On the Costs for Reliable Messaging in Web/Grid Service Environments

Shrideep Pallickara, Geoffrey Fox, Beytullah Yildiz, Sangmi Lee Pallickara, Sima Patel and Damodar Yemme
(spallick, gcf, byildiz, leesangm, skpatel, dyemme)@indiana.edu
Community Grids Lab, Indiana University.

Abstract

As Web Services have matured they have been substantially leveraged within the academic, research and business communities. An exemplar of this is the realignment, last year, of the dominant Grid application framework ─ Open Grid Services Infrastucture (OGSI) ─ with the emerging consensus within the Web Services community. Reliable messaging is an important component within the Web Services stack. There are two competing, and very similar, specifications within this domain viz. WS-ReliableMessaging (WSRM) and WS-Reliability (WSR); this work focuses on the WSRM specification. In this paper we provide an overview of the WSRM protocol, describe our implementation of WSRM, and present an analysis of the costs (in terms of latencies and memory utilizations) involved in the use of WSRM. Since WSRM is very similar to WS-Reliability we expect the performance of WSRM to be very similar to that of WSR. We hope that the work presented here helps researchers and systems designers gauge the suitability of Web Services based reliable messaging in their applications and also to make appropriate trade-offs, which includes inter alia interoperability, guarantees, quality of service and performance.

Key words: Web Services, WS-ReliableMessaging, guaranteed messaging, performance analysis

1. Introduction

The emerging Web Services stack comprising XML – the *lingua franca* of the various standards, SOAP [1] and WSDL [2] have enabled sophisticated interactions between services. WSDL describes message formats and message exchange patterns for services using XML, while services/entities interact through the exchange of SOAP messages. 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.

As Web Services have matured, reliable messaging (also known as guaranteed messaging) between service endpoints has become increasingly important. When service endpoints interact with each other, a need may arise to ensure that the interactions are routed reliably between them. In addition to the delivery guarantee these endpoints may also need assurances on the ordered delivery of these interactions. Finally, these delivery and ordering guarantees need to be met irrespective of the underlying transport used for communications. This area of reliable messaging now has two competing, though very similar, specifications: WS-ReliableMessaging [3] (hereafter WSRM) and WS-Reliability [4] (hereafter WSR). These specifications facilitate incremental addition of reliable messaging capabilities to the service endpoints. By funneling interactions through the implementations of these specifications, services can automatically inherit capabilities related to reliable and ordered delivery.

This complex area was previously was previously being addressed within the Web Services community using homegrown, proprietary, application specific solutions. This prevented interoperation with services outside the organization in which these protocols were implemented. Another related issue was the lack of an open-review process which would typically ferret out any bugs in the underlying protocol. The aforementioned specifications in the area of reliable messaging address both these areas.

Both these specifications have been exhaustively reviewed and deployed in a wide variety of settings. These wide deployments, and ease of use, have ensured that applications can more easily interoperate with each other. By eliminating the need for system designers to come up with proprietary solutions to the reliable delivery problem, researchers/designers can instead focus their efforts on the core problem within their respective application domains.

The work outlined in this paper focuses on WSRM. The over arching goal of this paper is to provide researchers and system designers of the costs involved in leveraging the WSRM specification. Since the WSR specification is very similar to the WSRM specification we expect these costs to be representative of those involved in the WSR specification too. By costs we refer to the CPU bound latencies and memory utilizations involved in the benchmarked operations. Application needs vary and there is never a one-size-fits-all solution: costs involved in WSRM may be acceptable in some cases and prohibitive in others. We hope this work can be used by researchers and system designers to make informed decisions about their Web Services reliable messaging strategy. Specifically, researchers can use this work to decide the tradeoffs involved in interoperability and ease-

of-use engendered in the WSRM specification to the acceptability of the costs involved in WSRM. In some cases the complexity — and concomitant time constraints — of developing an optimized (application/domain specific) reliable messaging protocol may be considered vis-à-vis the ability to simply plug-in a well-tested solution.

The remainder of this paper is organized as follows. In section 2 we include a brief overview of the WSRM specification. Section 3 describes our implementation strategy. We describe our performance measurements in section 4, with related work being described in section 5. Finally, in section 6 we outline our conclusions and future work.

2. WSRM

WSRM describes a protocol that facilitates the reliable delivery of messages between two web service endpoints in the presence of component, system or network failures. WSRM facilitates the reliable delivery of messages from the source (or originator) of messages to the sink (or destination) of messages. The delivery (and ordering) guarantees are valid over a group of messages, which is referred to as a sequence.

In WSRM prior to ensuring reliable delivery of messages between the endpoints, the source initiates an exchange with the sink pertaining to the creation of a Sequence. This Sequence is intended to facilitate the grouping of a set of related messages. This Sequence is identified by an identifier, typically a UUID. Other information associated with the Sequence include information regarding —

- The source and the sink
- Policy information related to protocol constants such as acknowledgement and retransmission intervals.
- Security related information if needed.

In WSRM all messages issued by a source exist within the context of a Sequence that was established prior to communications. Once a source has determined that all messages within a Sequence have been received at the sink, the source initiates an exchange to terminate this sequence. The specification allows for a maximum of 2^{64} - 1 messages within a Sequence. In the unlikely event that this number has been reached, a new Sequence needs to be established. The specification places no limits on the number of Sequences between a specific source and sink. However, it is expected that at any given time there is NO more than 1 active Sequence.

Every message from the source contains two pieces of information — the Sequence that this message is a part of and a monotonically increasing Message Number within this Sequence. These Message Numbers enable the tracking of problems, if any, in the intended message delivery at a sink by enabling the determination of out of order receipt of messages as well as message losses.

In WSRM a sink is expected to issue acknowledgements back to the source upon receipt of messages. This acknowledgement contains information pertaining to both the Sequence and the Message Numbers within this Sequence. An acknowledgement must be issued only after a certain time — the acknowledgement interval — has elapsed since the receipt of the first unacknowledged message. This acknowledgement may cover a single message or a group of messages within a Sequence. Upon receipt of this acknowledgement a source can determine which messages might have been lost in transit and proceed to retransmit the *missed* messages. Thus if a sink has acknowledged the receipt of messages `1 — 10` and `13 — 18`, the source can conclude that messages with Message Numbers `11` and `12` were lost en route to the sink and proceed to retransmit these messages.

A source may also pro-actively initiate the retransmission of a message for which that an acknowledgement has not been received within a specified time — the *retransmission interval* — after which it was issued. In WSRM error corrections can also be initiated at the sink; this is done through the use of negative acknowledgements which identify the message numbers that have not been received at a sink. Since Message Numbers increase monotonically, if Message Numbers `1,2,3,4` and `8` within a specific Sequence have been received at a sink, this sink can easily conclude that it has not received messages with message numbers `5,6` and `7` from the source.

WSRM provides for notification of errors in processing between the endpoints involved in reliable delivery. The range of errors can vary from an inability to decipher a message's content to complex errors pertaining to violations in implied agreements between the interacting source and sink.

2.1 Specifications leveraged by WSRM

WSRM leverages other specifications such as WS-Addressing [5] and WS-Policy [6]. WS-Addressing is a way to abstract from the underlying transport infrastructures the addressing needs of an application. WS-Addressing incorporates support for end point references (EPRs) and message information (MI) headers. EPRs standardize the format for referencing a Web service and Web service instances.

Before we proceed further, we make a brief note on the notation that we will be using throughout this paper. The notation `wsx:widget` corresponds to the schema element widget within the wsx specification's schema. Thus, the `wsa:To` element corresponds to the `To` schema element within the WS-Addressing schema, while the

wsrm:CreateSequence corresponds to the CreateSequence element within the WSRM schema.

The MI headers standardize information related to the origin (wsa:From) and destination (wsa:To) of message. It also standardizes elements that identify where replies and faults resulting from a message should be sent. Another element the wsa:Action element snapshots the semantic intent of message, while the wsa:RelatesTo element helps describe the relationship of a message to prior messages. Besides, the use of WS-Addressing for describing the source and the sink, WSRM also leverages fault reporting headers to report problems in the processing messages. Every message within WSRM has a unique identifier, typically a UUID, which is carried within the wsa:Message-ID information header. Finally, wsa:Action is also leveraged by WSRM to indicate the semantic intent of control messages such as Creation and Termination of sequences, and also Faults carried within the SOAP messages.

WSRM uses WS-Policy to exchange information regarding protocol constants such as acknowledgement intervals, retransmission intervals, inactivity timeouts and exponential backoffs. An entity may specify these constants for a specific Sequence or for a set of Sequences. WS-Policy can also be used to convey security related information.

WSRM is intended to enable incremental additional of capabilities to a Web Service endpoint. All that a web service endpoint needs to ensure is that ALL inbound and outbound SOAP messages are funneled through the WSRM implementation. Once this is done, the web service endpoint should be able to avail of WSRM's reliable messaging capabilities automatically.

3. Implementation of the WSRM

To facilitate incremental addition of capabilities at a web service endpoint, we had two objectives. First, the endpoint should have access to all WSRM related capabilities. Second, the interface to do so must very simple. In order to achieve this incremental addition all inbound and outbound SOAP messages at the web service endpoint should be funneled through the WSRM software.

In our implementation (Java-based) functionality related to the sink and sink roles in WSRM are encapsulated within the **SourceProcessor** and **SinkProcessor** respectively. Both these processors extend the **WsProcessor** base class which contains several of the capabilities that are need in both the **SourceProcessor** and **SinkProcessor**. First, 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. This method provides the entry point to capabilities encapsulated within approximately 300 Java classes related to WSRM processing. This scheme also satisfies our objective of enabling simplified addition of WSRM capabilities. 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 **SourceProcessor** and the **SinkProcessor** will be cascaded at the endpoint.

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**.

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 WSRM endpoints. For example, the 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 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 exception related to processing 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 SourceProcessor receiving a control message corresponding to a different WS specification such as a WS-Eventing subscribe request. An **IncorrectExchangeException** is thrown if the WsProcessor instance should not have received a specific exchange. For example if a WSRM SinkProcessor 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.

The capabilities within the **WsProcessor** and the **WsMessageFlow** enable the **SourceProcessor** and **SinkProcessor** to focus only on the logic related to the respective roles within the WSRM protocol. Upon receipt of an outgoing SOAP message the **SourceProcessor** checks to see if an active Sequence currently exists between the hosting endpoint and the remote endpoint. If one does not exist, the **SourceProcessor** automatically initiates a create sequence exchange to establish an active Sequence. For each active Sequence, the **SourceProcessor** also keeps track of the Message Number last assigned to ensure that they monotonically increase, starting from 1. The **SourceProcessor** 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 **SinkProcessor** responds to the requests to create a sequence, and also acknowledges any messages that are received from the source. The **SinkProcessor** issues acknowledgements (both positive and negative) at predefined intervals and also manages inactivity timeouts on Sequences. Finally, both the **SourceProcessor** and **SinkProcessor** 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 wsa:ReferenceProperties and wsa:ReferenceParameters element contained in a service endpoint's EPR. 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 EPR).

Upon receipt of a SOAP message, at either the **SourceProcessor** or the **SinkProcessor**, 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.

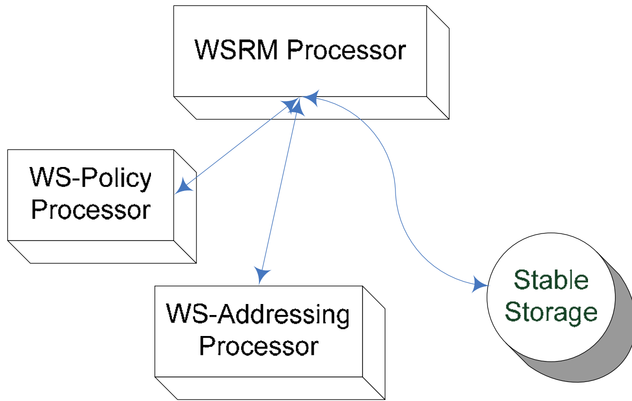

Figure 1: Overview of WSRM implementation

3.1 Processing Schemas

Figure 1 provides a high-level view of the architecture of our implementation

While implementing the WSRM specifications we were faced with an important decision regarding the choice of tool to use in processing the XML schema pertaining to WSRM, WS-Addressing, WS-Policy and SOAP. We were looking for a solution that allowed us to process XML from within the Java domain. There were three main choices. First, 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 [7]. Specifically, the problems related to insufficient (and in some cases incorrect) support for complex schema types, XML validation and serialization issues.

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

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

4. Performance Measurements

We now include performance measurements from our experiments. These experiments were performed on a 3.5 GHz Pentium IV machine with Sun's 1.4.2 Java Virtual Machine. For each measurement we performed the experiment 100 times. An outlier removal program was used to remove outliers, if any, in the result set. To detect outliers we first calculate the mean and standard deviation of the entire data set. This is then used to obtain a z-score for each data point, according to following formula: $z_i = \frac{x_i - \overline{x}}{s}$ where $\overline{x}$ is the mean and s is the standard deviation of the original sample. If the z-value is greater than 3, the corresponding data point is deemed an outlier.

For each run we also tracked the memory utilization. This was done by simply recording the memory utilization prior to the invocation of a specific operation and after the invocation. In some cases this calculation resulted in a negative utilization because of garbage collection (via the Java garbage collector thread) in the intervening period. We have measured several relevant performance aspects of our implementation. We now proceed to discuss each of this in detail. A synopsis of our results is also available in a separate table (Table 1) for the reader's perusal. This table lists the operation, the mean, the standard deviation, standard error, minimum and maximum values for the CPU bound latencies (in microseconds) and finally the memory utilization associated with the operation.

In our performance measurements we started off by measuring the time to create a SOAP messages within the Axis Web Services container (SOAPMessage) and using our XMLBeans representation of the SOAP schema (EnvelopeDocument). We found that the costs in terms of {latencies, memory utilization} for these operations were similar. For EnvelopeDocument the cost was {126.86 µSecs, 2192 B} while for SOAPMessage this cost was around {117.34 µSecs, 2192 B}. The standard deviation (and the corresponding standard error) was higher for SOAPMessage creation at 187.30 µSecs. Since every interaction between web service endpoints are encapsulated within SOAP messages, these costs represent the minimum costs that such interactions may incur.

To facilitate deployments within Apache's Axis and Sun's JWSDP container, we have developed utilities which facilitate conversions between the SOAP representations — SOAPMessage and EnvelopeDocument. The cost to convert an EnvelopeDocument into a SOAPMessage was around {2627.54 µSecs and 60816B} while the cost for converting a SOAPMessage into a EnvelopeDocument was around {827.58 µSecs, 34424B}.

One of the reasons for this disparity is that the Axis implementation renames the schema namespace qualifiers contained within the EnvelopeDocument.

Since WSRM heavily leverages the WS-Addressing specification we benchmarked some overheads related to WS-Addressing processing. Here, we first measured the costs associated with the creation of simple EPRs based on a simple URL String and the more elaborate EPR that contains the wsa:ReferenceProperties element. As might be expected the costs for the simple EPR (150.51 µSecs, 2648B) were better than those for the more elaborate EPR (397.34 µSecs, 7184B).

Next, we measured the costs involved in the creation of a SOAP message, targeted to a specific EPR, with the most basic WSA fields — wsa:To and wsa:MessageID within the SOAP message. In the second case, we included additional elements such as wsa:From, wsa:RelatesTo and the wsa:Action field. In both these cases the created SOAP message conformed to the rules outlined in the WS-Addressing specification. Here we found that the cost for creating the SOAP message with basic WS-Addressing elements were (397.34, 7184B) while the cost for additional elements was (537.81 µSecs, 13880B).

Upon receipt of a SOAP message, the first task that needs to be performed is the parsing of the SOAP message for the WS-Addressing elements. This is typically the precursor to further more specific parsing later on since the WSA elements indicate not only the semantic intent (wsa:Action) but also the context (wsa:Relates, wsa:MessageID) and also where errors need to be issued to in case there are problems. For example, once we have determined the semantic intent of a message from the wsa:Action to be a Create Sequence request, we may initiate operations to parse the wsrm:CreateSequence element within the Body of the SOAP message. In our benchmarks the cost for parsing the SOAP message for WS-Addressing elements was found to be {1224.752 µSecs, 61024B}. Since this operation is performed for every SOAP message this is a cost that will be incurred during each interaction between the service endpoints.

Next, we measured the costs involved in the creation of a WSRM create sequence request (352.16 µSecs, 16392B) and the response (335.21 µSecs, 18160B) generated upon the receipt of this request. These costs are in addition to any costs involved due to communication overheads between the service endpoints.

For every message received from the hosting service endpoint at the SourceProcessor, the appropriate wsrm:Sequence is added. This contains the identifier associated with the previously created Sequence and the Message Number assigned to this message. We measured the costs involved in the creation of this wsrm:Sequence element (44.72 µSecs, 2424B) and the costs involved in the addition (12.67 µSecs, 464B) of this element to the SOAP message received at the SourceProcessor.

A WSRM sink is expected to acknowledge messages at regular intervals (based on the acknowledgement interval). We have measured the costs involved in the creation of wsrm:SequenceAcknowledgement document based on a set of Message Numbers. We found this cost to be (516.58 µSecs, 20624B). This cost includes the costs involved in the creation of the one or more wsrm:AcknowledgementRange elements which cover acknowledgements for a group of messages. Thus if one is acknowledging Message Numbers 1,2,3,4,5,7,8,9,11,12,13 there would be 3 acknowledgement ranges corresponding to 1—5, 7— 9 and 11—13.

We also measured the costs involved in the creation of wsrm:TerminateSequence (24.66 µSecs, 2072B) and the time to create a WSRM Fault(519 µSecs, 18096) based on the rules outlined in the WSRM and WS-Addressing specifications.

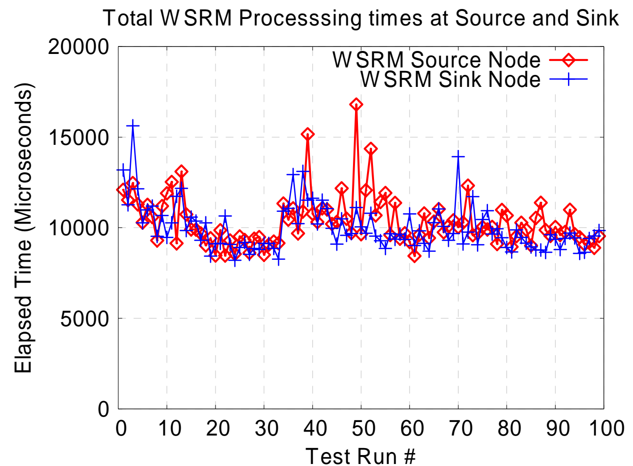

Figure 2: Total Processing times

Figure 2 depicts the total processing times at a WSRM source and sink. This includes the times for storage of message to stable storage at both source and sink. In our experiments the stable storage was based on MySQL. For MySQL we found the storage cost to be typically between 4-6millseconds for message sizes 100B-10KB. Only after the SOAP message with the added wsrm:Sequence element has been stored onto stable storage will the message be routed to the remote sink endpoint. The graph does not include communication overheads involved in communication between the service endpoints. So these costs are in addition to the networking costs involved. In our experience we have found this cost to vary from a few milliseconds in LAN settings to a few hundred milliseconds in WAN settings.

Table 1: Summary of results (All results in Microseconds)

Operation	Mean	Standard Deviation	Standard Error	Num of Outliers	Min Value	Max Value	Memory Utilization (Bytes)
Create an XMLBeans based Envelope Document	126.864	49.395	5.041	4	108	424	2192
Create an Axis based SOAPMessage	117.340	187.302	19.017	3	34	1183	1824
Convert an EnvelopeDocument to a SOAPMessage	2627.548	905.483	93.894	7	1722	5350	60816
Convert SOAPMessage to EnvelopeDocument	827.589	586.872	60.211	5	325	2802	34424
Create a WS-Addressing EPR (Contains just a URL address)	87.562	58.590	5.979	4	71	465	2072
Create a WS-Addressing EPR (Contains WSA ReferenceProperties)	150.515	96.764	9.927	5	112	705	2648
Create an Envelope targeted to a specific WSA EPR	397.340	200.396	20.669	6	267	1276	7184
Create an Envelope targeted to a specific WSA EPR with most WSA message information headers	537.814	347.497	35.283	3	344	2123	13880
Parse an EnvelopeDocument to retrieve Wsa Message Info Headers	1224.752	727.870	73.904	3	645	4573	61024
CreateWsrmSequenceRequest	352.163	260.997	26.364	2	229	1568	16392
CreateWsrmSequenceResponse	335.210	226.060	23.193	5	224	1174	18160
CreateWsrmSequenceDocument	44.724	4.733	0.478	2	42	75	2424
Add a WsrmSequenceDocument to an existing envelope. (Contains sequence identifier and message number)	12.670	0.494	0.050	3	12	14	464
Create a WSRM SequenceAcknowledgement based on a set of message numbers	516.583	248.274	25.339	4	335	1514	20624
CreateTerminateSequence	24.666	36.203	3.638	1	19	380	2072
CreateWsrmFault	519.802	294.699	30.077	4	347	1619	18096

5. Related Work

Traditional group based systems have a large body of work [8, 9] addressing the problems of reliable delivery and ordering. An exemplar of a group based system which addressed these issues is the Isis system, which pioneered the *virtual synchrony* model. Here a distributed system is allowed to partition under the assumption that there would be a unique partition which could make decisions on behalf of the system as a whole, without risk of contradictions arising in the other partitions and also during partition mergers. The Isis [10] model works extremely well works well for problems such as propagating updates to replicated sites. By incorporating variants of the virtual synchrony model systems such as Horus [11] and Transis [12] can handle concurrent views in different partitions.

In the area of publish/subscribe systems such as DACE [13], Gryphon [14] and NaradaBrokering [15] address the problem of reliable delivery from multiple producers to multiple consumers. These systems incorporate schemes which are based on asynchronous control-message exchanges to ensure reliable delivery. These systems also leverage stable storages to facilitate reliable delivery and also have extensions to enable exactly once delivery of messages.

Message queuing products (MQSeries) [16] leverage the store-and-forward approach where the queues are statically pre-configured to forward messages from one queue to another. This forwarding takes place only after the queue has first stored a message to stable storage. Powerful distributed object systems such as CORBA also have schemes for increased reliability for CORBA applications. More specifically, the FT-CORBA) [17] specification leverages entity redundancy (through replication) and defines interfaces, policies and services to achieve additional resilience.

In the area of Web Services, the WS-Reliability specification from Sun and Oracle includes support for

more or less the same set of capabilities as in WS-Reliable Messaging.

6. Conclusions and Future Work

In this paper we presented details about our implementation of the WS-ReliableMessaging specification. We also included results from our implementation. We hope that this work can be used by researchers and system designers to make informed decisions about their Web Services based reliable messaging strategy. Specifically, researchers can use this work to decide the tradeoffs that might need to be made within their application domain.

We have recently finished implementation of the WS-Reliability specification and will be releasing this software to the open source community soon. One area of future research that we feel holds promise if the the use of pull parsers to speed up some of the parsing operations. We are hopeful that this strategy would improve the overall performance costs involved and intend to explore this further in future works.

References

[1] M. Gudgin, et al, "SOAP Version 1.2 Part 1: Messaging Framework," June 2003. http://www.w3.org/TR/2003/REC-soap12-part1-20030624/

[2] Web Services Description Language (WSDL) 1.1 http://www.w3.org/TR/wsdl

[3] Web Services Reliable Messaging Protocol (WS-ReliableMessaging) ftp://www6.software.ibm.com/software/developer/library/ws-reliablemessaging200403.pdf

[4] Web Services Reliable Messaging TC WS-Reliability. http://www.oasis-open.org/committees/download.php/5155/WS-Reliability-2004-01-26.pdf

[5] Web Services Addressing (WSAddressing) ftp://www6.software.ibm.com/software/developer/library/wsadd200403.pdf

[6] Web Services Policy Framework (WS-Policy). IBM, BEA, Microsoft and SAP. http://www-128.ibm.com/developerworks/library/specification/ws-polfram/

[7] 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/.

[8] Vassos Hadzilacos and Sam Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical Report TR94-1425, Cornell University, Ithaca, NY-14853, May 1994.

[9] Kenneth Birman and Keith Marzullo. The role of order in distributed programs. Technical Report TR 89-1001, Cornell University, Ithaca, NY 14853, 1989.

[10] Kenneth Birman. Replication and Fault tolerance in the ISIS system. In Proceedings of the10th ACM Symposium on Operating Systems Principles, pages 79–86, Orcas Island, WA USA, 1985.

[11] R Renesse, K Birman, and S Maffeis. Horus: A flexible group communication system. In Communications of the ACM, volume 39(4). April 1996.

[12] D Dolev and D Malki. The Transis approach to high-availability cluster communication. In Communications of the ACM, vol 39(4). April 1996.

[13] Romain Boichat Effective Multicast programming in Large Scale Distributed Systems. Concurrency: Practice and Experience, 2000.

[14] Sumeer Bhola, Robert E. Strom, Saurabh Bagchi, Yuanyuan Zhao, Joshua S. Auerbach: Exactly-once Delivery in a Content-based Publish-Subscribe System. DSN 2002: 7-16

[15] Shrideep Pallickara and Geoffrey Fox. NaradaBrokering: A Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids. Proceedings of ACM/IFIP/USENIX International Middleware Conference. 2003.

[16] The IBM WebSphere MQ Family. http://www-3.ibm.com/software/integration/mqfamily/

[17] Object Management Group, Fault Tolerant CORBA Specification. OMG Document orbos/99-12-08 edition, December 1999.