

The FIRMS User Guide



FIRMS User Guide: Table of Contents

| | |
|---|-----------|
| 1. A background on acknowledgements and reliable delivery | 4 |
| 1.1 The WSRM & WSR specifications..... | 4 |
| 2. WSRM - Building applications | 6 |
| 2.1 WS-Addressing and the Creation of Endpoint references..... | 6 |
| 3. Leveraging Reliable Communications using WSRM | 7 |
| 3.1.1 Funneling interactions through the Sink and Source processor..... | 9 |
| 3.1.1.1 Deployments within the OMII Container..... | 9 |
| 3.1.1.1.1 Deployment within the OMII container using Axis Handlers..... | 9 |
| 3.1.1.1.1.1 Creation of WsInfraAxisHandler..... | 9 |
| 3.1.1.1.1.2 Developing Sink Handler..... | 10 |
| 3.1.1.1.1.3 Developing Source Request Handler..... | 11 |
| 3.1.1.1.1.4 Developing Source Response Handler..... | 12 |
| 3.1.1.1.1.5 Developing the WsrServiceA:..... | 13 |
| 3.1.1.1.1.6 Developing the WsrServiceB:..... | 14 |
| 3.1.1.1.2 Deployment on OMII container using Proxy Web Services..... | 14 |
| 3.1.1.1.2.1 WSRM Proxy Web Service..... | 14 |
| 3.1.1.1.2.2 Developing WsrSinkProcessor..... | 16 |
| 3.1.1.1.2.3 Developing WsrSourceProcessor..... | 17 |
| 3.1.1.1.2.4 Developing Source Client Service class..... | 18 |
| 3.1.1.1.2.5 Developing Sink Target Service class..... | 19 |
| 3.1.1.2 Deployments within the Filter Pipeline..... | 20 |
| 3.1.1.2.1 Creation of a Filter Pipeline:..... | 20 |
| 3.1.1.2.2 Developing the appropriate Filter..... | 21 |
| 3.1.1.2.3 Adding the newly created Filter to the Filter Pipeline:..... | 23 |
| 3.1.2 Issuing a message en route to the network..... | 23 |
| 3.1.3 Responses to requests issued by the sinks..... | 23 |
| 4. Brief Overview of Message Flow in WS-Reliability: | 24 |
| 5. WSR - Building applications | 24 |
| 5.1 WS-Addressing and the Creation of Endpoint references..... | 24 |
| 5.2 Dealing with the sink side..... | 25 |
| 5.3 Dealing with the source side..... | 26 |
| 5.3.1 Creating a Request..... | 26 |
| 5.3.2 Funneling interactions through the Sink and Source processor..... | 28 |
| 5.3.2.1 Deployments within the OMII Container..... | 28 |
| 5.3.2.1.1 Deployment within the OMII container using Axis Handlers..... | 29 |
| 5.3.2.1.1.1 Creation of WsInfraAxisHandler..... | 29 |
| 5.3.2.1.1.2 Developing Sink Handler..... | 29 |
| 5.3.2.1.1.3 Developing Source Handler..... | 31 |
| 5.3.2.1.1.4 Developing the WsrServiceA:..... | 32 |
| 5.3.2.1.1.5 Developing the WsrServiceB:..... | 32 |
| 5.3.2.1.1.6 Developing Target Service Handler..... | 32 |
| 5.3.2.1.1.7 Developing the Wsr Target Service (WsrTS):..... | 34 |
| 5.3.2.1.2 Deployment on OMII container using Proxy Web Services..... | 34 |
| 5.3.2.1.2.1 Proxy Web Service..... | 34 |
| 5.3.2.1.2.2 Developing SinkProxyHelper..... | 35 |
| 5.3.2.1.2.3 Developing SourceProxyHelper..... | 36 |
| 5.3.2.1.2.4 Developing Target Service Proxy:..... | 37 |
| 5.3.2.1.2.5 Developing TargetServiceProxyHelper..... | 39 |
| 5.3.2.2 Deployments within the Filter Pipeline..... | 40 |

| | | |
|-----------|---|----|
| 5.3.2.2.1 | Creation of a Filter Pipeline: | 40 |
| 5.3.2.2.2 | Developing the appropriate Filter..... | 40 |
| 5.3.2.2.3 | Adding the newly created Filter to the Filter Pipeline: | 42 |
| 5.3.3 | Issuing a message en route to the network | 42 |
| 5.3.4 | Responses to requests issued by the sinks | 42 |

FIRMS Users Guide

Community Grids Lab, Indiana University

This document is intended to provide users with information on developing applications based on WS-ReliableMessaging and WS-Reliability. This specification -- developed jointly by IBM, Microsoft and BEA can be found at

<ftp://www6.software.ibm.com/software/developer/library/ws-reliablemessaging200403.pdf>

This document first provides the reader with a brief overview of Message Flow in Ws-ReliableMessaging. For more details please refer Ws-ReliableMessaging Conceptual Overview Document.

WS-Reliability specification -- developed jointly by Fujitsu, Novell, Oracle, and Sun -- can be found at <http://docs.oasis-open.org/wsrn/2004/06/WS-Reliability-CD1.086.pdf>. This document first provides the reader with a brief overview of Message Flow in Ws-Reliability. For more details please refer Ws-Reliability Conceptual Overview Document. 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. A background on acknowledgements and reliable delivery

Entities involved in reliable messaging need to facilitate easy detection of errors in received sequences while also being able to fix these errors in sequences. In *sender-initiated* protocols a sender gets positive acknowledgments (ACKs) from all receivers periodically. A *positive acknowledgement* confirms the receipt of a specific event by a given receiver. This information along with the knowledge of the events, which an entity is supposed to receive, allows the identification of holes in the delivery sequence at any given node. The sender can then initiate retransmissions to fix these errors.

In *receiver-initiated* protocols errors in received sequences are detected at the receivers, This detection in turn triggers *negative acknowledgements* (NAK) to fix these holes in the delivered sequences and retrieve any previously undelivered events. In receiver initiated protocols the assumption at the sender is that the message has been received at the receiver unless indicated otherwise by the NAKs.

It should be noted that in sender-initiated protocols the error detection, initiation of error correction and the retransmission are all performed at the sender side. In receiver-initiated protocols the error detection and initiation of error corrections are performed at the receiver, while the retransmissions are performed by the sender. ACK based schemes can exist by themselves, while NAK based schemes cannot. This is because in a purely NAK based scheme there is no way for the sender to know for sure if a message was received and hence the sender can never clear the buffer allocated for messages that were sent by the sender.

1.1 The WSRM & WSR specifications

The specifications - WSR and WSRM - both of which are based on XML, address the issue of ensuring reliable delivery between two service endpoints. In this section we outline the similarities in the underlying principles that guide both these specifications. The similarities that we have identified are along the six related dimensions of acknowledgements, ordering and

duplicate eliminations, groups of messages and quality of service, timers, security and fault/diagnostic reporting.

Both the specifications use positive acknowledgements to ensure reliable delivery. This in turn implies that error detections, initiation of error corrections and subsequent retransmissions of “missed” messages can be performed at the sender side. A sender may also proactively initiate corrections based on the non-receipt of acknowledgements within a pre-defined interval.

The specifications also address the related issues of ordering and duplicate detection of messages issued by a source. A combination of these issues can also be used to facilitate exactly once delivery. Both the specifications facilitate guaranteed exactly-once delivery of messages, a very important quality of service that is highly relevant for transaction oriented applications; specifically banking, retailing and e-commerce.

Both the specifications also introduce the concept of a *group* (also referred to as a *sequence*) of messages. All messages that are part of a group of messages share a common group identifier. The specifications explicitly incorporate support for this concept by including the group identifier in protocol exchanges that take place between the two entities involved in reliable communications. Furthermore, in both the specifications the qualities of service constraints that can be specified on the delivery of messages are valid only within a group of messages, each with its own group identifier.

The specifications also introduce timer based operations for both messages (application and control) and group of messages. Individual and group of messages are considered invalid upon the expiry of timers associated with them. Finally, the delivery protocols in the specifications also incorporate the use of timers to initiate retransmissions and to time out retransmission attempts.

In terms of security both the specifications aim to leverage the WS-Security specification, which facilitates message level security. Message level security is independent of the security of the underlying transport and facilitates secure interactions over insecure communication links.

The specifications also provide for notification and exchange of errors in processing between the endpoints involved in reliable delivery. The range of errors supported in these specifications can vary from an inability to decipher a message’s content to complex errors pertaining to violations in implied agreements between the interacting entities.

PART I: - WS-ReliableMessaging

2. WSRM - Building applications

The FIRMS software provides a complete implementation of the WS-ReliableMessaging specification. All functionality related to the various roles – source and sink – with this specification have been implemented.

2.1 WS-Addressing and the Creation of Endpoint references

The WS-ReliableMessaging implementation uses 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. 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);
```

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.

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://docs.oasis-open.org/wsrn/2004/06/ws-reliability-1.1.xsd/ProcessRequest` is a valid action MIH element.

MessageId: This is typically a UUID which uniquely identifies a message.

3. Leveraging Reliable Communications using WSRM

Several WS-* specifications (such as WS-Eventing) require a lot of input from the users to facilitate interactions between entities. However, to facilitate reliable messaging between two endpoints, using FIRMS' implementation of WSRM, is to ensure that the message flows through the appropriate processors. There are two distinct roles within the WSRM - the source and the sink. The WSRM specification facilitates the reliable delivery of messages from the source to the sink. Thus if we were to consider two endpoints **A** and **B** (depicted in [Figure 1](#)), and if we were required to ensure reliable messaging from A to B, we need to ensure that messages generated at **A** flow through the source processor that is configured at endpoint **A** and the sink processor that is configured at endpoint **B**. If one needs to ensure bidirectional reliable communications, a source processor needs to be configured at endpoint **B** and a sink processor needs to be configured at endpoint **A**.

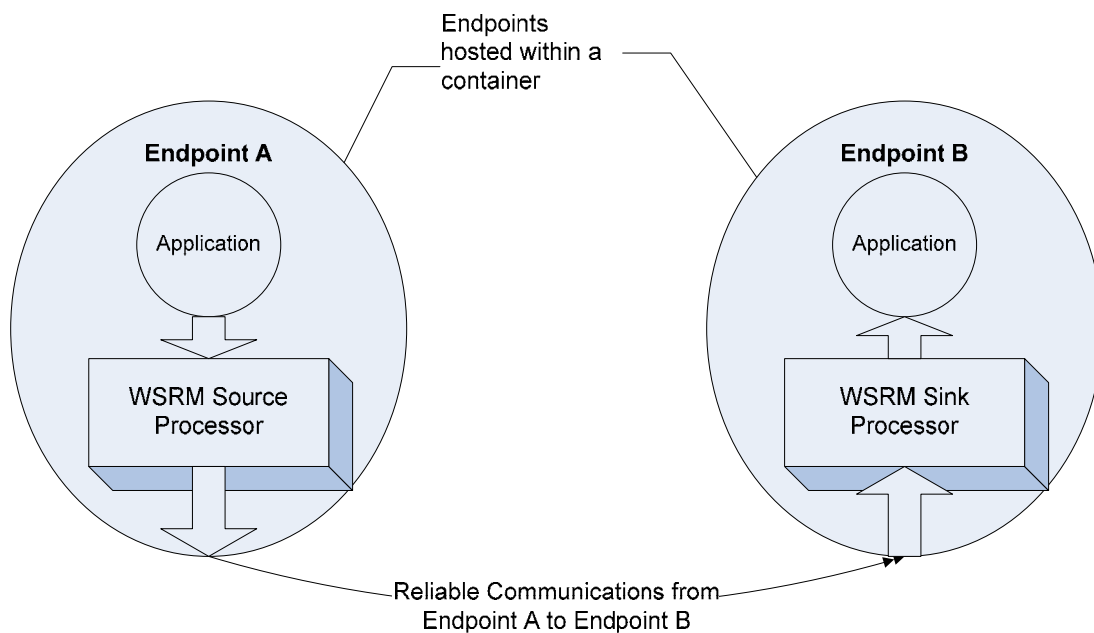


Figure 1: Example Scenario for WSRM communications

Let us now look closely at communications between endpoints **A** and **B**. Furthermore, for the purposes of this discussion let us assume that we are interested in reliable messaging for messages issued from **A** to **B**. In this case, we first configure a source-processor at endpoint **A** and a sink-processor at endpoint **B**. Second, all messages issued by the application at endpoint **A** are funneled through the source-processor. Third, all messages received from the network are funneled through the sink processor at endpoint **B**.

When endpoint A is ready to send a message to endpoint **B**, it creates a SOAP Message with the appropriate WS-Addressing element [`wsa:To`] indicate the endpoint to which the message is targeted. Since all messages are funneled through the source processor, the source-processor at endpoint A receives this message. This source processor then proceeds to initiate the following series of actions.

1. The source-processor at A checks to see if a Sequence (essentially a group of messages identified by a UUID) has been established for messages originating at **A** and targeted to **B**.

- a. If a Sequence has not been established, the source-processor at endpoint A initiates a `CreateSequence` *control message* to initiate the creation of sequence. In WSRM the creation of a Sequence is within the purview of the sink processor at the target endpoint. Upon receipt of this `CreateSequence` request, the sink-processor at the target endpoint B generates a `CreateSequenceResponse` which contains the new established Sequence information. In case there are problems with the `CreateSequence` request, an error/fault may be returned to the originator.
 - b. If a Sequence exists (or if one was established as outlined in item a), the source-processor at the originator endpoint A will associate this Sequence with the message. Additionally, for every Sequence a source-processor also keeps track of the number of messages that were sent by the source endpoint **A** to the target sink-endpoint **B**. For every unique application message (retransmissions, control messages etc are not within the purview of this numbering scheme) sent from **A** to **B** the source-processor at **A** increments the message number by 1. This message number is also included along with the Sequence information.
2. Upon receipt of such a message at the sink endpoint **B**, the sink-processor checks to see if there were any losses in messages that were sent prior to this message (the numbering information reveals such losses). If there were no losses and the message order is correct, the sink-processor releases the message to the application at **B**.
 - a. If there are problems with the received message, such as unknown Sequence Information or if the Sequence was terminated an error message is returned to the source.
 - b. If there are no problems, the message is stored onto stable storage and an acknowledgment is issued based on the acknowledgement interval.
 - c. If a message loss has been detected, the sink will initiate retransmissions by issuing a negative acknowledgement to the source endpoint **A**. This negative acknowledgement will include the message numbers and the Sequence information about the messages that were not received.

A source endpoint can also identify a message as being the last message of a Sequence. This must be specified within the application layer and the source-processor will initiate appropriate actions associated with such a message. The code snippet for identifying a message as the last message of Sequence is listed below.

```
QName qName = WsrnQNames.getInstance().getLastMessageNBX();
String value = "true";
boolean added = soapMessageAlteration.addToSoapHeader(envelopeDocument,
                                                       qName, value);
```

All actions related to ensuring reliable communications are handled by the source and sink processors at the endpoints. The functions performed by the processors include:

1. Initiating the creation of a Sequence. [Source]
2. Responding to a `CreateSequence` request and the creation of a Sequence. [Sink]
3. Tagging Sequence and Numbering information to a message. [Source]
4. Issuing acknowledgements (both positive and negative). [Sink]
5. Processing Acknowledgements and performing retransmissions. [Source].

The remainder of this documentation focuses on the deployment of the processors in various settings within the OMII container.

3.1.1 Funneling interactions through the Sink and Source processor

The capabilities available within the source and sink processors can be leveraged 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.

3.1.1.1 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.**

3.1.1.1.1 Deployment within the OMII container using Axis Handlers

3.1.1.1.1.1 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 initializeInfraAxisHandler() {
        axisMessageInjector = new AxisMessageInjector();
        axisMessageInjector.start();
        handlerVector = new Vector();
    }
    public void addHandler(Handler 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 {
}
}

```

3.1.1.1.2 Developing Sink Handler

Sink Handler extends the `WsInfraAxisHandler`. This handler will be in Request chain of `WsrmserviceA` and `WsrmserviceB`. Sink Handler in turn directly deals with Sink Node Processor. The code snippet below depicts the Sink Handler.

```

public class WsrmsinkHandler extends WsInfraAxisHandler {
    private final String identifier = "WsrmsinkHandler";
    private String moduleName = "WsrmsinkHandler: ";
    private WsaEprCreator eprCreator;
    private WsrmsinkNode wsrmsinkNode;
    private EndpointReferenceType endpointRef;
    WsrmsSequenceMonitor wsrmsSequenceMonitor;

    public WsrmsinkHandler() {
        this.initilize();
    } /**
     * Constructor for the Sink handler and It will initiate all
     required services for Wsrms Handler to process SOAPMessage. @throws
     DeploymentException
     */
    public WsrmsinkHandler() {
        this.initilize();
    }
    public void initilize() {
        try {
            String configInfo = WsrmsServiceParameters.getConfigFile();
            String address = WsrmsServiceParameters.getFromAddress();
            String methodName = WsrmsServiceParameters.getServiceMethodName();
            setIdentifier(identifier);

            eprCreator = WsaProcessingFactory.getWsaEprCreator();
            endpointRef = eprCreator.createEpr(address);
            wsrmsinkNode = new WsrmsinkNode(configInfo);
            wsrmsinkNode.setMessageFlow(this);
            wsrmsinkNode.setEndpointReference(endpointRef);
            //initilize monitor
            WsrmsSequenceMonitorFactory wsrmsSequenceMonitorFactory =
            WsrmsProcessingFactory.getWsrmsSequenceMonitorFactory();
            wsrmsSequenceMonitor =
            wsrmsSequenceMonitorFactory.getWsrmsSequenceMonitor(configInfo,
            this);

```

```

        wsrSequenceMonitor.startServices();
        this.setSoapActionURI(methodName);
        this.initializeInfraAxisHandler();
    }
    catch (Exception e) {
        System.out.println(moduleName + e);
    }
}
/** This will be invoked by AXIS Engine and must implement to
Modify Message.*/
public void invoke(MessageContext messageContext){
    . . . . .
}

/** This method will Process Message coming from the network or
Application to Sink.*/
public boolean processMessage(EnvelopeDocument envelopeDocument,
    int direction) throws UnknownExchangeException,
    IncorrectExchangeException, MessageFlowException,
    ProcessingException {
    . . . . .
}
}

```

3.1.1.1.3 Developing Source Request Handler

The creation of the Source handler is identical to the creation of the sink handlers, with the exception that the interactions are funneled through the SourceProcessor instead of the SinkProcessor. This handler also will be in request chain of WsrServiceA and WsrServiceB. The code snippet below depicts the Source Handler

```

public WsrSourceRequestHandler() extends WsInfraAxisHandler {
    private final String identifier = "WsrSourceHandler";
    private String moduleName = "WsrSourceRequestHandler: ";
    private WsaEprCreator eprCreator;
    private static WsrSourceNode wsrSourceNode;
    private EndpointReferenceType endpointRef;

    public WsrSourceRequestHandler() {
        this.initialize();
    }
    /** initialize required instances for Source handler. */
    public void initialize() {
        try {

            String configInfo = WsrServiceParameters.getConfigFile();
            String address = WsrServiceParameters.getFromAddress();
            String methodName = WsrServiceParameters.getServiceMethodName();
            eprCreator = WsaProcessingFactory.getWsaEprCreator();
            endpointRef = eprCreator.createEpr(address);
            WsrNodeFactoryImpl wsrNodeFactoryImpl =
            WsrNodeFactoryImpl.getInstance();
            if (wsrSourceNode == null) {
                wsrSourceNode =
                wsrNodeFactoryImpl.getWsrSourceNode(configInfo);
                wsrSourceNode.setMessageFlow(this);
            }
        }
    }
}

```

```

        wsrmsourceNode.setEndpointReference(endpointRef);
    }
    setIdentifier(identifier);
    this.setSoapActionURI(methodName);
    this.initializeInfraAxisHandler();
} catch (Exception e) {
    System.out.println(moduleName + e);
}
}
}
/** This will be invoked by AXIS Engine and must implement to
    Modify Message.*/
public void invoke(MessageContext messageContext){
    . . . . .
}

/** This method will Process Message coming from the network or
    Application to Sink.*/
public boolean processMessage(EnvelopeDocument envelopeDocument,
    int direction) throws UnknownExchangeException,
    IncorrectExchangeException, MessageFlowException,
    ProcessingException {
    . . . . .
}
}
}

```

3.1.1.1.14 Developing Source Response Handler

The creation of the Source request handler is identical to the creation of the sink handlers, with the exception that the interactions are funneled through the SourceProcessor instead of the SinkProcessor. This handler also will be in response chain of WsrmserviceA and WsrmserviceB. The code snippet below depicts the Source Response Handler

```

public class WsrmsourceResponseHandler extends WsInfraAxisHandler {
    private final String identifier = "WsrmsourceHandler";
    private String moduleName = "WsrmsourceResponseHandler : ";
    private WsaEprCreator eprCreator;
    private static WsrmsourceNode wsrmsourceNode;
    private EndpointReferenceType endpointRef;
    public WsrmsourceResponseHandler() {
        this.initialize();
    }
    /** initialize required intences for Source handler. */
    public void initialize() {
        try {

            String configInfo = WsrmserviceParameters.getConfigFile();
            String address = WsrmserviceParameters.getFromAddress();
            String methodName = WsrmserviceParameters.getServiceMethodName();
            String handlerClassNames = WsrmserviceParameters.getHandlerChain();
            eprCreator = WsaProcessingFactory.getWsaEprCreator();
            endpointRef = eprCreator.createEpr(address);
            WsrmsourceNodeFactoryImpl wsrmsourceNodeFactoryImpl = WsrmsourceNodeFactoryImpl.getInstance();
            if (wsrmsourceNode == null) {
                wsrmsourceNode = wsrmsourceNodeFactoryImpl.getWsrmsourceNode(configInfo);
            }
        }
    }
}

```

```

    wsrmsourceNode.setMessageFlow(this);
    wsrmsourceNode.setEndpointReference(endpointRef);
}
setIdentifier(identifier);
this.setSoapActionURI(methodName);
this.initializeInfraAxisHandler();
addResponseHandlerChain(handlerClassNames);
}
catch (Exception e) {
    System.out.println(e);
}
}
/** Creates handler objects by given class names and add them into chain of handler. */
private void addResponseHandlerChain(String classNames) {
    .....
}
/** Creates object for a handler class. It takes class name and return an Object */
private Object createObject(String className) {
    . . . . .
}
/** This will be invoked by AXIS Engine and must implement to
    Modify Message.*/
public void invoke(MessageContext messageContext){
    ...}

/** This method will Process Message coming from the network
    From Sink.*/
public boolean processMessage(EnvelopeDocument envelopeDocument,
    int direction) throws UnknownExchangeException,
    IncorrectExchangeException, MessageFlowException,
    ProcessingException, WsFaultException{
    ...}

/** creates Consumer application payload */
public SOAPMessage createSOAPMessage(String from, String to,
    String groupId, BigInteger seqNumber){
    ...}
/** Makes a service call to Sink webservice to send consumer application payload */
public final void enroutetoNetwork(SOAPMessage soapMessage)
    throws MessageFlowException{
    ...}
}

```

3.1.1.1.1.5 Developing the WsrmserviceA:

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

```

public class WsrmserviceA {
    private String moduleName = "WsrmserviceA : ";

```

```

public WsrmserviceA() {}
public void wsInfraProcessMethod(SOAPEnvelope req, SOAPEnvelope resp)
{
    . . .
}

```

3.1.1.1.6 Developing the WsrmserviceB:

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

```

public class WsrmserviceB {
    private String moduleName = "WsrmserviceB : ";
    public WsrmserviceB() {}
    public void wsInfraProcessMethod(SOAPEnvelope req, SOAPEnvelope resp)
    {
        . . .
    }
}

```

3.1.1.1.2 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.

3.1.1.1.2.1 WSRM Proxy Web Service

Code snippet for WSRM Proxy Web Service is depicted below.

```

public class WsrmserviceProxy {
    private String moduleName = "WsrmserviceProxy ";
    private WsrmsinkProcessor wsrmSinkProcessor;
    private WsrmsourceProcessor wsrmSourceProcessor;

    public WsrmserviceProxy() {
        wsrmSinkProcessor = new WsrmsinkProcessor();
        wsrmSourceProcessor = new WsrmsourceProcessor();
    }
    /** Handles reliable messaging. Takes messages, negotiates with other
    endpoint proxy services and sends the message */
    public void wsInfraProcessMethod(SOAPEnvelope req, SOAPEnvelope resp)
    {
        try {
            req = WsInfraEnvelopeModifier.removeDummyBodyElement(req);
        } catch (Exception e) {
            e.printStackTrace();
        }
        SOAPMessage soapMessage = null;
        try {
            soapMessage = cgl.narada.wsinfra.util.SoapMarshaller.

```

```

    unmarshallSoapMessage(req.toString().getBytes());
} catch (Exception e) {
    System.out.println(moduleName + e.toString());
}
SOAPContextImpl soapContext = new SOAPContextImpl(soapMessage);
try {
    SoapEnvelopeConversion soapEnvelopeConversion =
    SoapEnvelopeConversion.getInstance();
    EnvelopeDocument envelopeDocument = soapEnvelopeConversion
        .getEnvelopeDocument(soapMessage);
    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-Reliability exchanges.";
        }
        if (actionDocument != null) {
            action = actionDocument.getAction().getStringValue();
        }
    } catch (Exception e) {
        System.out.println(moduleName+ "Exception in Processing action
            " + e);
    }
    if (toDocument == null) {
        to = "";
    } else {
        to = toDocument.getTo().getStringValue();
    }
    if (fromDocument == null) {
        from = "";
    } else {
        from = fromDocument.getFrom().getAddress().getStringValue();
    }
    boolean continueProcessing=false;
    if(to.equalsIgnoreCase(WsrmServiceParameters.getFromAddress()))
    {
        try {

```

```

        continueProcessing = srmSinkProcessor.processMessage(soapContext,
            WsMessageFlow.FROM_NETWORK);
        if (continueProcessing) {
            String
                destination=WsrmServiceParameters.getTargetServiceAddress();
            String methodName = WsrmServiceParameters.getServiceMethodName();
            try {
                Service service = new Service();
                Call serviceCall = (Call) service.createCall();
                serviceCall.setTargetEndpointAddress(new
                    java.net.URL(destination));
                serviceCall.setSOAPActionURI(methodName);
                serviceCall.invoke(req);
                return;
            }
            catch (Exception e) {
                System.out.println(e);
                e.printStackTrace();
            }
        }
    }
    catch (Exception e) {
        System.out.println(moduleName + e.toString());
    }
    try {
        continueProcessing =
            wsrmsourceProcessor.processMessage(soapContext,
                WsMessageFlow.FROM_NETWORK);
    }catch (Exception e) {
        System.out.println(moduleName + e.toString());
    }
    }else{
        try {
            continueProcessing =
                wsrmsourceProcessor.processMessage(soapContext,
                    WsMessageFlow.FROM_APPLICATION);
        }catch (Exception e) {
            System.out.println(moduleName + e.toString());
        }
    }
    }catch (Exception e) {
        System.out.println(moduleName + e.toString());
    }
    }
}

```

3.1.1.1.2.2 Developing WsrmSinkProcessor

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

Code snippet for `WsrmSinkProcessor` is as shown below.

```

public class WsrmSinkProcessor extends WsInfraAxisHandler {
    private final String identifier = "WsrmSinkProcessor";
    private String moduleName = "WsrmSinkProcessor: ";
    private WsaEprCreator eprCreator;
}

```

```

private WsrmsSinkNode wsrmsSinkNode;
private EndpointReferenceType endpointRef;
WsrmsSequenceMonitor wsrmsSequenceMonitor;
public WsrmsSinkProcessor() {
    this.initilize();
}
/**Initilize required instances for Sink Handler. */
public void initilize() {
try {
    String configInfo = WsrmsServiceParameters.getConfigFile();
    String address = WsrmsServiceParameters.getFromAddress();
    String methodName = WsrmsServiceParameters.getServiceMethodName();
    setIdentifier(identifier);
    eprCreator = WsaProcessingFactory.getWsaEprCreator();
    endpointRef = eprCreator.createEpr(address);
    wsrmsSinkNode = new WsrmsSinkNode(configInfo);
    wsrmsSinkNode.setMessageFlow(this);
    wsrmsSinkNode.setEndpointReference(endpointRef);
    //initilize monitor
    WsrmsSequenceMonitorFactory wsrmsSequenceMonitorFactory =
        WsrmsProcessingFactory.getWsrmsSequenceMonitorFactory();
    wsrmsSequenceMonitor =
        wsrmsSequenceMonitorFactory.getWsrmsSequenceMonitor(configInfo,
            this);
    wsrmsSequenceMonitor.startServices();
    this.setSoapActionURI(methodName);
    this.initilizeInfraAxisHandler();
} catch (Exception ex) {
    ex.printStackTrace();
}
}
/** This method returns a boolean which indicates whether further
    processing should continue or if it should stop. A return value of
    true indicates that processing should continue; while <i>>false</i>
    indicates that processing should stop.*/
public boolean
    processMessage(SOAPContext soapContext, int direction) throws
        UnknownExchangeException, IncorrectExchangeException,
        MessageFlowException, ProcessingException {
    . . . . .
}
}
}

```

3.1.1.1.2.3 Developing WsrmsSourceProcessor

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

Code snippet for `WsrmsSourceProcessor` is as shown below.

```

public class WsrmsSourceProcessor extends WsInfraAxisHandler {
    private final String identifier = "WsrmsSourceProcessor";
    private String moduleName = "WsrmsSourceProcessor: ";
    private WsaEprCreator eprCreator;
    private static WsrmsSourceNode wsrmsSourceNode;
}

```

```

private EndpointReferenceType endpointRef;
public WsrmsSourceProcessor() {
this.initilize();
}
/**initilize required intences for Source handler.*/
public void initilize() {
try {
String configInfo = WsrmsServiceParameters.getConfigFile();
String address = WsrmsServiceParameters.getFromAddress();
String methodName = WsrmsServiceParameters.getServiceMethodName();
eprCreator = WsaProcessingFactory.getWsaEprCreator();
endpointRef = eprCreator.createEpr(address);
WsrmsNodeFactoryImpl wsrmsNodeFactoryImpl =
WsrmsNodeFactoryImpl.getInstance();
if (wsrmsSourceNode == null) {
wsrmsSourceNode =
wsrmsNodeFactoryImpl.getWsrmsSourceNode(configInfo);
wsrmsSourceNode.setMessageFlow(this);
wsrmsSourceNode.setEndpointReference(endpointRef);
}
setIdentifier(identifier);
this.setSoapActionURI(methodName);
this.initilizeInfraAxisHandler();
}catch (Exception ex) {
ex.printStackTrace();
}
}
/** This method returns a boolean which indicates whether further
processing should continue or if it should stop. A return value
of true indicates that processing should continue; while false
indicates that processing should stop. */
public boolean
processMessage(SOAPContext soapContext, int direction) throws
UnknownExchangeException, IncorrectExchangeException,
MessageFlowException, ProcessingException {
. . . . .
}

```

3.1.1.1.2.4 Developing Source Client Service class

The creation of the Source Client Web Service is identical to the creation of the Sink Target Service Web Service, with the exception that the interactions are funneled through the Source Proxy Web Service instead of the Sink Proxy Web Service.

The code snippet below depicts the creation of the WsrmsClientService.

```

public class WsrmsClientService {
private String moduleName = "WsrmsClientService : ";
public WsrmsClientService() {}
/** Creates a message and calls proxy service and receives response
messages */
public void wsInfraProcessMethod(SOAPEnvelope req, SOAPEnvelope resp) {
System.out.println("=====");
System.out.println(moduleName + " Message has been received :\n"+ req);
}
}

```



```

ParseWsaHeaders parseWsaHeaders = WsaProcessingFactory
    .getParseWsaHeaders();
try {
    fromDocument = parseWsaHeaders.getFrom(envelopeDocument);
    addressingHeaders = parseWsaHeaders
        .getAddressingHeaders(envelopeDocument);
} catch (ParsingException parseEx) {
    // Ignore this exception.
    System.out.println("Problems parsing the envelope"
        + parseEx.toString());
}
fromAdd= fromDocument.getFrom().getAddress().getStringValue();
String from = WsrServiceParameters.getFromAddress();
String to = fromAdd;
String methodName=WsrServiceParameters.getServiceMethodName();
String messageType=WsrServiceParameters.getResponseMessage();
SOAPEnvelope soapEnvelope = wsrCreateEnvelope.getEnvelope(
    messageType, from, to);
try {
    Service service = new Service();
    Call serviceCall = (Call) service.createCall();
    serviceCall.setTargetEndpointAddress(new java.net.URL(from));
    serviceCall.setSOAPActionURI(methodName);
    serviceCall.invoke(soapEnvelope);
}
catch (Exception e) {
    System.out.println(e);
    e.printStackTrace();
}
}
}

```

3.1.1.2 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 addresses 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.

3.1.1.2.1 Creation of a Filter Pipeline:

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

```

FilterPipelineFactory filterPipelineFactory =
    FilterPipelineFactory.getInstance();
filterPipeline =
filterPipelineFactory.newFilterPipeline("WsrPipeline");

```

3.1.1.2.2 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 WsrmsSinkFilter extends Filter {
    private final String identifier = "WsrmsSinkFilter";
    private WsaEprCreator eprCreator;
    private int numOfMessagesFromNetwork = 0;
    private int numOfMessagesFromApplication = 0;
    private WsrmsSinkNode wsrmsSinkNode;
    private EndpointReferenceType endpointRef;
    private String moduleName = "WsrmsSinkFilter: ";
    public WsrmsSinkFilter(String configInfo, String address)
        throws DeploymentException {
        eprCreator = WsaProcessingFactory.getWsaEprCreator();
        endpointRef = eprCreator.createEpr(address);
        wsrmsSinkNode = new WsrmsSinkNode(configInfo);
        wsrmsSinkNode.setMessageFlow(this);
        wsrmsSinkNode.setEndpointReference(endpointRef);
        setIdentifier(identifier);
        try {
            WsrmsSequenceMonitorFactory wsrmsSequenceMonitorFactory =
                WsrmsProcessingFactory.getWsrmsSequenceMonitorFactory();
            WsrmsSequenceMonitor wsrmsSequenceMonitor =
                wsrmsSequenceMonitorFactory.getWsrmsSequenceMonitor(configInfo,
                    this);
            wsrmsSequenceMonitor.startServices();
        } catch (WsrmsStorageException wsrmsStorageEx) {
            throw new DeploymentException(moduleName +
                wsrmsStorageEx.toString());
        }
    }
    /** This method returns a boolean which indicates whether further
        processing should continue or if it should stop. A return value
        of true indicates that processing should continue; while false
        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.";
        }
        System.out.println(moduleName + "Processing SOAP Context received
        from " + from);
        boolean continueOperations =
            wsrmsSinkNode.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 Wsrmsourcefilter extends Filter {
    private final String identifier = "Wsrmsourcefilter";
    private WsaEprCreator eprCreator;
    private int numofmessagesfromnetwork = 0;
    private int numofmessagesfromapplication = 0;
    private Wsrmsourcenode wsrmsourcenode;
    private EndpointReferenceType endpointRef;
    private String moduleName = "Wsrmsourcefilter: ";

    public Wsrmsourcefilter(String configInfo, String address)
        throws DeploymentException {
        eprCreator = WsaProcessingFactory.getWsaEprCreator();
        endpointRef = eprCreator.createEpr(address);
        wsrmsourcenode = new Wsrmsourcenode(configInfo);
        wsrmsourcenode.setMessageFlow(this);
        wsrmsourcenode.setEndpointReference(endpointRef);
        setIdentifier(identifier);
    }
    /** This method returns a boolean which indicates whether further processing should continue
    or if it should stop. A return value of true indicates that processing should continue; while false
    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.";
        }
        System.out.println(moduleName + "Processing SOAP Context received from " + from);
        try {
            SOAPMessage soapMessage = soapContext.getSOAPMessage();
            String stringRep = SoapPrinter.getStringRepresentation(soapMessage);
            System.out.println(moduleName + stringRep);
        } catch (Exception e) {
            System.out.println(moduleName + "Problems with the SOAP message " + e.toString());
        }

        boolean continueOperations = wsrmsourcenode.processExchange(soapContext, direction);
        return continueOperations;
    }
}

```

3.1.1.2.3 Adding the newly created Filter to the Filter Pipeline:

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

```
wsrSinkFilter = new WsrSinkFilter(sinkAddress);  
filterPipeline.addFilter(wsrSinkFilter);
```

3.1.2 Issuing a message en route to the network

Once a source has constructed request that need to be issued to sink it needs to ensure the propagation of this message over the network.

3.1.3 Responses to requests issued by the sinks

Responses to requests issued by the Sinks are propagated back to the Source. It should be noted that these responses may indicate successful requests (Acknowledgements) or they may indicate problems (Faults). If there are a problems these are typically encapsulated within a SOAP Fault Message in the SOAP body along with Response in SOAP Header. Source Node processes the response received from source and in case of Response pay load or some permanent fault it informs to the application.

PART II: - WS-Reliability

4. Brief Overview of Message Flow in WS-Reliability:

In Ws-Reliability Producer application sends simple SOAP message with addressing headers (From, To and Action Element) to Source Node processor. This operation is called Submit Operation. Source Node processor adds Reliability header (Request Header) to the received SOAP message and sends to Sink Node Processor to guarantee Reliability. Sink Node processor processes the message. It checks for errors and guarantees all the reliability features (message ordering, message duplication elimination and guaranteed delivery). If received message confirms to all the reliability features, it sends that message to Consumer application. This operation is called Deliver operation. Consumer application may or may not send response payload back to Source Node Processor. If there is a response payload from Consumer Application, it sends it to Sink Node Processor and finally sends it to Source Node Processor and to the Producer Application. This operation is called Response Operation. In turn, Sink Node Processor sends response (Fault or Acknowledgement) to Source Node Processor. Source Node Processor processes the response from the Sink Node Processor. If there is any Permanent fault in response then only Source Node Processor informs to Producer Application. This operation is called Notify operation. The above discussed scenario is depicted in the following **Figure 2**.

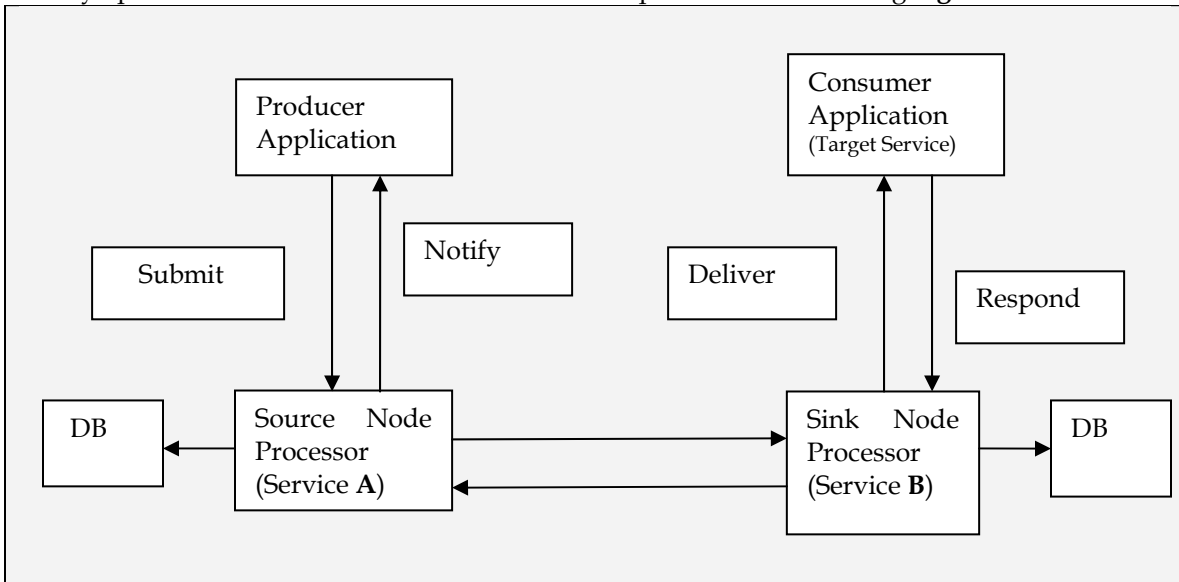


Figure 2 Message Flow in WS-Reliability

5. WSR - Building applications

The FIRMS software provides a complete implementation of the WS-Reliability1.1 specification. All functionality related to the various roles – source and sink – have been implemented.

5.1 WS-Addressing and the Creation of Endpoint references

The WS-Reliability implementation uses 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. 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);
```

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.

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://docs.oasis-open.org/wsrn/2004/06/ws-reliability-1.1.xsd/ProcessRequest` is a valid action MIH element.

MessageId: This is typically a UUID which uniquely identifies a message.

5.2 Dealing with the sink side

The sink node is responsible for generating response for the request received from source node. These response messages are automatically generated by sink upon receiving the request from the source. Sink functionality to guarantee reliability features like Message Ordering, Message Duplication Elimination and Guaranteed Message Delivery is all accessible through the `WsrSinkNode` class. Care must be taken to ensure that all incoming and outgoing messages from a node funnel through this class. Depending on the SOAP Message (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.

Following is the major functionalities encapsulated in this class.

1. It verifies the request received over the network to see if it is well-formed and conforms to the constraints/rules within the WS-Reliability specification.
2. Guarantees Reliability Features: Message Ordering, Message Duplication Elimination, Guaranteed Message Delivery
3. Group Termination.

4. Handles the Response Payload received from the Consumer application.

5.3 Dealing with the source side

Source Node can issue different types of requests depends on parameters set by source client application for request message. However we have a class — WsrSourceNode — which encapsulates the source's capabilities of generation of different requests depends on given parameters. An application source doesn't deal with the generation of well-formed requests since all processing related to this is handled by the SourceNode.

5.3.1 Creating a Request

A request in WS-Reliability requires specifying several parameters. We enumerate these below

1. Source EPR: This is the EPR of the source that sink wishes to send response back.
2. Sink EPR: This is the EPR of the sink. This enables the source to route requests.
3. GroupID: GroupID for the request message.
4. Reply Pattern: Reply Pattern by which source wants response back from sink. It can be Response Reply Pattern, Callback Reply Pattern or Poll Reply Pattern.
5. ReplyTo: Address at which source wants response back while using Callback reply pattern
6. Last Message: Specify true or false if the request is the last request of the Group.

First step is to generate simple EnvelopeDocument containing Addressing Headers.

Following code snippet below depicts generation of EnvelopeDocument with Addressing Headers (From, To and Action).

```
EndpointReferenceType sourceEpr
    = wsaEprCreator.createEpr(sourceAddress);
EndpointReferenceType sinkEpr = wsaEprCreator.createEpr(sinkAddress);
FromDocument from = FromDocument.Factory.newInstance();
from.setFrom(sourceEpr);
ActionDocument action = ActionDocument.Factory.newInstance();
action.addNewAction().setStringValue(wsrActions.getProcessRequest());
RelatesToDocument relatesTo = null;
EnvelopeDocument envelopeDocument = wsaEnvelopeCreator
    .createSoapEnvelope(sinkEpr, from, action, relatesTo);
```

Second step is to add parameters (GroupID, ReplyPattern, LastMessage, ReplyTo) as WS-Reliability Extension Headers.

The code snippet below depicts the addition of (GroupID, ReplyPattern, LastMessage, ReplyTo) parameters as WS-Reliability extension headers. All the QNames associated with above mentioned elements are available in `cgl.narada.wsinfra.wsr.WsrQNames` class.

```
if (groupId != null) {
    QName qName = WsrQNames.getInstance().getGroupId();
    boolean added = soapMessageAlteration.addToSoapHeader(
        envelopeDocument, qName, groupId);
}
if (replyPattern != null) {
    QName qName = WsrQNames.getInstance().getRPattern();
    boolean added = soapMessageAlteration.addToSoapHeader(
        envelopeDocument, qName, replyPattern);
}
```

```

if (replyTo != null) {
    QName qName = WsrQNames.getInstance().getReplyTo();
    boolean added = soapMessageAlteration.addToSoapHeader(
        envelopeDocument, qName, replyTo);
}
if (lastMessage != null) {
    QName qName = WsrQNames.getInstance().getLastMessage();
    boolean added = soapMessageAlteration.addToSoapHeader(
}

```

The snippet below depicts the structure of the SOAP Envelope generated by Source Application.

```

<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:wsr="http://docs.oasis-open.org/wsr/2004/06/ws-reliability-
1.1.xsd/wsrext" xmlns:nar="http://www.naradabrokering.org"
xmlns:add="http://schemas.xmlsoap.org/ws/2004/08/addressing">
  <Header>
    <wsr:LastMessage>>false</wsr:LastMessage>
    <wsr:ReplyPattern>Response</wsr:ReplyPattern>
    <wsr:GroupId>abcdefghi</wsr:GroupId>
    <add:Action>http://docs.oasis-open.org/wsr/2004/06/ws-reliability-
1.1.xsd/ProcessRequest</add:Action>
    <add:MessageID>b4ce1bb5-07c9-4f3d-b9c6-5acab07bbea1</add:MessageID>
    <add:From>
<add:Address>http://localhost:18080/axis/services/WsrServiceA</add:Addr
ess>
    </add:From>
    <add:To>http://localhost:18080/axis/services/WsrServiceB</add:To>
  </Header>
  <Body>
    <nar:Application-Content>Tracker :1</nar:Application-Content>
  </Body>
</Envelope>

```

And last step is to create the instance of Source node and then call processExchange() method of WsrSourceNode class, which will automatically creates Request and sends it to Sink Address. The code snippet below depicts the process.

```

WsrSourceNode wsrSourceProcessor;
String configFile=WsrServicesFactory.getSourceNodeConfigFile();
wsrSourceProcessor = new WsrSourceNode(configFile);
boolean continueOperations= wsrSourceProcessor.processExchange(
    envelopeDocument, direction);

```

The snippet below depicts the structure of the SOAP Envelope generated by Source Node.

```

<Envelope xmlns="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:wsr="http://docs.oasis-open.org/wsr/2004/06/ws-reliability-
1.1.xsd/wsrext" xmlns:nar="http://www.naradabrokering.org"
xmlns:add="http://schemas.xmlsoap.org/ws/2004/08/addressing"
xmlns:ws="http://docs.oasis-open.org/wsr/2004/06/ws-reliability-
1.1.xsd">
  <Header>
    <add:Action>http://docs.oasis-open.org/wsr/2004/06/ws-reliability-
1.1.xsd/ProcessRequest</add:Action>
    <add:MessageID>b4ce1bb5-07c9-4f3d-b9c6-5acab07bbea1</add:MessageID>
    <add:From>

```

```

<add:Address>http://localhost:18080/axis/services/WsrServiceA</add:Address>
  </add:From>
  <add:To>http://localhost:18080/axis/services/WsrServiceB</add:To>
  <ws:Request>
    <ws:MessageId groupId="abcdefghi">
      <ws:SequenceNum last="false" number="0" groupExpiryTime="2005-09-06T11:06:48.878-05:00"/>
    </ws:MessageId>
    <ws:ExpiryTime>2005-09-06T11:01:48.880-05:00</ws:ExpiryTime>
    <ws:ReplyPattern>
      <ws:Value>Response</ws:Value>
    </ws:ReplyPattern>
    <ws:AckRequested/>
    <ws:DuplicateElimination/>
    <ws:MessageOrder/>
  </ws:Request>
</Header>
<Body>
  <nar:Application-Content>Tracker :1</nar:Application-Content>
</Body>
</Envelope>

```

5.3.2 Funneling interactions through the Sink and Source processor

The capabilities available within the source and sink processors can be leveraged 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.

5.3.2.1 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.**

5.3.2.1.1 Deployment within the OMII container using Axis Handlers

5.3.2.1.1.1 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) {
    }

    /** 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 {
    }
}
```

5.3.2.1.1.2 Developing Sink Handler

Sink Handler extends the `WsInfraAxisHandler`. This handler will be in Request chain of `WsrServiceA` and `WsrServiceB`. Sink Handler in turn directly deals with Sink Node Processor. The code snippet below depicts the Sink Handler.

```
public class WsrSinkHandler extends WsInfraAxisHandler {
    private SOAPContextFactory soapContextFactory;
    private WsrNodeUtils wsrNodeUtils;
    private ParseWsaHeaders parseWsaHeaders;
    private AddressingHeaders addressingHeaders;
    private String moduleName = "WsrSinkHandler :";
    private String sourceAddress;
    private String sinkAddress;
    private String targetServiceAddress;
    private WsaEprCreator eprCreator;
    private WsrSinkNode wsrSinkProcessor;
```

```

private EndpointReferenceType sinkEpr, sourceEpr, targetServiceEpr;
private String handlerClassNames;
private String methodName;

/** Constructor for the Sink handler and It will initiate all
required services for Wsr Handler to process SOAPMessage. @throws
DeploymentException */

public WsrSinkHandler() throws DeploymentException {
    sourceAddress = WsrServicesFactory.getSourceAddress();
    sinkAddress = WsrServicesFactory.getSinkAddress();
    targetServiceAddress =
WsrServicesFactory.getTargetServiceAddress();
    eprCreator = WsaProcessingFactory.getWsaEprCreator();
    sourceEpr = eprCreator.createEpr(sourceAddress);
    sinkEpr = eprCreator.createEpr(sinkAddress);
    targetServiceEpr = eprCreator.createEpr(targetServiceAddress);
    String configFile = WsrServicesFactory.getSinkNodeConfigFile();
    wsrSinkProcessor = new WsrSinkNode(configFile);
    wsrSinkProcessor.setMessageFlow(this);
    wsrSinkProcessor.setEndpointReference(sinkEpr);
    wsrNodeUtils = WsrProcessingFactory.getWsrNodeUtils();
    handlerClassNames = WsrServicesFactory.getHandlerChain();
    methodName = WsrServicesFactory.getMethodName();
    setIdentifier(moduleName);
    this.setSoapActionURI(methodName);
    this.initalizeInfraAxisHandler();
    addResponseHandlerChain(handlerClassNames);
    try {
        WsrSinkGroupMonitorFactory wsrSinkGroupMonitorFactory =
WsrProcessingFactory.getWsrSinkGroupMonitorFactory();
        WsrSinkGroupMonitor wsrSinkGroupMonitor=
WsrSinkGroupMonitorFactory
            .getWsrSinkGroupMonitor(configFile, this);
        wsrSinkGroupMonitor.startServices();
    } catch (WsrStorageException wsrStorageEx) {
        throw new DeploymentException(moduleName +
wsrStorageEx.toString());
    }
}

/** This method will add response handler chain */
private void addResponseHandlerChain(String classNames) {
    . . . . .
}

/** This will be invoked by AXIS Engine and must implement to
Modify Message.*/
public void invoke(MessageContext messageContext){
    . . . . .
}

/** This method will Process Message coming from the network or
Application to Sink.*/
public boolean processMessage(EnvelopeDocument envelopeDocument,
int direction) throws UnknownExchangeException,

```

```

        IncorrectExchangeException, MessageFlowException,
        ProcessingException {
        . . . . .
    }
    /** This method enroutes message to application*/
    public final void enroutetoApplication(SOAPMessage soapMessage)
        throws MessageFlowException {
        . . . . .
    }
}

```

5.3.2.1.1.3 Developing Source Handler

The creation of the Source handler is identical to the creation of the sink handlers, with the exception that the interactions are funneled through the SourceProcessor instead of the SinkProcessor. This handler also will be in request chain of WsrServiceA and WsrServiceB. The code snippet below depicts the Source Handler

```

public WsrSourceHandler() throws DeploymentException {
    sourceAddress = WsrServicesFactory.getSourceAddress();
    eprCreator = WsaProcessingFactory.getWsaEprCreator();
    sourceEpr = eprCreator.createEpr(sourceAddress);
    String configFile=WsrServicesFactory.getSourceNodeConfigFile();
    wsrSourceProcessor = new WsrSourceNode(configFile);
    wsrSourceProcessor.setMessageFlow(this);
    wsrSourceProcessor.setEndpointReference(sourceEpr);
    wsrNodeUtils = WsrProcessingFactory.getWsrNodeUtils();
    handlerClassNames = WsrServicesFactory.getHandlerChain();
    methodName = WsrServicesFactory.getMethodName();
    setIdentifier(moduleName);
    this.setSoapActionURI(methodName);
    this.initalizeInfraAxisHandler();
    addResponseHandlerChain(handlerClassNames);
    try {
        WsrGroupMonitorFactory wsrGroupMonitorFactory =
            WsrProcessingFactory
                .getWsrGroupMonitorFactory();
        WsrGroupMonitor wsrGroupMonitor = wsrGroupMonitorFactory
            .getWsrGroupMonitor(configFile, this);
        wsrGroupMonitor.startServices();
    } catch (WsrStorageException wsrStorageEx) {
        throw new DeploymentException(moduleName +
            wsrStorageEx.toString());
    }
}
/** This method will add response handler chain */
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);
    }
}
}

```

```

/** This will be invoked by AXIS Engine and must implement to
    Modify Message.*/
public void invoke(MessageContext messageContext){
    . . . . .
}

/** This method will Process Message coming from the network or
    Application to Sink.*/
public boolean processMessage(EnvelopeDocument envelopeDocument,
    int direction) throws UnknownExchangeException,
    IncorrectExchangeException, MessageFlowException,
    ProcessingException {
    . . . . .
}
}

```

5.3.2.1.1.4 Developing the WsrServiceA:

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

```

public class WsrServiceA {
    private String moduleName = "WsrServiceA : ";
    public WsrServiceA() {}
    public void wsInfraProcessMethod(SOAPEnvelope req, SOAPEnvelope resp)
    {
        . . . .}
}

```

5.3.2.1.1.5 Developing the WsrServiceB:

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

```

public class WsrServiceB {
    private String moduleName = "WsrServiceB : ";
    public WsrServiceB() {}
    public void wsInfraProcessMethod(SOAPEnvelope req, SOAPEnvelope resp)
    {
        . . . .}
}

```

5.3.2.1.1.6 Developing Target Service Handler

The creation of the Target Service handler extends Basic Handler. This handler will be in request chain of Wsr Target Service (`WsrTS`). The major role of this handler is to create Consumer application payload to send back to source node via sink. The code snippet below depicts the Target Service Handler.

```

public class WsrTargetServiceHandler extends BasicHandler implements
WsMessageFlow {
    private SOAPContextFactory soapContextFactory;
    private WsrNodeUtils wsrNodeUtils;
    private ParseWsaHeaders parseWsaHeaders;
    private AddressingHeaders addressingHeaders;
    private String moduleName = "WsrTargetServiceHandler :";
    private String sourceAddress;
    private String sinkAddress;
    private String targetServiceAddress;
    private WsaEprCreator eprCreator;
    private WsrSinkNode wsrSinkProcessor;
    private EndpointReferenceType sinkEpr, sourceEpr, targetServiceEpr;
    private String handlerClassNames;
    private String methodName;
    private WsaEprCreator wsaEprCreator;
    private WsaEnvelopeCreator wsaEnvelopeCreator;
    private SoapEnvelopeConversion soapEnvelopeConversion;
    private SoapMessageAlteration soapMessageAlteration;
    private WsrSourceClientService wsrSourceClientService;
    private WsrActions wsrActions;
    private int tracker;
    /** constructor for the Target Service handler and initiates all
    required applications to process SOAPMessage at Target Service
    Application @throws DeploymentException */
    public WsrTargetServiceHandler() throws DeploymentException {
        sourceAddress = WsrServicesFactory.getSourceAddress();
        sinkAddress = WsrServicesFactory.getSinkAddress();
        targetServiceAddress =
            WsrServicesFactory.getTargetServiceAddress();
        eprCreator = WsaProcessingFactory.getWsaEprCreator();
        sourceEpr = eprCreator.createEpr(sourceAddress);
        sinkEpr = eprCreator.createEpr(sinkAddress);
        targetServiceEpr = eprCreator.createEpr(targetServiceAddress);
        wsrNodeUtils = WsrProcessingFactory.getWsrNodeUtils();
        handlerClassNames = WsrServicesFactory.getHandlerChain();
        methodName = WsrServicesFactory.getMethodName();
        wsrActions = WsrActions.getInstance();
        wsaEprCreator = WsaProcessingFactory.getWsaEprCreator();
        wsaEnvelopeCreator =
            WsaProcessingFactory.getWsaEnvelopeCreator();
        wsrSourceClientService = new WsrSourceClientService();
        soapEnvelopeConversion = SoapEnvelopeConversion.getInstance();
        soapMessageAlteration = SoapMessageAlteration.getInstance();
    }
    /** This will be invoked by AXIS Engine and must implement to
    Modify Message.*/
    public void invoke(MessageContext messageContext){
        ... }

    /** This method will Process Message coming from the network
    From Sink.*/
    public boolean processMessage(EnvelopeDocument envelopeDocument,
        int direction) throws UnknownExchangeException,
        IncorrectExchangeException, MessageFlowException,
        ProcessingException, WsFaultException{

```

```

    . . . }

    /** creates Consumer application payload */
    public SOAPMessage createSOAPMessage(String from, String to,
        String groupId, BigInteger seqNumber){

    . . . }

    /** Makes a service call to Sink webservice to send consumer
    application payload */
    public final void enroutetoNetwork(SOAPMessage soapMessage)
        throws MessageFlowException{
    . . . }
}

```

5.3.2.1.1.7 Developing the Wsr Target Service (WsrTS):

This Web Service is invoked after passing through the Target Service Handler in Axis. `WsInfraProcessMethod()` is written in this class and configured in the deployment descriptor. The code snippet below depicts the creation of the WsrTS.

```

public class WsrTargetService {
    private String moduleName = "WsrTargetService  ";
    public WsrTargetService () {}
    public void wsInfraProcessMethod(SOAPEnvelope req, SOAPEnvelope resp)
    {
    . . . }
}

```

5.3.2.1.2 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.

5.3.2.1.2.1 Proxy Web Service

Code snippet for proxy Web Service is depicted below.

```

public class WsrProxyService {
    private String moduleName = "WsrProxyService  ";
    private WsaEprCreator eprCreator;
    private EndpointReferenceType sinkEpr;
    private String sinkAddress;
    private String sourceAddress;
    private String targetServiceAddress;
    private WsrNodeUtils wsrNodeUtils;
    private WsrSinkProxyHelper wsrSinkProxyHelper;
    private WsrSourceProxyHelper wsrSourceProxyHelper;
    private String methodName;
}

```

```

public WsrProxyService() throws DeploymentException {
    sourceAddress = WsrProxyServicesFactory.getSourceAddress();
    sinkAddress = WsrProxyServicesFactory.getSinkAddress();
    wsrSinkProxyHelper = new WsrSinkProxyHelper();
    wsrSourceProxyHelper = new WsrSourceProxyHelper();
    targetServiceAddress =
        WsrProxyServicesFactory.getTargetServiceAddress();
    eprCreator = WsaProcessingFactory.getWsaEprCreator();
    wsrNodeUtils = WsrProcessingFactory.getWsrNodeUtils();
    methodName = WsrProxyServicesFactory.getMethodName();
}

/**Handles reliable messaging. Takes messages, negotiates with
other endpoint proxy services and sends the message */

public void wsInfraProcessMethod(SOAPEnvelope req, SOAPEnvelope
resp) throws WsFaultException, ParsingException,
ProcessingException {
    . . .
}
}

```

5.3.2.1.2.2 Developing SinkProxyHelper

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

```

public class WsrSinkProxyHelper extends WsInfraAxisHandler {
    private String moduleName = "WsrSinkProxyHelper";
    private WsrSinkNode wsrSinkProcessor;
    private static String sinkAddress
        =WsrProxyServicesFactory.getSinkAddress();
    private static String targetServiceAddress =
        WsrProxyServicesFactory.getTargetServiceAddress();
    private WsaEprCreator eprCreator;
    private EndpointReferenceType sinkEpr, tsEpr;
    private String handlerClassNames;
    private String methodName;

    /**This Constructor will initiate all services to process
SOAPMessage at Wsr Sink Proxy Web Service.
@throws DeploymentException*/

    public WsrSinkProxyHelper()throws DeploymentException {
        eprCreator = WsaProcessingFactory.getWsaEprCreator();
        sinkEpr = eprCreator.createEpr(sinkAddress);
        tsEpr = eprCreator.createEpr(targetServiceAddress);
        String configFile =
            WsrProxyServicesFactory.getSinkNodeConfigFile();
        wsrSinkProcessor = new WsrSinkNode(configFile);
        wsrSinkProcessor.setMessageFlow(this);
        wsrSinkProcessor.setEndpointReference(sinkEpr);
        methodName = WsrProxyServicesFactory.getMethodName();
        this.initalizeInfraAxisHandler();
    }
}

```

```

//Initialize the Monitor
try {
    WsrGroupMonitorFactory wsrGroupMonitorFactory =
        WsrProcessingFactory
            .getWsrGroupMonitorFactory();
    WsrGroupMonitor wsrGroupMonitor = wsrGroupMonitorFactory
        .getWsrGroupMonitor(configFile, this);
    wsrGroupMonitor.startServices();
} catch (WsrStorageException wsrStorageEx) {
    throw new DeploymentException(moduleName +
        wsrStorageEx.toString());
}

}

/** This will enrout the message to Application.
 * @param SOAPMessage soapMessage
 * @throws MessageFlowException
 */
public final void enroutToApplication(SOAPMessage soapMessage)
    throws MessageFlowException {
    . . .
}

/**
 * This will return the instance of Wsr Source Node.
 * @return WsrSourceNode
 */
public WsrSinkNode getProcessor() {
    return wsrSinkProcessor;
}
}

```

5.3.2.1.2.3 Developing SourceProxyHelper

This class is invoked from the Proxy Web Service and is responsible for sending messages over the network using the enroutToNetwork() method of WsrSourceNode class. This class extends the WsInfraAxisHandler class.

The code snippet below depicts the creation of the WsrSourceProxyHelper.

```

public class WsrSourceProxyHelper extends WsInfraAxisHandler {
    private String moduleName = "WsrSourceProxyHelper";
    private static String sourceAddress
        = WsrProxyServicesFactory.getSourceAddress();
    private static String targetServiceAddress
        = WsrProxyServicesFactory.getTargetServiceAddress();
    private WsaEprCreator eprCreator;
    private WsrSourceNode wsrSourceProcessor;
    private EndpointReferenceType sourceEpr, tsEpr;
    private String methodName;

    /**Constructor for the Source Proxy Helper will initiates all
    services for Wsr Source Node to process SOAPMessage
    @ throws DeploymentException*/
    public WsrSourceProxyHelper() throws DeploymentException {
        eprCreator = WsaProcessingFactory.getWsaEprCreator();
        sourceEpr = eprCreator.createEpr(sourceAddress);
    }
}

```

```

    tsEpr = eprCreator.createEpr(targetServiceAddress);
    String configFile =
        WsrProxyServicesFactory.getSourceNodeConfigFile();
    wsrSourceProcessor = new WsrSourceNode(configFile);
    wsrSourceProcessor.setMessageFlow(this);
    wsrSourceProcessor.setEndpointReference(sourceEpr);
    methodName = WsrProxyServicesFactory.getMethodName();
    this.initilizeInfraAxisHandler();
    //Initialize the Monitor
    try {
        WsrGroupMonitorFactory wsrGroupMonitorFactory =
            WsrProcessingFactory
                .getWsrGroupMonitorFactory();
        WsrGroupMonitor wsrGroupMonitor = wsrGroupMonitorFactory
            .getWsrGroupMonitor(configFile, this);
        wsrGroupMonitor.startServices();
    } catch (WsrStorageException wsrStorageEx) {
        throw new DeploymentException(moduleName +
            wsrStorageEx.toString());
    }
}

/**
 * This will return the instance of Source Node.
 * @return WsrSourceNode
 */
public WsrSourceNode getProcessor() {
    return wsrSourceProcessor;
}
}

```

5.3.2.1.2.4 Developing Target Service Proxy:

The code snippet below depicts the creation of a Target Service Proxy.

```

public class WsrTargetServiceProxy {
    private String moduleName = "WsrTargetServiceProxy";
    private WsaEprCreator eprCreator;
    private EndpointReferenceType sinkEpr, sourceEpr, targetServiceEpr;
    private String sinkAddress;
    private String sourceAddress;
    private String targetServiceAddress;
    private WsrNodeUtils wsrNodeUtils;
    private WsrTargetServiceProxyHelper wsrTargetServiceProxyHelper;
    private WsaEprCreator wsaEprCreator;
    private WsaEnvelopeCreator wsaEnvelopeCreator;
    private SoapEnvelopeConversion soapEnvelopeConversion;
    private WsrActions wsrActions;
    private SoapMessageAlteration soapMessageAlteration;
    private String methodName;
    private int tracker;
    /** This Constructor class will initiates all services to process
    SOAPMessage at Sink Node Application.*/
    public WsrTargetServiceProxy(){
        sourceAddress = WsrProxyServicesFactory.getSourceAddress();
        sinkAddress = WsrProxyServicesFactory.getSinkAddress();
        wsrTargetServiceProxyHelper = new

```

```

        WsrTargetServiceProxyHelper();
        targetServiceAddress
            =WsrProxyServicesFactory.getTargetServiceAddress();
        eprCreator = WsaProcessingFactory.getWsaEprCreator();
        wsrNodeUtils = WsrProcessingFactory.getWsrNodeUtils();
        sourceAddress = WsrProxyServicesFactory.getSourceAddress();
        sinkAddress = WsrProxyServicesFactory.getSinkAddress();
        targetServiceAddress
            =WsrProxyServicesFactory.getTargetServiceAddress();
        eprCreator = WsaProcessingFactory.getWsaEprCreator();
        sourceEpr = eprCreator.createEpr(sourceAddress);
        sinkEpr = eprCreator.createEpr(sinkAddress);
        targetServiceEpr = eprCreator.createEpr(targetServiceAddress);
        wsrNodeUtils = WsrProcessingFactory.getWsrNodeUtils();
        methodName = WsrProxyServicesFactory.getMethodName();
        wsrActions = WsrActions.getInstance();
        wsaEprCreator = WsaProcessingFactory.getWsaEprCreator();
        wsaEnvelopeCreator =
            WsaProcessingFactory.getWsaEnvelopeCreator();
        soapEnvelopeConversion = SoapEnvelopeConversion.getInstance();
        soapMessageAlteration = SoapMessageAlteration.getInstance();
    }
    /**
     * This will do the web service for Source in AXIS
     * @param org.apache.axis.message.SOAPEnvelope req
     * @param org.apache.axis.message.SOAPEnvelope res
     * @throws WsFaultException
     * @throws ParsingException
     * @throws ProcessingException
     */
    public void
        wsInfraProcessMethod(org.apache.axis.message.SOAPEnvelope req,
            org.apache.axis.message.SOAPEnvelope res)
            throws WsFaultException, ParsingException,
                ProcessingException {
        . . .
    }
    /**
     * 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(EnvelopeDocument envelopeDocument,
        int direction) throws UnknownExchangeException,
            IncorrectExchangeException, MessageFlowException,
                ProcessingException, WsFaultException {
        . . .
    }
    /**
     * This method will create SOAPMessage based on given parameters.

```

```

    * @param String from address
    * @param String to address
    * @param String groupId in the received message for destination.
    * @param String seqNumber
    * @return
    */
    public SOAPMessage createSOAPMessage(String from, String to,
        String groupId, BigInteger seqNumber) {
        . . .
    }
}

```

5.3.2.1.2.5 Developing TargetServiceProxyHelper

This class is invoked from Target Service Proxy web service and is responsible for sending messages over the network using the `enrouteToNetwork()` method. The code snippet below depicts the creation of a `WsrTargetServiceProxyHelper`.

```

public class WsrTargetServiceProxyHelper implements WsMessageFlow{

    private String moduleName = "WsrTargetServiceProxyHelper";
    private String sinkAddress;
    private String methodName;
    public WsrTargetServiceProxyHelper(){
        sinkAddress = WsrProxyServicesFactory.getSinkAddress();
        methodName = WsrProxyServicesFactory.getMethodName();
    }

    public void enrouteToApplication(SOAPMessage soapMessage){

    }

    /** This method will make service call to Sink Web services
    instead of Source directly. @param SOAPMessage*/
    public final void enrouteToNetwork(SOAPMessage soapMessage)
        throws MessageFlowException {
        try {
            org.apache.axis.message.SOAPEnvelope send
            = (org.apache.axis.message.SOAPEnvelope) soapMessage
                .getSOAPPart().getEnvelope();
            System.out.println(moduleName+
                "****SoapMessage****\n\n\n"+ send);
            Service service = new Service();
            Call call = (Call) service.createCall();
            call.setTargetEndpointAddress(sinkAddress);
            call.setSOAPActionURI(methodName);
            SOAPEnvelope env = (SOAPEnvelope) call.invoke(send);
        }catch (Exception e) {
            System.out.println(moduleName + " Exception here " +
                e);
        }
    }
}

```

5.3.2.2 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 JWS DP. Individual filters are similar in functionality (incremental addition of capabilities) and provide similar advantages (such as code reuse). However, it addresses 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

5.3.2.2.1 Creation of a Filter Pipeline:

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

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

5.3.2.2.2 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 WsrSinkFilter extends Filter {
    private final String identifier = "WsrSinkFilter";
    private WsaEprCreator eprCreator;
    private int numOfMessagesFromNetwork = 0;
    private int numOfMessagesFromApplication = 0;

    private WsrSinkNode wsrSinkNode;
    private EndpointReferenceType endpointRef;
    private String moduleName = "WsrSinkFilter: ";

    public WsrSinkFilter(String configInfo, String address)
        throws DeploymentException {
        eprCreator = WsaProcessingFactory.getWsaEprCreator();
        endpointRef = eprCreator.createEpr(address);
        wsrSinkNode = new WsrSinkNode(configInfo);
        wsrSinkNode.setMessageFlow(this);
        wsrSinkNode.setEndpointReference(endpointRef);
        setIdentifier(identifier);
        try {
            WsrGroupMonitorFactory wsrGroupMonitorFactory =
                WsrProcessingFactory.getWsrGroupMonitorFactory();
            WsrGroupMonitor wsrGroupMonitor =
                wsrGroupMonitorFactory.getWsrGroupMonitor(configInfo, this);
            wsrGroupMonitor.startServices();
```

```

    } catch (WsrStorageException wsrStorageEx) {
        throw new DeploymentException(moduleName +
            wsrStorageEx.toString());
    }
}
/** 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.";
    }

    System.out.println(moduleName + "Processing SOAP Context received
from " +
        from);
    boolean continueOperations =
        wsrSinkNode.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 WsrSourceFilter extends Filter {
    private final String identifier = "WsrSourceFilter";
    private WsaEprCreator eprCreator;
    private int numOfMessagesFromNetwork = 0;
    private int numOfMessagesFromApplication = 0;

    private WsrSourceNode wsrSourceNode;

    private EndpointReferenceType endpointRef;
    private String moduleName = "WsrSourceFilter: ";

    public WsrSourceFilter(String configInfo, String address)
        throws DeploymentException {
        eprCreator = WsaProcessingFactory.getWsaEprCreator();
        endpointRef = eprCreator.createEpr(address);
        wsrSourceNode = new WsrSourceNode(configInfo);
        wsrSourceNode.setMessageFlow(this);
        wsrSourceNode.setEndpointReference(endpointRef);
        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.";
    }

    System.out.println(moduleName + "Processing SOAP Context received
    from " + from);
    try {
        SOAPMessage soapMessage = soapContext.getSOAPMessage();
        String stringRep =
            SoapPrinter.getStringRepresentation(soapMessage);
        System.out.println(moduleName + stringRep);
    } catch (Exception e) {
        System.out.println(moduleName + "Problems with the SOAP message "
+
            e.toString());
    }

    boolean continueOperations =
        wsrSourceNode.processExchange(soapContext, direction);
    return continueOperations;
}

```

5.3.2.2.3 Adding the newly created Filter to the Filter Pipeline:

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

```

wsrSinkFilter = new WsrSinkFilter(sinkAddress);
filterPipeline.addFilter(wsrSinkFilter);

```

5.3.3 Issuing a message en route to the network

Once a source has constructed request that need to be issued to sink it needs to ensure the propagation of this message over the network.

5.3.4 Responses to requests issued by the sinks

Responses to requests issued by the Sinks are propagated back to the Source. It should be noted that these responses may indicate successful requests (Acknowledgements) or they may indicate problems (Faults). If there are a problems these are typically encapsulated within a SOAP Fault

Message in the SOAP body along with Response in SOAP Header. Source Node processes the response received from source and in case of Response pay load or some permanent fault it informs to the application.