkProcessor.createSubscribeRequest(sourceEpr, deliveryMode,
                                    endTo,
                                    filterDialect, filterConstraint,
                                    expiresAt);
```

The snippet below depicts the structure of the created Subscribe Request

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:add="http://schemas.xmlsoap.org/ws/2004/08/addressing"
xmlns:even="http://schemas. xmlsoap.org/ws/2004/08/eventing"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-ins tance">

<Header>
```

```
  <add:Action>http://schemas.xmlsoap.org/ws/2004/08/eventing/Subscribe</add:Action>
  <add:MessageID>5c6a1b7b-b495-4959-82aa-95fc366eac77</add:Message ID>
  <add:From>
    <add:Address>http://localhost:18080/axis/services/WseSink</add  :Address>
  </add:From>
  <add:To>http://localhost:18080/axis/services/WseSource</add:To>
</Header>

<Body>
  <even:Subscribe>
    <even:EndTo>
      <add:Address>http://localhost:18080/axis/services/WseSink</add:Address>
    </even:EndTo>
    <even:Delivery
Mode="http://schemas.xmlsoap.org/ws/2004/08/eventing/DeliveryModes/Push"/>
    <even:Expires xsi:type="xs:dateTime" xmlns:xs="http://www.w3.o
rg/2001/XMLSchema">2005-08- 08T13:03:44.100-05:00</even:Expires>
<even:Filter Dialect="http://www.w3.org/TR/1999/REC-xpath-19991116">declare namespace
c1='http://www.naradabrokering.org/catalog'$this//c1:catalog/c1:cd/c1:price >
20</even:Filter>
    <even:NotifyTo>
      <add:Address>http://localhost:18080/axis/services/WseSink</add:Address>
    </even:NotifyTo>
  </even:Subscribe>
</Body>

</Envelope>
```

2.3.2 Creation of a GetStatus Request

If a sink's request to subscribe to notifications from a source was successful, the source responds with a message confirming this success. This Subscribe response also includes a subscription identifier (a UUID) associated with the subscription request. This identifier is assigned by the source. In addition to this the source also includes information regarding the subscription manager associated with the source. Interactions pertaining to GetStatus, Renew, and Unsubscribe operations should be exchanged with the Subscription Manager contained within the response. Please note that ALL interactions with the subscription manager should also indicate the subscription identifier associated with the subscription.

Upon receipt of response associated with a successful subscription request, the sink processor gleans relevant information associated with the response. This includes the subscription manager associated with the request and the expiration time associated with the subscription.

In order to issue a GetStatus request all that a sink needs to specify is the subscription identifier associated with the subscription. The SinkProcessor constructs the appropriate GetStatus SOAP message targeted (through the wsa:To element) to the Subscription Manager managing this subscription. The code snippet below depicts how this is done.

```
String subscriptionIdentifier =
                          "2cd0faa7-3a53-46f4-adc1-f186381a94e4";
EnvelopeDocument envelopeDocument =
```

```
        wseSinkProcessor.createGetStatus(subscriptionIdentifier);
```

The snippet below depicts the structure of the created GetStatus Request

```
<Envelope xmlns=http://schemas.xmlsoap.org/soap/envelope/
   xmlns:add=http://schemas.xmlsoap.org/ws/2004/08/addressing
   xmlns:even="http://schemas.xmlsoap.org/ws/2004/08/eventing">

<Header>
  <add:Action>http://schemas.xmlsoap.org/ws/2004/08/eventing/GetStatus</add:Action>
  <add:MessageID>af775c15-5490-4350-9802-b1826ada9bba</add:MessageID>
  <add:From>
    <add:Address>http://localhost:18080/axis/services/WseSink</add:Address>
  </add:From>
  <add:To>http://localhost:18080/axis/services/WseSM</add:To>
  <even:Identifier>2cd0faa7-3a53-46f4-adc1-f186381a94e4</even:Identifier>
</Header>

<Body>
  <even:GetStatus/>
</Body>
</Envelope>
```

2.3.3 Creation of a Renew Request

In the creation of a subscription renewal request a sink needs to specify both the identifier associated with the subscription and also the duration up until which the sink may want the subscription to be renewed. Here, it must be noted that the actual time till which a subscription is renewed might be dependent on the subscription manager. The subscription manager may in some cases choose to limit the durations for which subscriptions may be active.

Once again, just as in the GetStatus Request case once the subscription identifier and the renewal interval have been specified the Sink Processor generates the appropriate SOAP message targeted (through the wsa:To element) to the Subscription Manager managing this subscription. The SOAP message constructed also has other WS-Addressing elements such as wsa:Action duly added as outlined in the WS-Eventing specification. The code snippet below depicts the process of issuing a renew request. Here, the renewal request specifies an expiry interval that is 2 months into the future (from the present time; i.e. if the current time is 1 August 1, 2005 the specified renewal interval is 1 October 2005).

```
String subscriptionIdentifier =
                    "2cd0faa7-3a53-46f4-adc1-f186381a94e4";
Calendar expiresAt = Calendar.getInstance();
expiresAt.add(Calendar.MONTH, 2);

EnvelopeDocument envelopeDocument =
   wseSinkProcessor.createRenew(subscriptionIdentifier, expiresAt);
```

The snippet below depicts the structure of the Renew Request

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"
```

```
xmlns:add="http://schemas.xmlsoap.org/ws/2004/08/addressing"
xmlns:even="http://schemas.xmlsoap.org/ws/2004/08/eventing"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

<Header>
  <add:Action>http://schemas.xmlsoap.org/ws/2004/08/eventing/Renew</add:Action>
  <add:MessageID>f6651a18-cd58-4e09-9051-8d369437449f</add:MessageID>
  <add:From>
    <add:Address>http://localhost:18080/axis/services/WseSink</add:Address>
  </add:From>
  <add:To>http://localhost:18080/axis/services/WseSM</add:To>
  <even:Identifier>2cd0faa7-3a53-46f4-adc1-f186381a94e4</even:Identifier>
</Header>

<Body>
<even:Renew>
  <even:Expires xsi:type="xs:dateTime"
xmlns:xs="http://www.w3.org/2001/XMLSchema">2005-10-01T13:07:15.757-
05:00</even:Expires>
</even:Renew>
</Body>

</Envelope>
```

2.3.4 Creation of an Unsubscribe Request

When a sink is no longer interested in receiving notifications corresponding to a previously registered subscription it needs to issue an Unsubscribe Request. In the creation of an Unsubscribe Request all that needs to be specified is the identifier associated with the subscription in question. Once again, just as in the Renew Request case once the subscription identifier has been specified the Sink Processor generates the appropriate SOAP message targeted (through the wsa:To element) to the Subscription Manager managing this subscription. The code snippet below outlines how this is done.

```
String subscriptionIdentifier =
                    "2cd0faa7-3a53-46f4-adc1-f186381a94e4";
EnvelopeDocument envelopeDocument =
   wseSinkProcessor.createUnsubscribe(subscriptionIdentifier);
```

The snippet below depicts the structure of the Unsubscribe Request

```
<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:add="http://schemas.xmlsoap.org/ws/2004/08/addressing"
xmlns:even="http://schemas.xmlsoap.org/ws/2004/08/eventing">

<Header>
  <add:Action>http://schemas.xmlsoap.org/ws/2004/08/eventing/Unsubscribe</add:Action>
  <add:MessageID>911c34d4-2ce2-44c9-b9cd-2551f5eef09a</add:MessageID>
  <add:From>
    <add:Address>http://localhost:18080/axis/services/WseSink</add:Address>
  </add:From>
  <add:To>http://localhost:18080/axis/services/WseSM</add:To>
  <even:Identifier>2cd0faa7-3a53-46f4-adc1-f186381a94e4</even:Identifier>
</Header>
```

```
<Body>
  <even:Unsubscribe/>
</Body>

</Envelope>
```

2.3.5 Funneling interactions through the Sink and Source processor

The capabilities available within the source and sink processors can be leveraged by applications by making sure that the incoming and outgoing messages are ALL funneled through the appropriate processor. In this section we describe how this is done. Specifically, we outline how this is done when working with services within the OMII container and also with our prototype Filter Pipeline.

The Source Processor copes with subscription requests and the dissemination of notifications to the previously registered sinks with matching subscriptions. The SinkProcessor not only facilitates the creation of appropriate requests, but once these requests are funneled through the processor, it keeps stores these requests so that it can correlate any responses that it receives with previously issued requests. If the response corresponds to a request that the Sink Processor is not aware of, a fault is sent back to the entity that issued the response.

2.3.5.1 Deployments within the OMII Container

For deployment within the OMII container we leverage the JAX-RPC Handler model. Here, services can specify handlers that reside in the processing path between the network communications port and the service implementation itself. These handlers operate on the SOAP message encapsulating the invocation request or response. During deployment a service implementation may specify multiple handlers that would operate on the request/response path of service invocations. This information is specified in the deployment descriptor associated with the service implementation, the order in which these handlers are specified in the deployment descriptor is what dictates the order in which these handlers are invoked by the container engine.

In general the handler approach promotes code reuse since different handlers corresponding to compressions, logging or timestamps can be utilized by multiple services. The order in which these handlers operate on the SOAP message needs to be consistent at the client and the service end-point. For e.g. if an encryption filter is the last filter on the client side, the message needs to pass through a decryption filter before any other filter can operate upon it. If there is a break in the consistency unpredictable results/behavior may ensue. It should also be noted that individual handlers are autonomous entities that have access to the entire SOAP message encapsulating the request/invocation. Individual handlers are allowed to modify both the header and body elements of SOAP messages. Please note that this handler is already available with the source code distribution.

2.3.5.1.1 Deployment within the OMII container using Axis Handlers

Creation of WsInfraAxisHandler

WsInfraAxisHandler is the base class for every handler in the cgl.narada.wsinfra.deployment.axis package. The processed Source and Sink messages are received from the enrouteToNetwork() method and passed into the sender thread. Although it extends the BasicHandler class, the invoke() method is overwritten by the extended classes.

The code snippet below depicts the creation of a WsInfraAxisHandler.

```
public class WsInfraAxisHandler extends BasicHandler
    implements WsMessageFlow {
 public void initilizeInfraAxisHandler() {
    axisMessageInjector = new AxisMessageInjector();
    axisMessageInjector.start();
    handlerVector = new Vector();
  }
  public void addHandler(Handler handler) {
      if(handler!=null)
        handlerVector.add(handler);
  }
/** Routes a message enroute to the application. The message is basically
      routed to a neighboring filter which is nearer to the application.
*/
  public final void enrouteToApplication(SOAPMessage soapMessage) throws
      MessageFlowException {
  }
  /**
   * Push up processed messages and send them into Sender Thread
   */
  public final void enrouteToNetwork(SOAPMessage soapMessage) throws
      MessageFlowException {
      . . . . . . . .  . .
   }
/**This method is overwritten by extend classes */
  public void invoke(MessageContext msgContext) throws AxisFault {
 }
}
```

Developing Sink Handler
Sink Hander extends the WsInfraAxisHandler.

```
Public class WseSinkHandler extends WsInfraAxisHandler {
/**This method will add Response handler chain */
      private WsaEprCreator eprCreator;
      private WseSinkProcessor wseSinkProcessor;
      private EndpointReferenceType sinkEpr;
      private WseNodeUtils wseNodeUtils;
      private String sinkAddress ;
      private WseActions wseActions;
      private String handlerClassNames;
      private String methodName;
```

```
      public WseSinkHandler() throws DeploymentException {
            wseSinkProcessor = WseSinkProcessor.getInstance();
            wseSinkProcessor.setMessageFlow(this);
            wseSinkProcessor.setSinkEPR(sinkEpr);
            wseNodeUtils = WseProcessingFactory.getWseNodeUtils();
            handlerClassNames = WseServicesFactory.getHandlerChain();
            methodName = WseServicesFactory.getMethodName();
            this.initilizeInfraAxisHandler();
            setSoapActionURI(methodName);
            addResponseHandlerChain(handlerClassNames);
      }
      private void addResponseHandlerChain(String classNames) {
       . . . . .
      }
/** This will be invoked by AXIS Engine and must implement to modify 
Message.*/
      public void invoke(MessageContext messageContext)throws AxisFault{
      . . . . .
      }
 /** This method will Process Message coming from the network to Sink.*/
      public void processExchange(EnvelopeDocument envelopeDocument,
      int direction)throws IncorrectExchangeException,
      ProcessingException, MessageFlowException,unknownExchangeException{
      . . . . .
      }
}
```

Developing WseSinkRequestLoggerHandler
This class will be in the Request chain of the WseSink Axis Web Services. This handler is configured in the Sink deployment descriptor request chain. The code snippet below depicts the creation of a Logger Handler.

```
public class WseSinkRequestLoggerHandler extends BasicHandler {
      public void invoke(MessageContext messageContext){
      Message resMessage = messageContext.getCurrentMessage();
        SOAPEnvelope env= resMessage.getSOAPEnvelope();
        Element envElement= env.getAsDOM();
        String strSOAPResponse = XMLUtils.ElementToString(envElement);
      }
}
```

Developing WsInfraResponseLoggerHandler
This handler class will be in the Response chain of all Axis Web Services invoked by another handler. This class will be in the response chain of handlers and configured in the properties file in the handler classes. The code snippet below depicts the creation of a WsInfraResponseLoggerHandler.

```
public class WsInfraResponseLoggerHandler extends BasicHandler {
      public WsInfraResponseLoggerHandler(){}
      public void invoke(MessageContext messageContext){
           try{
              Message resMessage = messageContext.getCurrentMessage();
              SOAPEnvelope env= resMessage.getSOAPEnvelope();
```

```
              Element envElement= env.getAsDOM();
           String strSOAPResponse = XMLUtils.ElementToString(envElement);
        }
}
```

Developing the Sink Web Service

This Web Service is invoked after passing through the request chain of the sink handlers in Axis. WsInfraProcessMethod() is written in this class and configured in the deployment descriptor. The code snippet below depicts the creation of the WseSinkWebService.

```
public class WseSinkWebService {
public void wsInfraProcessMethod(org.apache.axis.message.SOAPEnvelope
              req, org.apache.axis.message.SOAPEnvelope res)
                  throws WsFaultException,ParsingException,
            ProcessingException{
        . . . . . . . . . .
      }
}
```

Developing Source Handlers

The creation of the Source handlers is identical to the creation of the sink handlers, with the exception that the interactions are funneled through the SourceProcessor instead of the SinkProcessor. The code snippet below depicts the Source Handler

```
public class WseSourceHandler extends WsInfraAxisHandler{
public WseSourceHandler()   throws DeploymentException {
        ...........
        wseSourceProcessor = WseSourceProcessor.getInstance();
        wseSourceProcessor.setMessageFlow(this);
        wseSourceProcessor.setSourceEPR(sourceEpr);
        wseSourceProcessor.setSubscriptionManagerEPR(subManagerEpr);
        wseNodeUtils = WseProcessingFactory.getWseNodeUtils();
        handlerClassNames = WseServicesFactory.getHandlerChain();
        methodName = WseServicesFactory.getMethodName();
        this.initilizeInfraAxisHandler();
        setSoapActionURI(methodName);
        addResponseHandlerChain(handlerClassNames);
 }
 private void addResponseHandlerChain(String classNames) {
        StringTokenizer classNameTokenizer = new
        StringTokenizer(classNames, ",");
        while (classNameTokenizer.hasMoreTokens()) {
            String className = classNameTokenizer.nextToken();
            org.apache.axis.handlers.BasicHandler handler =
                (org.apache.axis.handlers.BasicHandler)
            createObject(className);
            this.addHandler(handler);
          }
 }
 public void invoke(MessageContext messageContext){
  . . . . .
 }
}
```

Developing Subscription Manager Handlers

The creation of the Subscription Manager handlers is identical to the creation of the sink or source handlers, with the exception that the interactions are funneled through the SubscriptionManagerProcessor instead of the SinkProcessor. The code snippet below depicts the Subscription Manager Handler

```
public class WseSubscriptionManagerHandler extends WsInfraAxisHandler{
 public WseSubscriptionManagerHandler()throws DeploymentException {
            subscriptionManagerAddress
                   = WseServicesFactory.getSubscriptionManagerAddress();
          eprCreator = WsaProcessingFactory.getWsaEprCreator();
          subManagerEpr=eprCreator.createEpr(subscriptionManagerAddress);
          wseSubscriptionManagerProcessor
                 = WseSubscriptionManagerProcessor.getInstance();
          wseSubscriptionManagerProcessor.setMessageFlow(this);
          wseSubscriptionManagerProcessor
                       .setSubscriptionManagerEPR(subManagerEpr);
          handlerClassNames = WseServicesFactory.getHandlerChain();
          methodName = WseServicesFactory.getMethodName();
          this.initilizeInfraAxisHandler();
          setSoapActionURI(methodName);
          addResponseHandlerChain(handlerClassNames);
        }
   private void addResponseHandlerChain(String classNames) {
   . . . . .
   }
   public void invoke(MessageContext messageContext) {
   . . . . . . . . .
   }
}
```

2.3.5.1.2 Deployment on OMII container using Proxy Web Services.

In proxy model Web Service, there are no handlers and all processing is done in the WsInfraProcessMethod(). WsInfraProcessMethod is responsible for getting instances of processors and invoking the other Web Services.

Sink proxy Web Service

Code snippet for sink proxy Web Service is depicted below.

```
public class WseSinkProxyWebService {
      private SOAPContextFactory soapContextFactory;
      private WseNodeUtils wseNodeUtils;
      private static String sinkAddress=
      WseProxyServicesFactory.getSinkAddress();
      private WseSinkProxyHelper wseSinkProxyHelper;
  public WseSinkProxyWebService(){
            try{
             wseSinkProxyHelper = new WseSinkProxyHelper();
            }catch(Exception ex){System.out.println(moduleName+ex);}
      }
  public void wsInfraProcessMethod(org.apache.axis.message.SOAPEnvelope
      req,org.apache.axis.message.SOAPEnvelope res)throws
```

```
      WsFaultException,ParsingException, ProcessingException{
      try{
       SOAPMessage soapMessage =
       SoapMarshaller.unmarshallSoapMessage(req.toString().getBytes());
       SoapEnvelopeConversion soapEnvelopeConversion
       = SoapEnvelopeConversion.getInstance();
       WseActions wseActions = WseActions.getInstance();
       EnvelopeDocument envelopeDocument =
       soapEnvelopeConversion.getEnvelopeDocument(soapMessage);
       wseNodeUtils = WseProcessingFactory.getWseNodeUtils();
      String from = null;
      String to = null;
      ToDocument toDocument = null;
      FromDocument fromDocument = null;
      AddressingHeaders addressingHeaders= null;
      ParseWsaHeaders parseWsaHeaders =
      WsaProcessingFactory.getParseWsaHeaders();
      try {
         toDocument = parseWsaHeaders.getTo(envelopeDocument);
         fromDocument = parseWsaHeaders.getFrom(envelopeDocument);
         addressingHeaders =
         parseWsaHeaders.getAddressingHeaders(envelopeDocument);
      } catch (ParsingException parseEx) {
        /** Ignore this exception. */
       System.out.println("Problems parsing the envelope"
                   + parseEx.toString());
      }
      String action = null;
      ActionDocument actionDocument = null;
      try{
         actionDocument = addressingHeaders.getAction();
         if (actionDocument == null) {
          String reason = moduleName + "Action missing. " +
          "    Processor only deals with WS-Eventing exchanges.";  }
         if (actionDocument != null) {
           action = actionDocument.getAction().getStringValue();   }
      }catch(Exception e){System.out.println(
       moduleName+"Exception in Processing action "+e);}
       to = toDocument.getTo().getStringValue();
      if(to.equalsIgnoreCase(sinkAddress)){
      System.out.println("*****Received message from Network*****");
      wseSinkProxyHelper.processExchange(
       envelopeDocument,WsMessageFlow.FROM_NETWORK);
      }else{
      System.out.println("***Received message from Application*****");
      wseSinkProxyHelper.processExchange(
      envelopeDocument,WsMessageFlow.FROM_APPLICATION); }
    }catch(UnknownExchangeException e){
      System.out.println(moduleName+" Exception is "+e);
      }catch(IncorrectExchangeException e){
      }catch(MessageFlowException e){
      }catch(Exception e){}
 }
}
```

Developing SinkProxyHelper

This class is invoked from the Sink Web Service and responsible for sending messages over the network using the enrouteToNetwork() method. This class extends the WsInfraAxisHandler class. Code snippet for WseSinkProxyHelper is as shown below.

```
public class WseSinkProxyHelper extends WsInfraAxisHandler{
    public WseSinkProxyHelper() throws DeploymentException {
            eprCreator = WsaProcessingFactory.getWsaEprCreator();
            sinkEpr = eprCreator.createEpr(sinkAddress);
            wseSinkProcessor = WseSinkProcessor.getInstance();
            wseSinkProcessor.setMessageFlow(this);
            wseSinkProcessor.setSinkEPR(sinkEpr);
            handlerClassNames = seProxyServicesFactory.getHandlerChain();
            methodName = WseProxyServicesFactory.getMethodName();
            this.initilizeInfraAxisHandler();
      }
    public void addHandler(WsInfraAxisHandler myHandler){
            handlers.add(myHandler);
      }
    public void processExchange(EnvelopeDocument envelopeDocument,
               int direction)throws IncorrectExchangeException,
            ProcessingException, MessageFlowException,
            UnknownExchangeException{
            if(direction == WsMessageFlow.FROM_NETWORK)
             wseSinkProcessor.processExchange(
             envelopeDocument, WsMessageFlow.FROM_NETWORK);
            if(direction == WsMessageFlow.FROM_APPLICATION){
            boolean validate =  wseSinkProcessor.processExchange(
            envelopeDocument,WsMessageFlow.FROM_APPLICATION);
            if(validate){
            SOAPMessage soapMessage = getSOAPMessage(envelopeDocument);
             enrouteToNetwork(soapMessage);
            }
}
```

Developing Source Proxy Web Service class

The creation of the Source Proxy Web Service is identical to the creation of the sink proxy Web Service, with the exception that the interactions are funneled through the Source Processer instead of the SinkProcessor.

The code snippet below depicts the creation of the WseSourceProxyWebService.

```
public class WseSourceProxyWebService {
      private WseNodeUtils wseNodeUtils;
      private WseSourceProxyHelper wseSourceProxyHelper;
      public void
      wsInfraProcessMethod(org.apache.axis.message.SOAPEnvelope req,
                  org.apache.axis.message.SOAPEnvelope res)
              throws WsFaultException,ParsingException,
              ProcessingException{
              try{
            SOAPMessage soapMessage =
         SoapMarshaller.unmarshallSoapMessage(req.toString().getBytes());
```

```
            SoapEnvelopeConversion soapEnvelopeConversion
                  =  SoapEnvelopeConversion.getInstance();
            wseSourceProxyHelper =  new WseSourceProxyHelper();
            EnvelopeDocument envelopeDocument =
            soapEnvelopeConversion.getEnvelopeDocument(soapMessage);
            String sourceAddress =
            WseProxyServicesFactory.getSourceAddress();
            String smAddress =
            WseProxyServicesFactory.getSubscriptionManagerAddress();
            wseNodeUtils = WseProcessingFactory.getWseNodeUtils();

                  String from = null;
                  String to = null;
                  ToDocument toDocument = null;
                  FromDocument fromDocument = null;
                  AddressingHeaders addressingHeaders= null;
                ParseWsaHeaders parseWsaHeaders =
                WsaProcessingFactory.getParseWsaHeaders();
            try {
            toDocument = parseWsaHeaders.getTo(envelopeDocument);
            fromDocument = parseWsaHeaders.getFrom(envelopeDocument);
            addressingHeaders =
            parseWsaHeaders.getAddressingHeaders(envelopeDocument);
            } catch (ParsingException parseEx) {
            /** Ignore this exception. */
            System.out.println("Problems parsing the envelope"
             + parseEx.toString());
            }
            String action = null;
             ActionDocument actionDocument = null;
            try{
                actionDocument = addressingHeaders.getAction();
                if (actionDocument == null) {
                 String reason = moduleName + "Action missing. " +
                "   Processor only deals with WS-Eventing exchanges.";
                }
               if (actionDocument != null) {
                   action = actionDocument.getAction().getStringValue();
                   }
                  }catch(Exception e){System.out.println(moduleName
                  +"Exception in Processing action "+e);}
                System.out.println("\n"+moduleName+"*** actionType : "
                   +action+"***\n");
                  if(toDocument==null)
                        to = "";
                  else
                        to = toDocument.getTo().getStringValue();
                  if(to.equals(""))
                   wseSourceProxyHelper.getProcessor().processExchange(
                  envelopeDocument,WsMessageFlow.FROM_APPLICATION);
                  else
                  wseSourceProxyHelper.getProcessor().processExchange(
                  envelopeDocument,WsMessageFlow.FROM_NETWORK);
        }catch(UnknownExchangeException ex){
            System.out.println(moduleName+" UnknownExchangeException :
```

```
"+ex); }catch(IncorrectExchangeException ex){
      System.out.println(moduleName+" IncorrectExchangeException : "+ex);
      }catch(DeploymentException ex){
            System.out.println(moduleName+ " DeploymentException : "+ex);
            }catch(MessageFlowException ex){
      System.out.println(moduleName+ " MessageFlowException : "+ex);
            }catch(Exception ex){
      System.out.println(moduleName+" Exception in processMessage()
"+ex);}
}
}
```

Developing SourceProxyHelper

This class is invoked from the Source Web Service and is responsible for sending messages over the network using the enrouteToNetwork() method. This class extends the WsInfraAxisHandler class.

The code snippet below depicts the creation of the WseSourceProxyHelper.

```
public class WseSourceProxyHelper extends WsInfraAxisHandler{
      private static String sourceAddress
           = WseProxyServicesFactory.getSourceAddress();
      private static String subscriptionManagerAddress
            = WseProxyServicesFactory.getSubscriptionManagerAddress();
      private WsaEprCreator eprCreator;
      private WseSourceProcessor wseSourceProcessor;
      private EndpointReferenceType sourceEpr, subManagerEpr;
      private String handlerClassNames;
      private String methodName;
      public WseSourceProxyHelper()
      throws DeploymentException {
         eprCreator = WsaProcessingFactory.getWsaEprCreator();
         sourceEpr = eprCreator.createEpr(sourceAddress);
        subManagerEpr = eprCreator.createEpr(subscriptionManagerAddress);
          wseSourceProcessor = WseSourceProcessor.getInstance();
          wseSourceProcessor.setMessageFlow(this);
          wseSourceProcessor.setSourceEPR(sourceEpr);
          wseSourceProcessor.setSubscriptionManagerEPR(subManagerEpr);
          handlerClassNames = WseProxyServicesFactory.getHandlerChain();
          methodName = WseProxyServicesFactory.getMethodName();
          this.initilizeInfraAxisHandler();
     }
     public WseSourceProcessor getProcessor(){
            return wseSourceProcessor;
      }


}
```

Developing Subscription Manager Proxy Web Service

The creation of the Subscription Manager Web Service is identical to the creation of the sink & source proxy Web Service, with the exception that the interactions are funneled through the Subscription Manager Processer instead of the SinkProcessor.

The code snippet below depicts the creation of a Subscription Manager Web Service.

```
public class WseSubscriptionManagerProxyWebService {
      private WseSubscriptionManagerProxyHelper
      wseSubscriptionManagerProxyHelper;
      public void
      wsInfraProcessMethod(org.apache.axis.message.SOAPEnvelope req,
      org.apache.axis.message.SOAPEnvelope res) throws WsFaultException,
      ParsingException, ProcessingException {
      try {
        wseSubscriptionManagerProxyHelper
        = new WseSubscriptionManagerProxyHelper();
      SOAPMessage soapMessage = SoapMarshaller.unmarshallSoapMessage(req
      .toString().getBytes());
      SoapEnvelopeConversion soapEnvelopeConversion
         = SoapEnvelopeConversion.getInstance();
      EnvelopeDocument envelopeDocument = soapEnvelopeConversion
            .getEnvelopeDocument(soapMessage);
      wseSubscriptionManagerProxyHelper.getProcessor().processExchange(
      envelopeDocument, WsMessageFlow.FROM_NETWORK);
      } catch (UnknownExchangeException e) {
            System.out.println(moduleName + " Exception is " + e);
      } catch (IncorrectExchangeException e) {
            System.out.println(moduleName + " Exception is " + e);
      } catch (MessageFlowException e) {
            System.out.println(moduleName + " Exception is " + e);
      } catch (Exception e) {
            System.out.println(moduleName + " Exception is " + e);
      }
  }
}
```

Developing SubscriptionManagerProxyHelper

This class is invoked from Subscription Manager Web Service and is responsible for sending messages over the network using the enrouteToNetwork() method. This class extends the WsInfraAxisHandler class.

The code snippet below depicts the creation of a WseSubscriptionManagerProxyHelper.

```
public class WseSubscriptionManagerProxyHelper extends
WsInfraAxisHandler{
      private WsaEprCreator eprCreator;
      private WseSubscriptionManagerProcessor
      wseSubscriptionManagerProcessor;
      private static String subscriptionManagerAddress =
        WseProxyServicesFactory.getSubscriptionManagerAddress();
      private EndpointReferenceType subManagerEpr;
      private String handlerClassNames;
```

```
      private String methodName;
      public WseSubscriptionManagerProxyHelper()
          throws DeploymentException {
       eprCreator = WsaProcessingFactory.getWsaEprCreator();
       subManagerEpr = eprCreator.createEpr(subscriptionManagerAddress);
       wseSubscriptionManagerProcessor =
           WseSubscriptionManagerProcessor.getInstance();
       wseSubscriptionManagerProcessor.setMessageFlow(this);
       wseSubscriptionManagerProcessor.setSubscriptionManagerEPR(
          subManagerEpr);
       handlerClassNames = WseProxyServicesFactory.getHandlerChain();
       methodName = WseProxyServicesFactory.getMethodName();
        this.initilizeInfraAxisHandler();
      }
      public WseSubscriptionManagerProcessor getProcessor(){
            return wseSubscriptionManagerProcessor;     }
}
```

2.3.5.2 Deployments within the Filter Pipeline

The filter pipeline model is a prototype system that we built to exhaustively test Web Service specification implementations outside the realms of traditional Web Service containers such as Apache's Axis and Sun's JWSDP. Individual filters are similar in functionality (incremental addition of capabilities) and provide similar advantages (such as code reuse). However, it address several of problems originating from the static nature of Handler Chains and the ability to truly process SOAP messages as autonomous entities.

There are a few steps in dealing with the Filter Pipeline. This includes

1. Creation of the Filter Pipeline
2. Developing the appropriate filter
3. Adding the newly created filter to the Filter pipeline.

Creation of a Filter Pipeline:
The code snippet below depicts the creation of a Filter Pipeline.

```
FilterPipelineFactory filterPipelineFactory =
            FilterPipelineFactory.getInstance();
filterPipeline = filterPipelineFactory.newFilterPipeline("WsePipeline");
```

Developing the appropriate Filter
The Filter whether it is the source filter or the sink filter should extend the cgl.narada.wsinfra.deployment.Filter class. This class is an abstract class, and the implementer needs to implement just the processMessage() method.

In the case of the SinkFilter it is also necessary to initialize the node's Endpoint Reference.

```
public class WseSinkFilter extends Filter {
  private final String identifier = "WseSinkFilter";
  private WsaEprCreator eprCreator;
  private int numOfMessagesFromNetwork = 0;
  private int numOfMessagesFromApplication = 0;

  private WseSinkProcessor wseSinkProcessor;

```

```
  private EndpointReferenceType sinkEpr;
  private String moduleName = "WseSinkFilter: ";

  public WseSinkFilter(String sinkAddress)
    throws DeploymentException {
    eprCreator = WsaProcessingFactory.getWsaEprCreator();
    sinkEpr = eprCreator.createEpr(sinkAddress);

    wseSinkProcessor = WseSinkProcessor.getInstance();
    wseSinkProcessor.setMessageFlow(this);
    wseSinkProcessor.setSinkEPR(sinkEpr);

    setIdentifier(identifier);
  }



  /** This method returns a boolean which indicates whether further
      processing should continue or if it should stop. A return value of
      <i>true</i> indicates that processing should continue; while
      <i>false</i>
      indicates that processing should stop. Note that it is the
      filter-pipeline which is responsible for stopping this processing.
*/
  public boolean
  processMessage(SOAPContext soapContext, int direction)
    throws UnknownExchangeException, IncorrectExchangeException,
    MessageFlowException, ProcessingException {
    String from = "the APPLICATION.";
    if (direction == WsMessageFlow.FROM_NETWORK) {
      from = "the Network.";
    }

    boolean continueOperations =
      wseSinkProcessor.processExchange(soapContext, direction);

    return continueOperations;
  }
}
```

The creation of the Source Filter is identical to the creation of the sink filter, with the exception that the interactions are funneled through the SourceProcessor instead of the SinkProcessor. The code snippet below depicts the Source Filter

```
public class WseSourceFilter extends Filter {
  private final String identifier = "WseSourceFilter";
  private WsaEprCreator eprCreator;
  private int numOfMessagesFromNetwork = 0;
  private int numOfMessagesFromApplication = 0;

  private WseSourceProcessor wseSourceProcessor;

  private EndpointReferenceType sourceEpr, subManagerEpr;
```

```
  private String moduleName = "WseSourceFilter: ";

  public WseSourceFilter(String sourceAddress, String subManagerAddress)
    throws DeploymentException {
    eprCreator = WsaProcessingFactory.getWsaEprCreator();
    sourceEpr = eprCreator.createEpr(sourceAddress);
    subManagerEpr = eprCreator.createEpr(subManagerAddress);

    wseSourceProcessor = WseSourceProcessor.getInstance();
    wseSourceProcessor.setMessageFlow(this);
    wseSourceProcessor.setSourceEPR(sourceEpr);
    wseSourceProcessor.setSubscriptionManagerEPR(subManagerEpr);

    setIdentifier(identifier);
  }



  /** This method returns a boolean which indicates whether further
      processing should continue or if it should stop. A return value of
      <i>true</i> indicates that processing should continue; while
      a value of <i>false</i>
      indicates that processing should stop. Note that it is the
      filter-pipeline which is responsible for stopping this processing.
 */
  public boolean
  processMessage(SOAPContext soapContext, int direction)
    throws UnknownExchangeException, IncorrectExchangeException,
    MessageFlowException, ProcessingException {
    String from = "the APPLICATION.";
    if (direction == WsMessageFlow.FROM_NETWORK) {
      from = "the Network.";
    }
    boolean continueOperations =
      wseSourceProcessor.processExchange(soapContext, direction);

    return continueOperations;
  }
}
```

Adding the newly created Filter to the Filter Pipeline:
The example below depicts the addition of the Filters to the Filter Pipeline:

```
wseSinkFilter = new WseSinkFilter(sinkAddress);
filterPipeline.addFilter(wseSinkFilter);
```

2.3.6 Issuing a message en route to the network

Once a sink has constructed requests that need to be issued to either the Source (subscribe requests) or the Subscription Manager (GetStatus, Renew and Unsubscribe) it needs to ensure the propagation of this message over the network.

2.3.7 Responses to requests issued by the sinks

Responses to requests issued by the Sinks are propagated back to the application. It should be noted that these responses may indicate successful requests or they may indicate problems. If there are a problems these are typically encapsulated within a SOAP Fault Message. In the case of responses that are not FAULT messages, the Sink Processor correlates these responses with previously stored requests. In case the message cannot be correlated with a previously issued response or if the response is not well-formed the Sink Processor will issue a fault back to the entity that issued the response.

Irrespective of whether it is a fault message or a valid response to a previously issued request, the SOAP message is propagated to the application.