A Retrospective on the Development of Web Service Specifications

Shrideep Pallickara, Geoffrey Fox, Mehmet S Aktas, Harshawardhan Gadgil, Beytullah Yildiz,
Sangyoon Oh, Sima Patel, Marlon Pierce and Damodar Yemme
(spallick, gcf, maktas,hgadgil, byildiz, ohsangy,skpatel, mpierc, dyemme)@indiana.edu
Community Grids Lab, Indiana University

1. Introduction

Web Services have gained considerable traction over the past several years, and are being increasingly leveraged within the academic, business and research communities. The Service Oriented Architecture (SOA) model engendered within Web Services provides a simple and flexible framework for building sophisticated applications. A slew of specifications addressing several core areas – such as reliable messaging, addressing, security etc – within distributed systems have emerged recently. The term WS-* is used as an umbrella term to collectively refer to these specifications. The use of XML throughout the Web Services stack of specifications facilitates interactions between services implemented in different languages, running on different platforms, and over multiple transports. This use of XML distinguishes Web Services from previous efforts such as CORBA (Common Object Resource Broker Architecture) to simply building distributed systems.

In this chapter we describe our experiences with several Web Service specifications. In general lessons learnt, and design decisions made, during these implementations would be applicable to several other specifications. We begin this chapter with some observations regarding Web Service specifications in section 2. This includes a discussion on the SOAP-centric (Simple Object Access Protocol) [Gudgin 2003] nature of the specifications, their reliance on one-way asynchronous message exchanges, how a specification can itself leverage other specifications, and, finally, how these specifications are intended to be stackable to facilitate use in tandem with each other. Several specifications leverage the WS-Addressing [Box 2004a] specification and this specification has, in recent years, become the de facto standard to target Web Service instances; we also include a brief description of this specification.

WS-* specifications are XML-based and have schemas associated with them. In section 3 we describe the choices available to designers for processing these schemas. In subsequent sections we describe the implementation strategy for various specifications that we have implemented. These include WS-ReliableMessaging [Bilorusets 2004] (hereafter WSRM), WS-Reliability [Oasis-WSR 2004] (hereafter WSR), WS-Eventing [Box 2004b] (hereafter WSE), WS-Context [7], the Universal Description, Discovery and Integration (UDDI) [Bunting 2003], WS-Management [Arora 2005] and WS-Transfer [Alexander 2004a]. Depending on the interactions and exchanges that are part of these specifications the complexity of the implementation and corresponding deployments varies.

The WSRM and WSR specifications pertain to providing support for reliable messaging between Web service endpoints. These aforementioned specifications guarantee delivery of messages in the presence of failures and disconnects; endpoints can also retrieve lost messages after a failure. The WSE specification provides a mechanism for routing notifications from the producers to the registered consumers. Consumers can register their interest in specific messages using XPath queries; only messages that satisfy the previously specified constraint are routed to a consumer. Implementation of the WSRM, WSR and WSE specifications are outlined in section 4.

WS-Management facilitates the efficient management of distributed systems; this specification identifies a core set of Web Service specifications and usage requirements to expose a common set of operations central to all systems management. Our implementation is described in section 5.

WS-Context models session metadata as an external entity where more than two services can access/store highly dynamic shared metadata. Extensions to the WS-Context specification and implementation are described in section 6. The UDDI specification defines a searchable repository of Web Service Description Language (WSDL) [Christensen 2001] specifications of Grid/Web Services. We have designed and implemented a hybrid information service that provides semantics for both types of information that are defined by the WS-Context and UDDI Specifications; this is described in section 7. Issues related to the use of Web Services in power and compute constrained devices such as mobile devices are described in section 8.

In section 9, we discuss issues within a dominant Web Service container – Apache Axis – vis-à-vis support for interactions mandated within Web Service specifications. We also describe deployment strategies to cope with the constraints imposed within the container. In section 10, we describe use cases for these specifications. Specifically, we outline how we leveraged implementations of various WS-* specifications in various projects and settings. Finally, in section 11 we outline our conclusions.

2. Some observation about WS-* specifications

WS-* specifications typically tend to address cores areas or areas where the demand is sufficiently high that it makes sense to eschew proprietary solutions. In some cases, if there is a common thread among several specifications, this needs to be abstracted into a specification in its own right. Exemplars of this include WS-Addressing which provides a scheme to address Web Service endpoints and WS-Policy which provides a framework for exchanging policy information between the service endpoints.

Most of these specifications are developed such that they can be leveraged by other specifications. The specifications also provide a framework to facilitate the incremental addition of capabilities at a given service endpoint. In some cases, these specifications are stackable and when used together provides capabilities available in the stacked specifications.

The specifications also specify a WSDL document, which describes message formats and message exchange patterns associated with them. However, all communications and exchanges outlined within these specifications are to be encapsulated within stand-alone SOAP (Simple Object Access Protocol) messages. SOAP is an XML-based protocol, and provides a framework for building self-contained messages that can reference elements from other schemas. This extensibility mechanism allows a given SOAP message to contain elements from schemas related to different specifications (such as WS-Addressing, WS-Security etc) at the same time should the need arise. SOAP messages are also used by service endpoints to report faults and errors related to processing messages. The extensibility framework in SOAP allows a given SOAP message to contain elements from schemas related to different specifications.

Here, we also note that SOAP-based asynchronous interactions and the stackable nature of these specifications are particularly well-suited for one-way messaging. However, most Web Service containers (at least for Java) are designed around the RPC-style request-response model as its primary mode of interaction.

2.1 WS-Addressing

WS-Addressing is a way to abstract, from the underlying transport infrastructure, the addressing needs of an application. WS-Addressing is thus central to most Web Service specifications. WS-Addressing incorporates support for end point references and message information headers. End point references standardize the format for referencing (and passing around references to) both a Web service and Web service instances as well. The message information headers standardize information, pertaining to message processing, related to replies, faults, actions and the relationship to prior messages. This is especially useful in cases where there would be multiple dedicated entities dealing with these different cases. The message information headers elements comprise the following:

- **To** (mandatory element): This specifies the intended receiver of message. If there are end point references contained in the SOAP header element, this identifies the node that is responsible for routing the message to the 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 generated 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` element.
- **MessageId**: This is typically a UUID which uniquely identifies a message. This is also used to 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.

3. Processing the XML schemas

One of the most important decisions while developing Web Service specifications is the choice of the tool to use while processing the XML schema related to the specification being implemented (along with those that are leveraged within the aforementioned specification) and SOAP. We were looking for a solution that allowed us to process XML from within the Java domain. Here, there were four main choices.

First, we could develop these Java classes along with the parsing capabilities ourselves. This is the approach that was used in Apache's Sandesha project which provides an implementation of WSRM. This approach is error-prone and is generally quite difficult. Furthermore, this approach quickly becomes infeasible as the complexity of the schema increases, and also as the number of leveraged specifications increases. Another possible approach is to process these messages based on the DOM (Document Object Model) model; here, the development process would be quite complicated; presently, we are not aware of any system that leverages this scheme.

Second, we could use the Axis Web Service container's *wsdl2java* compiler. Issues (in version 1.2) related to this tool's support for schemas have been documented in Ref [Gibbs 2003]. Specifically, the problems related to insufficient (and, in some cases, incorrect) support for complex schema types, XML validation and serialization issues. This precludes the use of this tool in several settings.

The third approach was to use the JAXB specification – a specification from Sun to deal with XML and Java data-bindings. The classes generated through JAXB though better than what is generated using Axis' *wsdl2java* still does not provide complete support for the XML Schema. One may thus run into situations where one may find inaccessible data elements. We looked into both the JAXB reference implementation from Sun and JaxMe from Apache (which is an open source implementation of JAXB) before deciding not to use this approach for processing the WS-* schemas.

The final approach involves utilizing tools which focus on complete schema support. Here, there were two candidates –- XMLBeans and Castor –- which provide good support for XML Schemas. We settled on XMLBeans because of two reasons. First, it is an open-source effort. Originally developed by BEA, it was contributed by BEA to the Apache Software Foundation. Second, in our opinion, it provides the best and most complete support for the XML schema of all the tools currently available. It allows us to validate instance documents and also facilitates simple but sophisticated navigation of XML documents. Finally, the XML generated by the corresponding Java classes is true XML that conforms to, and can be validated against, the original schema.

4. WSRM, WSR and WSE

In this section, we describe our scheme for the implementation of the WSRM, WSR and the WSE specifications. One thing to note about these specifications is that they are intended to provide incremental addition of capabilities at a given service endpoint. The WSRM and WSR specifications pertain to providing reliable delivery of messages between service endpoints, while the WSE specification provides a framework for routing notifications from a source to registered sinks. We first provide an overview of these systems before we describe the implementation strategy for these specifications. Here we note that these implementations are available as part of the NaradaBrokering project (http://www.naradabrokering.org/). Furthermore, implementation strategies for WSRM and WSR are identical, and we have chosen to describe only WSRM in this section.

4.1 The base framework for implementing the specifications

In order to facilitate incremental addition of capabilities at an endpoint, functionality related to the specification should be encapsulated in a *processor* which processes these SOAP messages. We first developed the most generic version of this processor – the `WsProcessor` – which can be leveraged by the implementations. Furthermore, in the specifications that we consider there are multiple roles that have been outlined within the same specification; in this case, each role needs to extend this basic processor with additional functionality.

The basic `WsProcessor` serves the following functions. First, it provides a framework for funneling interactions in a manner that is suitable for incrementally adding capabilities to the service endpoint. Second, it provides a framework for delegating the networking requirements to another interface; this enables the developer to simply focus on implementing the specification without having to focus on developing a scheme to transporting SOAP messages. Finally, the `WsProcessor` also provides a framework for reporting a variety of exceptions related to deployments and faults that have been outlined within the specification being implemented.

The `WsProcessor` contains a method `processExchange()` which can be used by the endpoint to funnel all inbound and outbound messages to and from the endpoint. By funneling all messages through the processors we also have the capability of shielding the web service endpoints from some of the control messages that are exchanged as part of the routine exchanges between WS-* endpoints. For example, a web service endpoint need not know about (or cope with) control messages related the acknowledgements and the creation/termination of Sequences in WSRM.

Included below is the definition of the `processExchange()` method. Using the `SOAPContext` it is possible to retrieve the encapsulated SOAP message. The logic related to the processing of the funneled SOAP messages is different depending on whether the SOAP message was received from the application or network. Exceptions thrown by this method are all *checked* exceptions and can thus be trapped using appropriate try-catch blocks. Depending on type of the exception that is thrown, either an appropriate SOAP Fault is constructed and routed to the

relevant location, or it triggers an exception related to processing the message at the node in question. A processor decides on processing a SOAP message based one of three parameters

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

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

If the `WsProcessor` instance does not know how to process a certain message, it throws an `UnknownMessageException` an example of this scenario is a WSRM processor receiving a control message corresponding to a different Web Service specification such as a WSE `Subscribe` request. An `IncorrectExchangeException` is thrown if the `WsProcessor` instance should not have received a specific exchange. For example if a WSRM sink receives a `wsrm:Acknowledgement` it would throw this particular exception since acknowledgements are processed by the source. `MessageFlowException` reports problems related to networking within the container environment within which the `WsProcessor` is hosted. The `ProcessingException` corresponds to errors related to processing the received SOAP message. This is typically due to errors related to the inability to locate protocol elements within the SOAP message, the use of incorrect (or different versions of) schemas and no values being supplied for some schema elements.

If the `ProcessingException` was caused due to a malformed SOAP message received over the network, an appropriate SOAP Fault message is routed back to the remote endpoint. If a `ProcessingException` was thrown due to messages received from the hosting web service endpoint or if networking problems are reported in the `MessageFlowException` processing related to the SOAP message is terminated immediately.

Another class of interest is the `WsMessageFlow` class. This interface contains two methods `enrouteToApplication()` and `enrouteToNetwork()` which are leveraged by the `WsProcessor` to route SOAP messages (requests, responses or faults) *en route* to the hosting web service or a network endpoint respectively. The `WsProcessor` has methods which enable the registration of `WsMessageFow` instances. Since the `WsProcessor` delegates the actual transmission of messages to Web Service container-specific implementations of the `WsMessageFlow`, it can be deployed in a wide variety of settings within different Web Service containers such as Apache Axis and Sun's JWSDP by registering the appropriate `WsMessageFlow` instance with the `WsProcessor`. The capabilities within the `WsProcessor` and the `WsMessageFlow` enable the developer to focus only on the logic related to the respective roles within the specifications being implemented.

4.2 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 qualities-of-service (QoS) 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 QoS constraints specified on the delivery of messages are valid only within a group of messages, each of which has its own group identifier.

The specifications also introduce timer-based operations for both messages (application and control) and groups of messages. Individual and groups of messages are considered invalid upon the expiry of timers associated with them. Finally, the delivery protocols in these 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.

4.2.1 WSRM Implementation

In our implementation (Java-based) functionality related to the sink and sink roles in WSRM are encapsulated within the `WSRMSourceProcessor` and `WSRMSinkProcessor` respectively. Both these processors extend the `WsProcessor` base class. It should be noted that a given endpoint may be a source, sink or both for the reliable delivery of SOAP messages. In the case that the endpoint is both a source and a sink, both the `WSRMSourceProcessor` and the `WSRMSinkProcessor` will be cascaded at the endpoint.

Upon receipt of an outgoing SOAP message the `WSRMSourceProcessor` checks to see if an active `Sequence` currently exists between the hosting endpoint and the remote endpoint. If one does not exist, the `WSRMSourceProcessor` automatically initiates a create sequence exchange to establish an active `Sequence`. For each active `Sequence`, the `WSRMSourceProcessor` also keeps track of the `Message Number` last assigned to ensure that they monotonically increase, starting from 1. The `WSRMSourceProcessor` performs other functions as outlined in the WSRM specification which includes *inter alia* the processing of acknowledgements, issuing retransmissions and managing inactivity related timeouts on Sequences. The `WSRMSinkProcessor` responds to the requests to create a sequence, and also acknowledges any messages that are received from the source. The `WSRMSinkProcessor` issues acknowledgements (both positive and negative) at predefined intervals and also manages inactivity timeouts on `Sequence`s. Finally, both the `WSRMSourceProcessor` and `WSRMSinkProcessor` detect any problems related to malformed SOAP messages and violations of the protocol, and throw the appropriate faults as outlined in the WSRM specification.

Since WSRM leverages capabilities within WS-Addressing and WS-Policy we also had to implement Processors which incorporate support for rules and functionalities related to these specifications. While generating responses to a targeted web service, WS-Addressing rules need to be followed in dealing with the elements contained within a service's end point reference. Similarly responses, and faults are targeted to a web service or designated intermediaries based on the information encapsulated in other WS-Addressing elements such as `wsa:ReplyTo` and `wsa:FaultTo` elements. The WS-Policy specification is used to deal with policy issues related to sequences. An entity may specify policy elements from an entire range of sequences. The WSRM processors leverages capabilities available within these WS-Addressing and WS-Policy processors to enforce rules/constraints, parsing and interpretation of elements, and the generation of appropriate SOAP messages (as in WS-Addressing rules related to the creation of a SOAP message targeted to a specific endpoint).

Upon receipt of a SOAP message, at either the `WSRMSourceProcessor` or the `WSRMSinkProcessor`, the first set of headers that need to be processed are those related to WS-Addressing. For example, the first header that is processed in typically the `wsa:From` element which identifies the originator of the message. The `wsa:To` element is also checked to make sure that the SOAP message is indeed intended for the hosting web service endpoint. In the case of control exchanges, the semantic intent of the SOAP message is conveyed through the `wsa:Action` element in WS-Addressing. Similarly, the relationship between a response and a previously issued request is captured in the `wsa:RelatesTo` element.

WSRM requires the availability of a stable storage at every endpoint. The storage service leverages the JDBC API which allows interactions with any SQL-compliant database. Our implementation has been tested with two relational databases — MySQL and PostgreSQL. Comprehensive details, and results, pertaining to the WSRM implementation can be found in [Pallickara 2005].

Several WS-* specifications (such as WSE) 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, all an entity needs to do 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**.

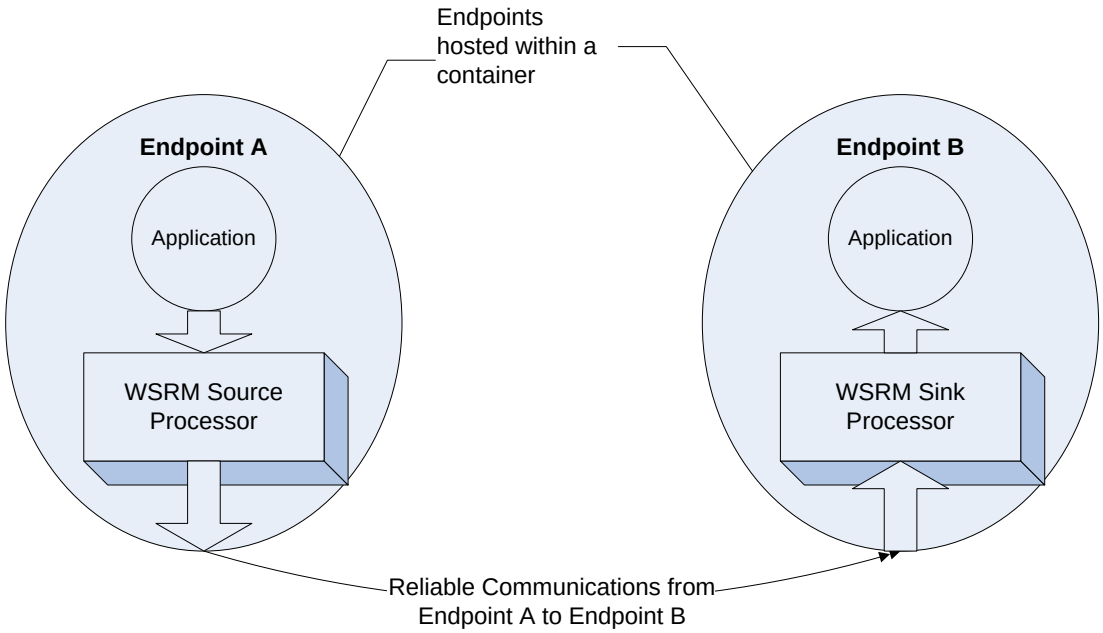

Figure 1: Example Scenario for WSRM communications

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 1.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.

4.3 WSE

WSE 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. WSE, 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 WSE 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 co-location of the source and the subscription manager on the same machine. The subscription manager performs the following operations

- It is responsible for enabling sinks to 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.

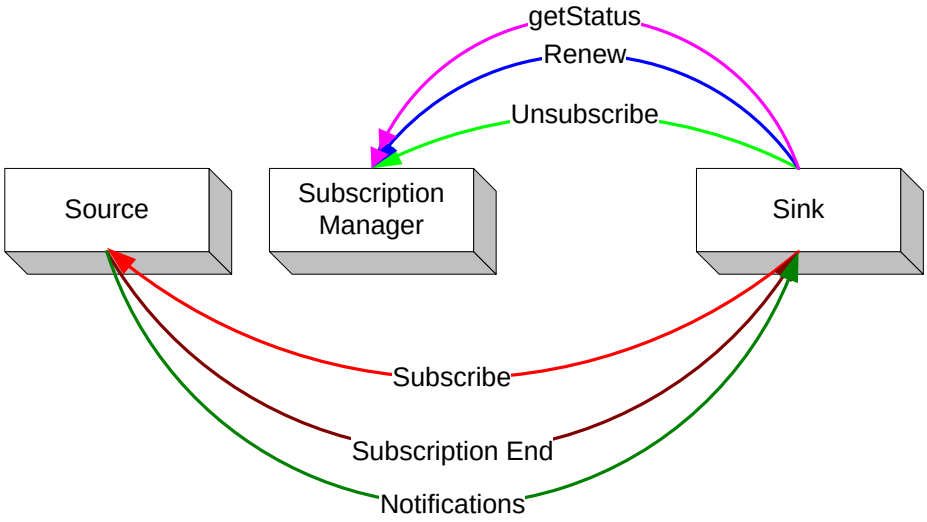

Figure 2: WSE - Chief components

Figure 2 depicts the chief components in WSE. 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.

4.3.1 WSE Implementation

There are three distinct roles within WSE viz. source, sink and subscription manager. In our implementations we have a processor corresponding to each of these roles. These processors contain methods to generate appropriate SOAP requests, responses and faults as outlined in the specification.

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 `WseSourceProcessor` class. Care must be taken to ensure that all incoming and outgoing messages for a given endpoint be funneled through this class. Depending on the `SOAPMessage` (and the information encapsulated

therein) and whether the message was received from the application or over the network, this processor deals with exchanges as outlined by the specification.

The functions performed by the `WseSourceProcessor` are enumerated below

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 WSE 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 WSE source functionality available within this class is accessible to the application. A source thus needs to be only concerned with its primary role –- generation of notifications.

Next, the `WseSubscriptionManagerProcessor` needs to be deployed. A subscription manager could be associated with multiple sources at the same time. Once a sink has subscribed with the source, all subsequent actions such as `GetStatus`, `Renew` and `Unsubscribe` should be directed at the Subscription Manager specified in the source's response to the original subscribe request.

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 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 `WseSinkProcessor`. 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. 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.
2. `Renew`: This exchange allows a sink to renew previously registered subscriptions, so that they expire at a later time.
3. `Unsubscribe`: This indicates that a sink is no longer interested in the receipt of notifications corresponding to a previously registered subscription.

Upon receipt of a `SubscribeResponse` to the Subscribe request the `WseSinkProcessor` keeps track of both the subscription identifier (a UUID) and the `SubscriptionManager` information contained in the received response. When an endpoint needs to perform actions such as `GetStatus`, `Renew` and `Unsubscribe` all that it needs to specify is the subscription identifier, and the `WseSinkProcessor` constructs the corresponding SOAP message targeted to the appropriate Subscription Manager.

The WSE 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 develop and register 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 that needs 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()`which is related to the actual matching operation. The signature of this method has been included for the reader's perusal.

```
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)
```

5. Management within Distributed Systems

As application complexity grows, the need for efficient management of system arises. Various system specific management architectures have been developed previously, and have been quite successful in their areas. Examples include SNMP (Simple Network Management Protocol) (Case 1990) and CMIP (Warrier 1990). The chief drawback in these management systems is interoperability. To address interoperability, the distributed systems community has been orienting towards the Web Services framework, and the corresponding WS-* suite of specifications that defines rich functions while allowing services to be composed to meet varied QoS requirements.

A crucial application of the Web Services architecture is in the area of systems management. WS Management (Arora 2005) and WS Distributed Management (WSDM) (HP 2005) are two competing specifications in the area of management using Web Services architecture. Both specifications focus on providing a Web service model for building system and application management solutions specifically focusing on resource management. This includes basic capabilities such as creating and deleting resource instances, setting and querying service specific properties, and providing an event-driven model to connect services based on the publish / subscribe paradigm.

WSDM on the other hand breaks management in two parts, Management using Web Services (MUWS) and Management of Web Services (MOWS). MUWS focuses on providing a unifying layer on top of existing management specifications such as CIM from DMTF, SNMP and OMI models. MOWS presents a model where a Web Service is itself treated as a manageable resource. Thus, MOWS will serve to provide support for the management framework and support varied activities such as service metering, auditing, SLA management, problem detection and root cause failure analysis, service deployment, performance profiling and life-cycle management.

WS-Management on the other hand attempts to identify a core set of Web Service specifications and usage requirements to expose a common set of operations central to all systems management. These minimum functionality includes ability to discover management resources, CREATE, DELETE, RENAME, GET and PUT individual management resources, ENUMERATE contents of containers and collections, SUBSCRIBE to events emitted by managed resources, and EXECUTE resource-specific management methods. Thus, the majority of overlapping areas with the WSDM specification are in the MUWS specification.

5.1 Implementation of WS-Management

In this section we describe our implementation of the WS-Management specification. Our choice of leveraging WS-Management was mainly motivated by the simplicity of WS-Management and also the ability to leverage WSE. We have been using the management architecture for modeling the management of a distributed brokering infrastructure (Gadgil 2006).

The WS-Management framework only defines the minimum required interactions; the application thus is free to extend beyond this minimum specification. Furthermore, a manageable endpoint is not required to support all interactions specified (such as GET, PUT, CREATE, DELETE, RENAME) but only those that make sense in the particular context of the application. Not all manageable resources would provide enumeration or the eventing model; however, it is required that if an application intends to support these models, then it must leverage the WS Enumeration and WSE specifications, respectively

5.1.1 Leveraged Specifications

WS-Management leverages the following specifications

1. WS–Addressing **(**Box 2004a**)** for referencing resource endpoints
2. WS–Transfer (Alexander 2004a) for providing the common minimum set of actions, namely GET, PUT, CREATE and DELETE. WS-Management defines an additional verb RENAME to allow renaming or resources.
3. WS–Enumeration (Alexander 2004b) for retrieving contents of large containers / collections / logs etc…
4. WS-Eventing to serve as a notification model on the publish / subscribe paradigm
5. SOAP version 1.2

We implemented WS-Transfer and WS-Enumeration while WSE (whose implementation was described in section 4) was leveraged from NaradaBrokering.

5.1.2 Implementation

WS Management relies heavily on the SOAP 1.2 specification, specifically for modeling faults. SAAJ (Soap Attachments API for Java) version 1.3 supports SOAP 1.2, however to maintain compatibility with other leveraged software, we implemented our own SOAP marshalling and un-marshalling framework using XMLBeans.

We also implemented our own prototype Web Service Engine that processes SOAP messages. Although the most commonly used method of transport is SOAP over HTTP, we plan to use the NaradaBrokering messaging substrate for delivering SOAP messages between endpoints. Our Web Service engine can use HTTP as well as NaradaBrokering's publish / subscribe mechanism for delivering and receiving events. We believe this to be a novel characteristic of our architecture since this allows us to seamlessly leverage a variety of features provided by the NaradaBrokering messaging substrate such as reliable delivery, exactly-once delivery and message-level security. Furthermore, NaradaBrokering supports a variety of transport protocols and tunneling through firewalls which allows us to apply the management architecture to remote resources.

5.1.3 Processing Messages

The message processing flow in our framework is depicted in **Figure 3**. In our architecture, the actual message transport is handled by the `Message Received` and `Send Response` units, which provide multiple ways of delivering messages. In the future, additional processing elements may be plugged in here to meet QoS requirements such as reliable delivery and security by utilizing appropriate Web Service specifications.

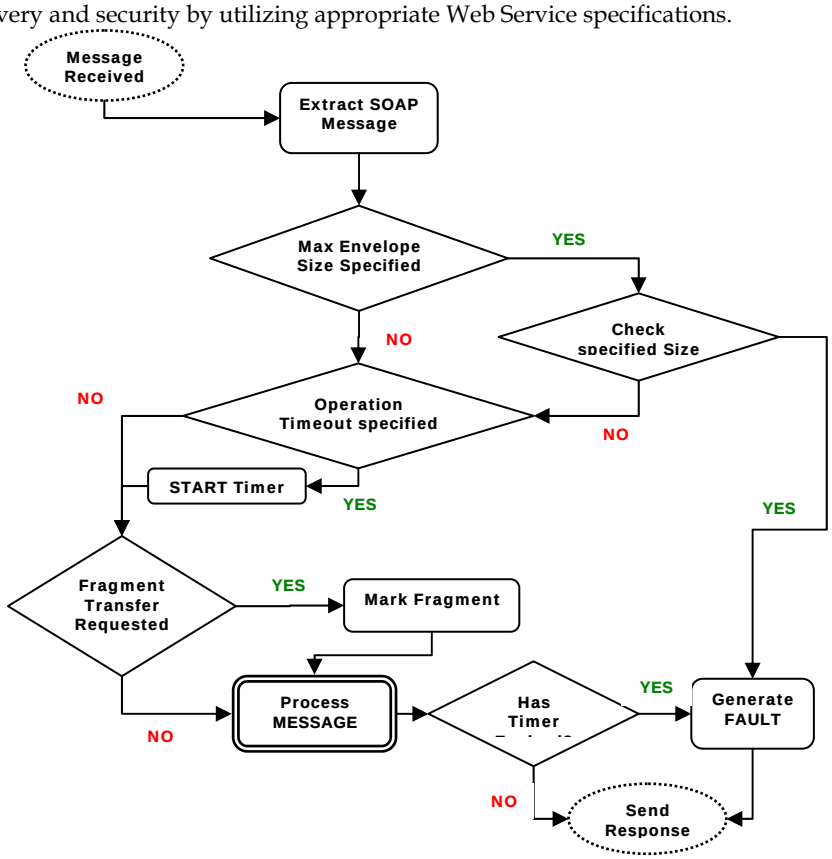

Figure 3 Flow Chart for processing WS Management Messages

The processing begins by checking the maximum envelope size for the response. WS-Management specifies that if such an element is present and a size is specified the minimum size must be 8192 octets to reliably encode all possible faults. This element may be discarded if `mustUnderstand` is set to `FALSE`; when this is `TRUE` and if the value is less than 8192, a fault is thrown.

An operation timeout may be specified to indicate that a response is desired within the specified timeframe. If specified, a timer is started. If a response is indeed generated (success OR failure) the timer is cancelled. However, if the timer expires before the processing has finished, a `TimedOut` fault is sent back to the requestor. WS-Management states that any state changes that may have occurred during the processing are later inspected by the requestor by making one or more `GET` requests to retrieve the service state.

Finally, the `Process Message` block is invoked. Depending on the characteristics of a particular resource, only a subset of operations may be supported. We describe each of the three main operations below.

5.1.4 WS-Transfer

Application developers can provide the ability to GET, PUT, CREATE, DELETE, and RENAME individual management resources by providing implementation for the abstract operation methods of the WS-Management processor. For example, to support a GET operation the developer simply implements the following method

```
public abstract void processWxfGet(EnvelopeDocument envelopeDocument,
                                   MessageHeaders headers,
                                   XmlFragmentDocument xmlFrag)
             throws WSManServiceException;
```

If a resource supports GET and PUT operations but does not support CREATE, DELETE and RENAME, then the developer has to throw an `UnsupportedFeature` fault for the unsupported verbs.

In our architecture, WS-Enumeration and WSE have been implemented by a completely different set of abstract classes. Thus, if a service wishes to provide an Enumeration capability, it can provide an implementation of Enumeration and register itself with the WS-Management processor. An `UnsupportedFeature` fault is immediately thrown if the WS-Enumeration/WSE processor is not registered and a corresponding request is received.

5.1.5 WS Enumeration

WS-Enumeration processing requires the system to do book-keeping of enumeration requests which are referred to as Enumeration Context. The WS-Enumeration processor maintains the context info (which could be a simple UUID). A service is free to extend the basic enumeration processor to provide additional service specific features. Every time a PULL or any other request is received, the WS-Enumeration processor checks the enumeration context and validates it. Validation process checks for expiry of the enumeration context and an `InvalidEnumerationContext` fault is automatically thrown.

5.1.6 Looking ahead

As part of the future work in this area we plan to investigate how we can incorporate WS Security in our present architecture. Service policies dictate the service requirements, capabilities and define the handling of management related requests. WS Policy (Bajaj 2006) provides a general purpose model and corresponding syntax to describe such policies. We are currently investigating the use of WS Policy in our work. Typically, a client may need to list available resources, obtain XML schemas or WSDL definitions or perform other discovery tasks. WS Management recommends WS Metadata Exchange (Ballinger 2004) for these tasks. In the future we will investigate incorporating a discovery processor to facilitate discovery of service-specific metadata. Recently the Web Services community announced the merger (Cline 2006) of WS-Management and WSDM specifications. We plan to investigate how to support the upcoming management specification within our current implementation.

6. Extending the WS-Context Specification:

Often Web Services are assembled into short-term service collections that are gathered together into a meta-application (such as a workflow) and collaborating with each other to perform a particular task. For example, an airline reservation system could consist of several Web Services, which are combined together to process reservation requests, update customer records, and send confirmations to clients. As these services interact with each other they generate session state which is simply a data value that evolves as result of Web Service interactions and persists across the interactions. As the applications, employing Web Service oriented architectures, need to discover, inspect, manipulate state information in order to correlate the activities of participating services a need arises for specifications that would standardize the management of distributed session state information.

The Web Service Context (WS-Context) Specification (Bunting2003) was introduced to define a simple mechanism to share and keep track of common context information shared between multiple participants in Web Service interactions. A participating application can also discover results (which is stored as context) of other participants' execution. The context here has information such as unique ID and shared data. It allows a collection of actions to

take place for a common outcome. The WS-Context Specification also defines a Web Service interface – the Context Manager, which in turn allows applications to retrieve and set data associated with a context.

There are other specifications, such as the Web Service Resource Framework (WSRF) (Czajkowski2004) and WS-Metadata Exchange (WS-ME) (Ballinger2004), that have been introduced to define stateful interactions among services. Among these existing specifications, which standardize service communications, we chose WS-Context Specifications to tackle the problem of managing distributed session state. Unlike the other service communication specifications, WS-Context models a session metadata repository as an external entity where more than two services can easily access/store highly dynamic, shared metadata.

We find various limitations in WS-Context Specification in supporting stateful interactions of Web Services. First, the Context Manager, a component defined by WS-Context to provide access/storage to state information, has limited functionalities such as the two primary operations: GetContext and SetContext. However, Grid applications present extensive metadata needs which in turn requires advanced search, access and store interfaces to distributed session state information. Second, the WS-Context Specification does not define an information model for the Context Manager component. So, there is a need for a data model to store the state information in persistent data structures. Third, the WS-Context Specification is only focused on defining stateful interactions of Web Services. However, there is a need for a specification which can provide an interface for not only stateful interactions but also the stateless and interaction-independent information associated with Web Services. In order to address these limitations, we have investigated XML Metadata Services that can be used as the Context Manager and that can provide a uniform programming interface to both stateless and stateful service metadata.

We designed and built a hybrid WS-Context compliant metadata catalog service (Aktas2005) supporting both handling and discovery of not only quasi-static, stateless metadata, but also session related metadata. We based the information model and programming interface of our system on two widely used specifications: Web Services Context, and Universal Description, Discovery, and Integration (UDDI) (Bellwood2003).

We utilized Xerces software (http://xerces.apache.org/xerces-j/) as the XML Schema Processor to process the extended version of WS-Context Specification Schema. Xerces is an open-source, high-performance XML parser component developed by the Apache XML Project. We have focused on the two base elements of the semantics and implementation of the proposed system: a) information model (data semantics), b) XML programming interface (semantics for publication and inquiry, security and proprietary XML API). The information model is composed of various entities, such as session, service, and context entities. These entities are the information holders; in other words, directories where distributed session state information is stored.

The programming interface of the hybrid WS-Context service introduces various additional publishing/discovery and information security capabilities. The additional XML API capabilities may broadly be categorized as: a) functions operating on dynamic, session-related metadata space, b) functions operating on static, stateless metadata space, c) hybrid functions operating on both metadata spaces and d) information security related functions. The dynamic metadata functions are used to enable the system to track the associations between sessions and contexts by expanding on primary functionalities of WS-Context XML API set. The static, interaction-independent metadata functions are used to provide a programming interface to the static metadata space. These functions simply forward the incoming SOAP messages to the extended UDDI XML metadata service for handling of query and publishing requests. The hybrid functions provide publishing/discovery capabilities supporting both dynamic and static service metadata. These functions integrate results coming from the two separate metadata spaces. The information security related functions provide an authentication and authorization mechanism, as the shared information may not be open to anyone. Further design documentations are also available at http://www.opengrids.org/wscontext.

7. Extending the UDDI Specification:

As SOA principles have gained importance, there is a need for methodologies to locate desired services that provide access to their capability descriptions. Geographical Information Systems (GIS) provide very useful problems in supporting "virtual organizations" and their associated information systems. These systems are composed of various archival data services (Web Feature Services), data sources (Web-enabled sensors), and map generating services. Organizations like the Open Geospatial Consortium (OGC) define the metadata standards. All of these services are metadata-rich, as each of them must describe their capabilities (What sorts of features do they provide? What geographic bounding boxes do they support?) This is an example of the very general problem of managing information about Web Services.

One approach to service-metadata management problem is the Universal Description, Discovery and Integration (UDDI) specification. UDDI is domain-independent standardized method for publishing/discovering information about Web Services. It offers users a unified and systematic way to find service providers through a centralized

registry of services. As it is WS-Interoperability (WS-I) compatible, UDDI has the advantage being interoperable with most existing Grid/Web Service standards.

We observed that the adoption of UDDI Specification in various domains such as GIS is slow, since existing UDDI specification has following limitations. First, UDDI introduces keyword-based retrieval mechanism. It does not allow advanced metadata-oriented query capabilities on the registry. Second, UDDI does not take into account the volatile behavior of services. Since Web Services may come and go, and since the information associated with services might be dynamically changing, there may be stale data in registry entries. Third, since UDDI is domain-independent, it does not provide domain-specific query capabilities in particular for GIS domain such as spatial queries. There is a need for integration between the OGC metadata standards and the UDDI Service information model.

In order to provide solutions to these limitations, various solutions have been introduced. UDDI-M (Dialani2002) and UDDIe (ShaikhAli2003) projects introduce the idea of associating metadata and lifetime with UDDI Registry service descriptions where retrieval relies on the matches of attribute name-value pairs between service description and service requests. UDDI-MT (Miles2003, Miles2004) improves the metadata representation from attribute name-value pairs into RDF triples. A similar approach to leveraging UDDI Specifications was introduced by METEOR-S (Verma2003) project, which identifies different semantics when describing a service such as data, functional, quality of service and executions.

As an alternative solution, we designed an XML Metadata Service to provide a solution to the general problem of managing static, stateless information about Web Services. The prototype was implemented as a domain-independent metadata service to meet the information requirements of the different application domains. To support the specific metadata requirements of Geographical Information Systems, this prototype implementation was further extended to support geospatial queries on the metadata associated to service entries. We used the UDDI specifications in our design. We also designed extensions to existing UDDI Specifications as outlined earlier. Similar to previous solutions, we too extend UDDI's Information Model, by providing an extension where we associate metadata ((name, value) pairs and life-time with service descriptions. Apart from the existing methodologies, we provide various advanced capabilities such as domain-independent and GIS-domain-specific query/publishing capabilities as well as dynamic aggregation and searching of geospatial services. We based the implementation of our design on jUDDI (http://ws.apache.org/juddi, version 0.9r3), a free, open source, and Java-based implementation of the UDDI specification. We used Xerces as the default XML Schema Processor.

We have expanded on the two base elements of the existing semantics of UDDI Specifications: a) information model (data semantics) and b) XML programming interface (semantics for publication and inquiry XML API). The extended UDDI information model included service attribute and service entities. Its programming interface provides metadata-oriented publishing/discovery capabilities by expanding on existing UDDI XML API set. The additional XML API set introduced various capabilities such as a) publishing additional metadata associated with service entries, b) posing metadata-oriented, geospatial, and domain-independent queries on the extended UDDI service. The domain-independent search capability is a more general purpose extension to the UDDI data model that allows us to insert arbitrary XML metadata into the repository. This may be searched using XPATH queries, which is a standard way of searching XML documents. This allows us to support other XML-based metadata descriptions developed for other classes of services besides GIS. The Globus/IBM-led WSRF effort is an important example.

8. Web Services and Mobile Devices

In this section we address issues related to the use of Web Services in mobile computing. Despite the fact that there have been several advances in the area of mobile computing in recent years, applying current Web Service communication models to mobile computing may result in unacceptable performance overheads. Two main factors contribute to these problems. First, the encoding and decoding of verbose XML-based SOAP messages consumes scare CPU resources on the device. Second, the performance and quality gap between wireless and wired communication will not close quickly. These issues are depicted in Figure 4.

The use of XML to describe data invariably results in increases in the size of the final representation. This size increase can be as high as an order of magnitudes, if the document structure is especially redundant – for example in the case of arrays. Encoding data into a SOAP message requires a text-conversion, where the in-memory representation is converted into a textual format. The decoding process does the reverse work; if the data is non-textual, such as a floating point number, the conversion is very expensive in terms of performance overhead, which is especially significant for relatively low-powered mobile devices. We have designed a framework – *Handheld Flexible Representation (or HHFR)* – that addresses these issues in the context of mobile computing.

HHFR works best for Web Services, where the two participating nodes exchange a series of messages, which we define as a stream. For applications using a specific service repeatedly, messages in the stream have the same structure and the same data-type for information items. Most of the message headers are unchanged in the stream.

Therefore, the structure and type of SOAP message contents and unchanging SOAP headers may be transmitted only once, and rest of the messages in the stream have only payloads.

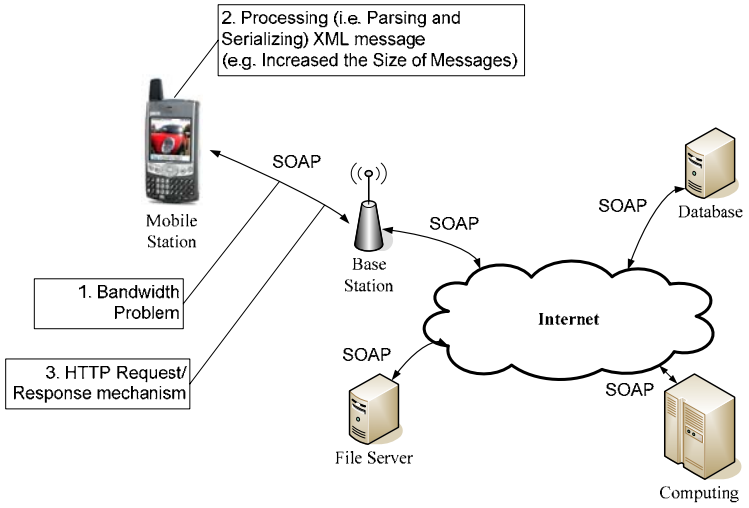

Figure 4: Potential Problems in using Conventional Web Services on Mobile devices

The HHFR implementation leverages the Data Format Description Language (DFDL) for negotiation representations, message streaming, and uses a metadata repository to store redundant or unchanging static data. A normal session1 of the runtime system is as follows. First, an HHFR-capable endpoint sends a negotiation request to the intended endpoint. The negotiation request is a conventional SOAP message that includes characteristics of the stream to be used for communications. The negotiation decides on the optimal representation of messages within the stream. Subsequent stream messages are then exchanged based on this negotiated representation. The unchanging parts (static metadata) of the streamed messages are stored into a dynamic metadata repository during the session. If the service endpoint is not HHFR-capable communications revert to conventional SOAP messaging. This strategy where only the fragments of data that have changed for successive messages are exchanged accrues several benefits in the utilization of CPU and networking resources at the mobile device.

9. Deployment Related Issues within Axis

Axis is the most dominant Web Services container within the Java domain, and is used within most Java-based Web Services. In this section, we describe issues within the Axis container vis-à-vis interactions mandated within Web Service specifications. These issues pertain to Axis version 1.2RC3. A comprehensive discussion of these issues, and results related to the workarounds to these problems can be found in [Yildiz 2006]

In addition to the services themselves, several containers (including Axis) incorporate support for Handlers or Filters which facilitate incremental addition of capabilities at a service endpoint. An example of a Handler is an encryption handler which encrypts messages originating from a client and an inverse-Handler at the Service side which performs the appropriate decryption. By setting up appropriate Handlers (and the corresponding inverse Handlers) in the request and response flows originating from a service endpoint, that endpoint's capability is enhanced without the need for making changes to the application. One typically configures Handlers through a deployment descriptor file that is part of the Web Service container. Finally, several Handlers could be cascaded together to comprise a Handler Chain. A given handler provides the natural location for setting up a role associated within a given WS-* specification. The WSRM sink role can be configured in a handler within the handler-chain at the destination service endpoint. Although the Axis architecture provides very good functionalities, there are several areas where we see a need for improvement. We enumerate this below.

1. Within Axis currently only the Clients are allowed to inject (or initiate) messages.
2. In Axis every message is considered a request which should have its accompanying response within a pre-defined period of time. This does not fit very well with interactions where no responses are issued.

[1] We use this term to refer to application session which may have one or more streams in it. In HHFR, the session includes a conventional SOAP message exchange for a negotiation and flexible representation message exchanges through high performance channel stream.

3. No ability to gracefully terminate processing related to a message within the Handler chain associated with a service.
4. Handlers cannot initiate messages on their own.
5. Static configuration of the handler chain.

9.1 Message Initiation

Clients are the only entities that are able to initiate messages as requests. Server-side components, either a handler in the handler-chain or the target service, do not have this capability. The only message initiation is via a request/response mechanism where requests can be initiated only by the clients. There are several scenarios that indicate the need for message initiation in the server part.

Consider the case of `Acknowledgements` in WSRM which requires message initiation from the WSRM Sink to the Source. The WSRM specification requires endpoints to comply with the acknowledgement and retransmission intervals that are exchanged prior to the creation of a Sequence. This implies that in acknowledgements may be issued several seconds after the receipt of a message. Furthermore, a single acknowledgement may encapsulate information pertaining to the receipt of several thousand messages. The limitations within the request-response paradigm are clear in such scenarios. It is clear that one-way messaging and message-initiation would resolve this particular problem. The problems outlined here also arise during retransmissions initiated by a WSRM source. WSE is another case where message initiation is needed. A message may need to be reproduced to send the copies to multiple subscribing endpoints.

9.2 Other request-response based problems

Sometimes an entity may need to send a message to another entity without the need to receive a response. For example, when we issue a WSRM acknowledgement we are not looking for an acknowledgement to the WSRM-acknowledgement.

9.3 Ability to terminate processing related to Message within a Handler chain

Currently there is no ability to gracefully terminate processing related to a message within the Handler chain associated with a service. This implies that once a message has been received within a handler chain there is no graceful way to prevent this message from reaching the Service. A good example of the need for this feature would be the case of acknowledgments in WSRM. Only the WSRM handler needs to know whether the WSRM sink has received the message that was previously sent; there is no need to inundate the application with every acknowledgement that has been received. Also, since most WS specifications are aimed at incrementally adding functionality to a service, situations may arise where similar such handshakes/control-messages should be stopped from reaching the actual service. In most cases such messages end up causing problems at the service. Currently the only way to do so is to terminate processing is to throw an Exception. Furthermore, there is no way to correctly access or interact with the handler-chain managing a specific handler.

9.4 Handlers cannot inject messages

This issue was mentioned earlier too, but needs to be clarified in the case of handlers too. Most WS specifications are naturally implemented as handlers that can augment a service's capabilities. However, such handlers need to be able to initiate control-messages on their own accord. While there is access to the Handler Chain there is no access to message propagation features. This feature may be made available through the `AxisEngine` class or through the handler chain itself.

9.5 Static Handler chains

In Axis handlers are currently statically configured – the configuration being made when a service is being deployed. A handler can not be added or removed from a service. Current Axis architecture allows cloning the handler chain, the cloned chain can then replace the current one after required changes have been applied; however, this is not sufficient. Dynamic configuration of handler chains will be very useful. A deployment may have lots of handlers but a user/handler should be able to select a group of handlers that a message would need to go through.

This is especially true in cases of retransmissions where the handler may have already processed the message resulting in duplicate processing which may or may not lead to errors. Similarly, security requirements may result in a message be passed through a different set of handler chains – here handlers related to message digest, encryption and signing may be added to the outgoing path in the handler chain.

9.6 Solutions to some of the problems

We have workaround solutions for some issues above; instead of blocking a message, a message can be set to a dummy task. We had to choose this method because we wanted to stop the propagation of the message without getting an exception. By letting the message arrive at the endpoint we have prevented an exception being thrown. On the other hand, there exists a downside to this solution. This adds to performance costs because of the processing overheads for the dummy task. The costs are acceptable within the current Axis architecture.

Although messages can not be initiated by the server part, we can bypass this restriction by using an Axis client wherever a message initiation is required. In this architecture, service endpoints will have client and server capability. This results in both the endpoints being maintained within a container.

Next, we developed a sender thread which is responsible for sending any message to the other node. Since we are using one-way messaging, we assign the responsibility of sending messages to this thread instead of dealing with the Axis message response mechanism. The thread enables us to send messages in a non-blocking fashion.

We did not use a response chain because we decided on building our own chain structure. After some point in the response handler chain, we diverted the massage path to the sender thread and added our own handler chain. This allowed us to do one-way messaging without breaking any of the already deployed handler structure at the service; this can easily support dynamic handler chains.

10. Use cases for various specifications

In this section we provide examples of the deployments of several of the specifications that have been discussed in this article.

10.1 WSE, WSRM and WSR

The WSE, WSRM and WSR specifications are part of the OMII Container (3.0.0) available for download from http://www.omii.ac.uk/. All of these specifications leverage the WS-Addressing specification. Furthermore, to support exchanges outlined within these specifications, while coping with the constraints within several containers, we deployed strategies outlined in sections 9.6.

10.2 WSE, WS-Management, WS-Transfer

As discussed in section 5.0 the WS-Management leverages the WSE specification for its notification needs. We have leveraged the WS-Management specification implementation as a framework for managing distributed resources. Our specific use case consists of managing the NaradaBrokering messaging middleware, which consists of a large number of dynamic peers – messaging brokers.

The messaging brokers require resource-specific configuration such as ports and a global configuration such as interconnections that make a specific broker topology. Run-time metrics are gathered via monitoring techniques or obtained via runtime events generated by the resource. We measure various aspects of the system that enable us to understand the performance of the system and, in some cases, provide hints on improving the performance. This naturally leads to re-deployment of the brokering network with a different configuration. To summarize, we need an architecture that enables us to rapidly bring-up and tear down a broker network. It is also required to set specific configuration settings for every broker and have the ability to change the configuration on-the-fly. We term these actions collectively, as management of the brokering infrastructure.

To aid the management of brokers and broker networks, we have modeled several resource specific operations which have been summarized in Table 1.

Table 1: Broker Network Management: Summary of operations and specifications leveraged

Operation / Event	Functionality provided	What part of specification has been leveraged
Get/Set Configuration	Reads / Writes the broker specific configuration	WS Transfer – GET / PUT
Create / Delete Broker	Instantiates a new instance of the Broker / Shuts down an existing instance of broker	WS Transfer – CREATE / DELETE
Create / Delete Link	Creates a link between two brokers/ Deletes an existing Link	WS Transfer – CREATE / DELETE
Link Lost Exception	If a broker detects that an outgoing link was broken, this exception is thrown	WS Eventing – Event notification

10.3 Extended UDDI and hybrid WS-Context

Extended UDDI and hybrid WS-Context services have been used as the metadata management components of various application use domains. The first example is a workflow session metadata manager, a vital component of workflow-style Grid applications. A workflow session metadata manager is responsible for providing store/access/search interface to metadata generated during workflow execution. The second example is a metadata catalog service. A catalog service is a metadata service that stores both prescriptive and descriptive information about Grid/Web Services. The third example is a third-party metadata repository, also called as Context-store. This component is used in a fast web service communication model in collaborative mobile computing environments where the redundant parts of the exchanged messages are stored.

10.3.1 The workflow session metadata manager

The hybrid WS-Context service is being used as the workflow session metadata manager in two practical example usage domains: Pattern Informatics and The Interdependent Energy Infrastructure Simulation System (IEISS). Pattern Informatics, a technique to detect seismic activities and make earthquake predictions, was developed at University of Southern California at Davis. The Pattern Informatics GIS Grid (Aydin2005) integrates the Pattern Informatics code with publicly-available, Open GIS Consortium(OGC)-compatible, geo-spatial data and visualization services. The Interdependent Energy Infrastructure Simulation System is a suite of analysis software tools developed by Los Alamos National Laboratory (LANL). IEISS provides assessment of the technical, economic and security implications of the energy interdependencies (LANL2006). The IEISS GIS Grid, a workflow-style GIS Grid application developed at LANL, supports IEISS analysis tools by integrating them with openly available geo-spatial data sources and visualization services. Both Pattern Informatics and IEISS systems need an Information Service, which can be utilized as the workflow session metadata manager. In these applications, participants of a workflow must know about the state of the system, so that they can perform their assigned tasks within a specific sequence.

The hybrid WS-Context Service is used as a workflow session metadata manager, which is responsible for storing transient metadata, needed to describe distributed session state information in a workflow. The hybrid WS-Context service allows users to access session state information by either pull or push based approaches. In pull-based approach, each participant continuously checks with the system if the state is changed. For instance, some application domains may employ various browser-based applications; here, pushing the states to the web-applications through the HTTP server is rather complicated. So, the pull-based approach can be used in those domains to interact with the service to get the state updates. In push-based approach, participants are notified of the state changes. The push-based approach is mainly used to interact with the workflow session metadata manager in order to reduce the server load caused by continuous information polling.

10.3.2 The metadata catalog service

The two applications: Pattern Informatics and IEISS GIS systems are composed of various GIS compatible data and map generating Grid/Web services. Thus, both of these application domains need a metadata catalogs service, which would provide a unified and systematic way to find service through a registry of services. The extended UDDI metadata service is used as the Metadata Catalog Service which is responsible for providing access/store interface to both prescriptive and descriptive metadata about services. GIS-based Grid applications are composed of various archival services, data sources, and visualization services. Services such as the Web Map (OpenGIS2006a) and Web Feature (OpenGIS2006b) service, because they are generic, must provide additional, descriptive metadata in order to be useful. The problem is simple: a client may interact with two different Web Feature Services in exactly the same way (the WSDL is the same), but the two Web Feature Services may hold different data. One, for example, may contain GPS data for the Western United States while the other has GPS data for Northern Japan. Clients must be able to query information services that encode (in standard formats) all the necessary information, or metadata, that enables the client to connect to the desired service. Thus, we see the need for a metadata catalog service, which would manage metadata associated to all these Grid/Web Services, and make them discoverable. A client should be able to get "capabilities" metadata file either from the service itself or from the metadata catalog. Thus, these metadata catalog services are also expected to have a dynamic metadata retrieval capability, which enables the system to dynamically retrieve the capability metadata file from the service under consideration. The extended UDDI service introduces capabilities addressing the metadata management requirements of the GIS domain.

10.3.3 The Context-store for high performance SOAP

The Handheld Flexible Representation (HHFR) is an application designed to provide efficient and optimized message exchange paradigm in mobile Web Service environment (Oh2005). The HHFR architecture provides layers, which optimize and stream messages to achieve high performance mobile Web Service communication. The HHFR system utilizes the hybrid WS-Context services as a third-party repository, (i.e. Context-store) to store the redundant/unchanging parts of the messages exchanged between services. This way the size of the exchanged messages can be reduced to achieve optimized Web Service communication. A Context-store component is a metadata service responsible for storing redundant/unchanging parts of SOAP messages exchanged in service communication.

11. Conclusion

In this chapter we provided a discussion of the various Web Service specifications that we have implemented. We are hopeful that the strategies (and implementations) outlined in this chapter can be leveraged by other developers. In this chapter we have touched upon several aspects of building distributed applications using Web Services: these include targeting (WS-Addressing), content distribution (WSE), guaranteed messaging (WSR and WSRM), the management of resources (WS-Management and the suite of specifications it leverages), context (modified WS-Context) and discovery (extended UDDI). To leverage Web Services within compute and power constrained devices such as PDAs we included a discussion of HHFR. We have also summarized use cases for these various specifications in the preceding section.

Though these specifications serve different functions, it is possible to build systems that leverage several of these specifications. For example, in the case of managing a distributed messaging system we leveraged several Web Service specifications such as WSE, WS-Management, WS-Addressing, WS-Enumeration and WS-Transfer. This work shows how software services can be managed using a Web Service management protocol. This is important as it lays the groundwork for applications to be very adaptive to changing needs. For example, one may dynamically switch protocols (e.g. TCP, UDP, ParallelTCP) at runtime depending on the application being managed such as audio-video conferencing or GIS-based Grid applications.

While implementing these WS-* specifications we found that in some cases the dominant containers have not kept pace with some of the specifications; specifically this pertained to processing SOAP messages within a container. Here, workarounds had to be developed to cope with the constraints within the containers.

One of the problems that we encountered while developing these specifications was that they would change quite often: sometimes, this resulted in us chasing moving targets with the specification evolving every few weeks. Each new specification has its own unique schema, so an architect has to decide which schema to settle on and proceed from thereon. There have been significant improvements in this area as specifications have matured. Another area which needs to be addressed very soon is the presence of multiple specifications in the same domain. The functionality provided in these competing specifications tends to be more or less the same. This makes it particularly difficult for a systems designer to decide which specification to bet on. Having to cope with multiple specifications in the same domain increases *complexity* and raises *interoperability* issues –- problems which Web Services started out to solve in the first place.

Problems with WS-* specifications notwithstanding, Web Services have immense potential and offer significant benefits. Web Services facilitate the development of loosely-coupled, asynchronous, and interoperable systems. WS-* specifications can be looked upon as a building blocks for the development of distributed applications. Using implementations of these WS-* specification a systems designer is able to quickly put together a robust, distributed application.

References

- [Aktas2005] Aktas, M.S., Fox, G. C., and Pierce, M. E., (2005), Managing Dynamic Metadata as Context, in Istanbul International Computational Science and Engineering Conference (ICCSE2005 http://www.iccse.org/)
- [Alexander 2004a] Alexander, J., et. al. (2004). Web Service Transfer (WS – Transfer), from http://msdn.microsoft.com/library/en-us/dnglobspec/html/ws-transfer.pdf
- [Alexander 2004b] Alexander, J., et. al. (2004). Web Service Enumeration (WS – Enumeration), from http://msdn.microsoft.com/library/en-us/dnglobspec/html/ws-enumeration.pdf
- [Arora] Arora, A., et. al. (2005). Web Service Management (WS – Management), Available from https://wiseman.dev.java.net/specs/2005/06/management.pdf
- [Aydin2005] Galip Aydin et al (2005). A. SERVOGrid Complexity Computational Environments (CCE) Integrated Performance Analysis. In the proceedings of the 6th IEEE/ACM International Workshop on Grid Computing. 2005.

- [Bajaj 2006] Bajaj, S., et. al. (2006). Web Services Policy Framework (WS-Policy) from http://specs.xmlsoap.org/ws/2004/09/policy/ws-policy.pdf
- [Ballinger2004] Ballinger, K., et al., (2004), The Web Services Metadata Exchange Specification, http://specs.xmlsoap.org/ws/2004/09/mex/WS-MetadataExchange.pdf.
- [Bellwood2003] Bellwood, T., Clement, L., and Riegen, C., (2003), UDDI Version 3.0.1: UDDI Spec Technical Committee Specification, http://uddi.org/pubs/uddi-v3.0.1-20031014.htm.
- [Bilorusets 2004] Ruslan Bilorusets et al. Web Services Reliable Messaging Protocol (WS-ReliableMessaging) http://www-128.ibm.com/developerworks/library/specification/ws-rm/
- [Box 2004a] Box. D et al. Web Services Addressing (WSAddressing) http://www.w3.org/Submission/ws-addressing.
- [Box 2004b] Box, D et al. Web Services Eventing. Microsoft, IBM & BEA. http://ftpna2.bea.com/pub/downloads/WS-Eventing.pdf
- [Bunting2003] Bunting, B., et al Web Services Context (WS-Context) version 1.0 http://www.arjuna.com/library/specs/ws_caf_1-0/WS-CTX.pdf.
- [Case 1990] Case, J., Fedor, M., Schoffstall, M., & Davin, J. (1990). Simple Network Management Protocol, Network Working Group Request for Comments 1157, from http://www.ietf.org/rfc/rfc1157.txt
- [Christensen 2001] Erik Christensen et al Web Services Description Language (WSDL) 1.1 http://www.w3.org/TR/wsdl
- [Cline 2006] Cline, K., et. al. (2006). Toward Converging Web Service Standards for Resources, Events, and Management, from http://msdn.microsoft.com/library/en-us/dnwebsrv/html/convergence.asp
- [Czajkowski2004] Czajkowski, K., Ferguson, D., Foster, I., Frey, J., Graham, S., Sedukhin, I., Snelling, D., Tuecke, S., Vambenepe, W., (2004), The WS-Resource Framework, available at http://www.globus.org/wsrf/specs/ws-wsrf.pdf.
- [Dialani2002] Dialani, V., (2002), UDDI-M Version 1.0 API Specification, Southampton, University of Southampton, UK.
- [Gibbs 2003] Kevin Gibbs, Brian D Goodman, IBM Elias Torres. Create Web services using Apache Axis and Castor. IBM Developer Works. http://www-106.ibm.com/developerworks/webservices/library/ws-castor/
- [Gudgin 2003] M. Gudgin, et al, "SOAP Version 1.2 Part 1: Messaging Framework," June 2003. http://www.w3.org/TR/2003/REC-soap12-part1-20030624/
- [Gadgil 2006] Gadgil, H., Fox, G., Pallickara, S. & Pierce, M. (2006). Managing Grid Messaging Middleware Challenges of Large Applications in Distributed Environments, Paris, France
- [HP 2005] HP (2005). Web Service Distributed Management (WSDM) from http://devresource.hp.com/drc/specifications/wsdm/index.jsp
- [LANL2006] LANL, Los Alamos National Laboratory, The Interdependent Energy Infrastructure Simulation System (IEISS) project, web site is available at http://www.lanl.gov/orgs/d/d4/interdepend.
- [Miles2003] Miles, S., et al., (2003), Personalized Grid Service Discovery, Nineteenth Annual UK Performance Engineering Workshop (UKPEW'03), University of Warwick, Coventry, England.
- [Miles2004] Miles, S., Papay, J., Payne, T., Decker, K., Moreau, L., (2004), Towards a Protocol for the Attachment of Semantic Descriptions to Grid Services, In The Second European across Grids Conference, Nicosia, Cyprus.
- [Oasis-WSR 2004] OASIS. Web Services Reliability TC WS-Reliability. http://www.oasis-open.org/committees/download.php/ 5155/WS-Reliability-2004-01-26.pdf
- [OGC] OGC The Open Geospatial Consortiom (OGC), web site available at http://www.opengis.org.
- [Oh2005] Oh, S., et al. Optimized communication using the SOAP infoset for mobile multimedia collaboration applications. Proceedings of the International Symposium on Collaborative Technologies and Systems 2005.
- [OpenGIS2006a] Open Geospatial Consortium Inc., OpenGIS Web Map Service (WMS) Specification available at http://www.opengeospatial.org/standards/wms. 2006.
- [OpenGIS2006b] Open Geospatial Consortium Inc., OpenGIS Web Feature Service (WFS) Specification available at http://www.opengeospatial.org/standards/wfs. 2006.
- [Pallickara 2005] Shrideep Pallickara et al. On the Costs for Reliable Messaging in Web/Grid Service Environments. Proceedings of the 2005 IEEE International Conference on e-Science & Grid Computing. Melbourne, Australia. 344-351.
- [ShaikhAli2003] ShaikhAli, A., Rana, O., Al-Ali, R., Walker, D., (2003), UDDIe: An Extended Registry for Web Services. Proceedings of the Service Oriented Computing: Models, Architectures and Applications, SAINT-2003 IEEE Computer Society Press. , Orlando Florida, USA.
- [Verma2003] Verma, K., Sivashanmugam, K., Sheth, A., Patil, A., Oundhakar, S. and Miller, J., (2003), METEOR–S WSDI: A Scalable P2P Infrastructure of Registries for Semantic Publication and Discovery of Web Services, Journal of Information Technology and Management.
- [Warrier 1990] Warrier J., Besaw L., LaBarre L., Handspicker, B. (1990). The Common Management Information Services and Protocols for the Internet, Network Working Group Request for Comments 1189, from http://www.ietf.org/rfc/rfc1189.txt
- [Yildiz 2006] Beytullah Yildiz, Shrideep Pallickara and Geoffrey Fox. Experiences with deploying services within the Axis container. Proceedings of the 2006 IEEE International Conference on Internet and Web Applications and Services. French Caribbean.