

Automatic code generation for LYE, a high-performance caching SOAP implementation

Venkatesh Prasad Ranganath Andrew King Daniel Andresen
Department of Computing and Information Sciences
Kansas State University
234 Nichols Hall, Manhattan KS, 66506, USA

Abstract

In this paper, we present our experience in automating the XML schema driven serialization approach within the Apache Axis 1 and Axis 2 SOAP frameworks. We have generalized our previous template based approach to serialization [7] as a generic XML schema driven serialization approach and realized the generic approach via two stylistically different and non-intrusive implementation strategies. We illustrate the benefits of our approach – performance improvements of up to 89% with low memory overhead – by empirically comparing it with the Java Bean based approaches employed in Apache Axis 1 and Axis 2.

KEY WORDS: SOAP, distributed computing, eCommerce, network protocols

1. Introduction

Service-oriented architectures (SOAs) have become a critical new technology for eCommerce, high-performance computing, and the Computational Grid. In response to the need for a standard to support web services, SOAP has become the standard binding for the emerging Web Services Description Language (WSDL) [8, 9]. SOAP is based on XML [2] and thus achieves high interoperability when it comes to exchange of information in a distributed computing environment. While carrying the advantages that accrue with XML, it has several disadvantages that restrict its usage. SOAP calls have a large overhead due to the considerable execution time required to process XML messages. In this paper, we partially mitigate a primary shortcoming of SOAP: *its speed of execution*. We do this by automating the XML schema driven serialization approach within the Apache Axis 1 and Axis 2 SOAP frameworks. Specifically, we generalize our previous template based approach to serialization [7] as a generic XML schema driven serial-

ization approach and realize the approach via two stylistically different and non-intrusive implementation strategies. Both approaches yield substantial improvements over the existing implementation.

An overview of SOAP along with the associated practical concerns and efforts are presented in Section 2. In the following section, we provide an details about Apache Axis 1 and Axis 2. Details about the XML schema driven approach and its implementation in Axis frameworks is discussed in Section 4. A comparative and empirical evaluation of our approach is provided in Section 5. We conclude the paper in Section 7 following an exposition about possible extensions of our current effort in Section 6.

2. Background

2.1. Overview of SOAP

Simple Object Access Protocol (SOAP) defines a standard packaging format for transmitting XML data between applications on a network. The protocol is designed to carry the data (*payload*) to be transmitted wrapped in metadata (*envelope*) relevant to data transmission (routing), access (security), etc. Further, SOAP is leveraged by *Java API for XML-based RPC (JAX-RPC)* to enable XML based remote procedure call (RPC) in Java.

SOAP-based RPC can be viewed to be composed of messages and services. A *message* is composed of an envelope and a payload. A *service* is an endpoint that accepts a request, processes the request, and returns a response. In terms of RPC, the services can be perceived as the RPC modules that expose various operations/procedures, the incoming messages can be perceived as requests containing the procedure name along with arguments, and the outgoing messages can be perceived as responses containing the return values from the remote procedure call. Typically, a SOAP engine (such as Apache Axis) waits for SOAP request messages, deserializes and decodes the SOAP envelope in the request message, routes the payload to the ap-

propriate service, *deserializes the payload*, executes the service with the payload data, *serializes the result data from the service into the response payload*, embeds the response payload in a SOAP response message, and uses the network transport to transmit the SOAP response message back to the client.

The information that identifies the services and the offered operations and describes the structure of parameters and return value of the offered operations is described via the XML-based *Web Services Description Language (WSDL)* in a web service descriptor.

2.2. Concerns

The reliance on XML to encode the objects and operations makes SOAP an ideal protocol to connect heterogeneous systems implemented in different programming languages, built using different software frameworks, and/or based on different application level network protocols. However, as indicated by various studies [3, 5, 4], the reliance on XML makes SOAP-based RPC inefficient in comparison with other distributed computing solutions such as Java RMI and CORBA.

As SOAP messages are XML based, they need to be encoded as text. Although this is beneficial, Bustamante et.al [3] observed that there is a dramatic difference in the amount of encoding necessary for transmitting data in XML form as opposed to being binary encoding like in CORBA.

Kohlhoff et.al [6] state that XML related performance cost is insufficient to explain SOAPs poor performance. SOAP message compression was an attempt to optimize SOAP that was later discarded as the cost of compression and decompression annulled the benefits. In the same effort, compact XML tags were used to reduce the length of the XML tag names. This had negligible improvement on encoding, which suggests that the major cost of the XML encoding and decoding is in the structural complexity and syntactic elements, rather than message data.

Orthogonally, in our previous efforts [1, 7], we observed that caching could improve the encoding phase of message serialization in the context of SOAP-based RPC and provide upto 250-600% improvement in sample applications. Further, we observed that serialization could be optimized by leveraging the message type structure information embedded in the web service descriptor to precompute the skeleton of the message [7]. We shall discuss this approach in detail in the rest of this paper.

3. Apache Axis

The *Axis* project from *Apache* foundation provides *Axis 1* and *Axis 2*, two versions of Java-based SOAP implementations. As we have used these implementations in experi-

ments and to realize our techniques, we shall provide a brief overview of these two versions of Axis followed by a detailed exposition about our techniques and how they are realized in Axis.

3.1. Axis 1

The Axis 1 distribution contains three main parts. The first part is the server side framework for SOAP based web services (region (a) in Figure 1). The second part is the SOAP engine (region (b) in Figure 1), which interacts with the server side code (Target in Figure 1) and web clients that use SOAP based RPC. The third part is a collection of utility programs that aid web service authors to automatically generate code skeletons for the SOAP methods described in WSDL and to automate the publishing of the web service onto an Axis 1 server. Our techniques are applicable to the Axis 1 SOAP engine.

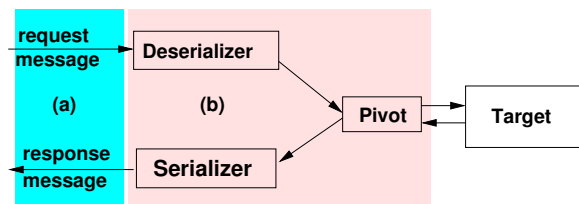


Figure 1. Server-side control flow in Axis.

Axis 1 provides a serialization pipeline along with a collection of serializers that are capable of serializing instances of Java classes that implement the `java.lang.Serializable` interface into a XML fragment. This pipeline serializes the object resulting from the target. The distribution contains serializers for values of primitive data types available in the Java programming language. The serialization of any plain old Java objects occurring in the object resulting from the target is handled by the generic `BeanSerializer` by leveraging Java reflection mechanism.

More specifically, the `BeanSerializer` recursively visits each object reachable from the result object (data) and, at each object, the serializer uses Java Reflection API to retrieve the type (class) of the object to learn the type of member objects and decide the serialization strategy for these member objects. During this process, the `BeanSerializer` also attempts to collect any schema information, such as namespaces, occurrence constraints, etc, corresponding to the object types and use it in serialization. After every object reachable from the result object has been visited and serialized, the message is handed over to the server side framework (region (a) in Figure 1), which then transmits the message to the client.

In Axis 1, developers can control (de)serializers used to (de)serialize various types used in web services, hence, customize the (de)serialization process. This is enabled via the Axis 1 specific *Web Service Deployment Descriptor (WSDDD)*¹ used to deploy web services into an Axis 1 server.

3.2. Axis 2

The overall architecture of Axis 2 is similar to that of Axis 1 except for Axis 2's reliance on AxiOM to serialize and deserialize Java objects. *AxiOM* (Axis Object Model) is a set of Java classes that can be used to represent the structure and contents of an XML document in the form of an object tree. These classes also provide features to manipulate the XML document represented by the object tree.

Upon processing a web service descriptor, the *WSDL2Java* provided by Axis 2 generates Java source code for the serializers and deserializers of the complex types defined in the descriptor. These (de)serializers can then be compiled and used at runtime by seamlessly deploying them into Axis 2 server.

In contrast to Axis 1 serialization pipeline, the Axis 2 serialization pipeline does not serialize the result object from the service. Instead, the serialization pipeline recursively visits the reachable objects and constructs an hierarchical *Object Model (OM) Tree* (similar to the Document Object Model (DOM)) that corresponds to the structure of the XML representation (inclusive of the XML tags) of the result object. This object tree is subsequently serialized by the server side framework (region (a) in Figure 1) before transmitting the message to the client. Succinctly, the serialization pipeline in Axis 1 provides the XML representation of the result object in string form while the pipeline in Axis 2 provides the XML representation in OM tree form.

The OM tree has a `serialize` method that is invoked to serialize the represented XML document into a string. Similar to the serialization process in Axis 1, the `serialize` method traverses the OM tree to generate the string representation of the XML fragment represented by each node of the tree. Like Axis 1, Axis 2 also provides the facility to plug in custom (de)serializers that can build an OM tree.

3.3. Overhead

Despite the fixed message structure for a method of a web service, the Axis 1 serialization pipeline *rediscovers* the structure of the result object for each response message. This is compounded by the use of Java Reflection facility to determine the members of the object type for

purposes of visiting the reachable objects. Such repeated reflection-based introspection unnecessarily contributes to the response time of applications, such as airline reservation systems, where the shape of the response remains constant for a given method. In addition, the repeated concatenation of XML tags contributes to the cost of serialization.

Both of the above mentioned overheads apply to Axis 2 as well. Furthermore, the Axis 2 serialization pipeline visits the objects reachable from the result object while constructing the OM tree and performs a similar traversal while serializing the OM tree. In applications involving high frequency requests and/or large results, the extra traversal can degrade the performance of the SOAP server. Also, given the high cost of object allocation, the creation of an object during the construction of the OM tree will further contribute to the degradation of performance.

4. XML Schema Driven Message Serialization

The type information embedded in XML schemas is used by XML (de)serialization frameworks (Castor, JiBX, XStream, etc) to automatically generate (de)serializers that can convert XML fragments into Java objects and vice versa. As seen in Figure 4, XML schema fragments that describe various XML types used in messages by a SOAP service are embedded in the corresponding web service descriptor. Hence, in our earlier effort [7], we proposed an approach to generate serializers from these XML schema fragments in the context of SOAP services. Recently, we automated our approach by implementing generators of serializers that are readily pluggable into Axis 1. In this section, we shall describe our implementation along with its variations and how our approach differs from the default approach available in Axis 1 and Axis 2.

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="SampleService">
  <types>
    <schema>
      <complexType name="ComplexObject">
        <sequence>
          <element name="varFloat" type="xsd:float"/>
          <element name="varInt" type="xsd:int"/>
          <element name="varString" type="xsd:string"/>
        </sequence>
      </complexType>
    </schema>
  </types>
</definitions>
```

Figure 2. Sample descriptor containing the XML schema corresponding to a type composed of float, integer, and string

¹ We use the term “descriptor” to imply *web service descriptor* and **not** web service deployment descriptor.

4.1. Implementation

Typically, an Axis 1 based web service developer obtains or writes a web service descriptor in WSDL and feeds it to WSDL2Java utility program to generate code skeleton for the service. The developer then adds appropriate business logic to the code skeleton, compiles the code, and publishes the new web service by using AdminClient utility.

In an attempt to keep the application of our approach simple, we implemented a new tool named WSDL2Ser that processes web service descriptors and generates custom serializers in Java. Given the similarity of the tools, we imagine that our implementation can be integrated into WSDL2Java.

In our implementation, the serialization logic accepts the result object as input and generates the XML representation of the result object in string form. Hence, the generated serializers can replace the BeanSerializers in Axis 1 framework. As the Axis 2 framework provides a result object to the serializer and expects an OM tree as the result of serialization, the Axis 1 compatible serializers cannot be used as-is with Axis 2. Instead, we need to introduce a thin Axis 2 compliant serializer *S2* that accepts the result object and uses it to construct and return an Axis 1 compatible serializer *S1* conforming to the OM tree interface. Hence, with minor adaptations, our approach is applicable to both Axis 1 and Axis 2. At the time of writing this paper, we are still realizing generators for Axis 2. However, we expect the results in the context of Axis 2 will be similar to those in the context of Axis 1.

4.2. Variations

The WSDL2Ser program reads the web service descriptor that generates a class for the each message type in the descriptor. The generated class conforms to the API as required by Axis 1 (Axis 2) framework. The serialization method in the class contains a template of the message in its string form. The method also contains the logic to traverse the object to be serialized, collect values, and instantiate the template based on the collected values.

We have implemented two variations of serializers: *template engine based serializers* and *pure Java based serializers*.

4.2.1. Template Engine based Serialization A template engine (such as FreeMarker² or Velocity³) provides the facility to define and instantiate a template. Template engines enable developers to focus on realizing the structure of the content instead of focusing on the logic required to realize

the structure. Further, they add a layer of abstraction that enables optimizations to the engine to be easily absorbed by client applications.

Driven by the above advantages, we employed the Velocity template engine to realize the template engine based variation of our approach. Given a web service descriptor, WSDL2Ser outputs two artifacts: 1) a velocity template that captures the structure of message type in its string form and 2) a Java serializer class that loads the generated template and exercises the template engine at runtime to serialize a result object.

The generated velocity template and the corresponding Java serializer class for the descriptor in Figure 4 are given in Figures 4.2.1 and 4.2.1, respectively. The compile time concretization of XML tags as string constants in the template enables the template engine to splice consecutive tags to reduce the number of string concatenation operations required during serialization. More importantly, direct access to components of the result object via getter methods (Figure 4.2.1) (instead of via reflection) contributes to a large reduction of the serialization time.

```
<multiRef id="{multiRefid}" ${soapencprefix}:root="0"
  soapenv:encodingStyle=
    "http://schemas.xmlsoap.org/soap/encoding/"
    ${xsiprefix}:type="{complexprefix}:ComplexObject"
    xmlns:${soapencprefix}="{soapencuri}"
    xmlns:${complexprefix}="{complexuri}">
  <varFloat ${xsiprefix}:type="{xsdprefix}:float">
    ${varFloat_Value}
  </varFloat>
  <varInt ${xsiprefix}:type="{xsdprefix}:int">
    ${varInt_Value}
  </varInt>
  <varString ${xsiprefix}:type="{xsdprefix}:string">
    ${varString_Value}
  </varString>
</multiRef>
```

Figure 3. Auto-generated Velocity Template corresponding to the XML Schema in Figure 4

4.2.2. Pure Java based Serialization Despite the benefits, template engine based serialization entails the cost of exercising the template engine. Hence, if the structure of the message type is simple then it is faster to use features and libraries native to the Java programming language to realize serialization.

In the second variation of our approach, we partially evaluate the process of instantiating the template by considering the template to be instantiated as a constant. Hence, the resulting serializer (Figure 4.2.2) is merely a sequence of append calls on an instance of java.lang.StringBuffer class interspersed with

² <http://www.freemarker.org>

³ <http://jakarta.apache.org/velocity>

```

private String serializeObj(ComplexObject temp) {
    StringWriter writer = new StringWriter();
    vtlContext.put("varFloat_Value",
        Boolean.toString(temp.getvarFloat()));
    vtlContext.put("varInt_Value",
        Double.toString(temp.getvarInt()));
    vtlContext.put("varString_Value",
        Integer.toString(temp.getvarString()));
    vtlTemplate.merge(vtlContext, writer);
    return writer.toString();
}

```

Figure 4. Auto-generated Template-based serializer corresponding to the template in Figure 4

calls to the getter methods of the result object. The benefits of the template based variation also apply to this variation along with the additional benefits of avoiding the cost associated with exercising the template engine.

4.3. Comparison

During serialization in Axis 1 and Axis 2, a significant amount of time is spent to determine the fields of the objects constituting the result object and how to build the message based on the type of the objects. Further, the creation of helper objects (such as `Attribute` lists (in Axis 1) and the `OM TreeNodes` (in Axis 2) contribute to the serialization costs in Axis 1 and Axis 2, respectively.

These extra costs are avoided in the XML schema driven serialization approach. As the structure of the message is encoded into the serializers at compile time (compiling the serializers from the XML schema), the generated serializer does not discover the message structure or create helper objects during serialization. Hence, the cost of serialization is reduced in terms of time and memory.

In addition, the number of runtime string concatenations is also minimized in the proposed approach as consecutive static parts (tags) of the XML fragment are concatenated at compile time.

5. Experiments

We have validated the benefits of our approaches by measuring and comparing the time taken to convert a Java object (the response from the target) into a valid XML fragment that can be embedded into a XML SOAP response message.

5.1. Setup

In our experiments, we considered the default serialization approach supported by Axis 1 (version 1.3) and Axis

2 (version 0.94) along with the template based and Java `StringBuffer` based variants of the XML schema driven approach. The two versions of Axis used were the latest releases available at the time of this writing. Each of the proposed approaches were exercised with four SOAP services that differed in the complexity of the structure of the response: 1) *Simple* service returned an integer value, 2) *Tuple* service returned a tuple object containing a string value, an integer value, a boolean value, and a long value, 3) *Array* service returned an array of 100 string values, and 4) *Complex* service returned a complex object composed of an array of 100 string values, an array of 100 integer values, and four tuple values (described previously). The first two services were designed to represent applications such as stock tickers and sportscasts while the others were designed to represent weather applets and involved business applications that involve large data transfer.

In each experiment, the SOAP server was set up to exercise a particular serialization approach. The client sequentially issued identical requests for the same SOAP service to the server. Both the client and the server were co-located on a 3.06 GHz Windows XP computer equipped with 1GB of RAM, and were executed on top of Sun's JVM version 1.5.0_03-b07. Sun's Java performance measurement package `sun.misc.perf` was used to collect timing data.

5.2. Results

The data from the experiments are provided in Table 5.2 and Figure 6. For each SOAP service type, we used Axis 1 as the baseline to measure the relative improvement of performance.

The Schema driven approaches resulted in better overall performance and relative improvement for all four SOAP service types. *Java String based Schema driven approach* provided the best performance (55-89% improvement in comparison with Axis 1) as the conversion of the Java result object to its SOAP conformant string representation was *partially evaluated*. The realization of the residual conversion operations via simple features of the Java language and runtime library also contributed to the improved performance. Although the *template based Schema driven approach* performed better than the default approaches available in Axis 1 and Axis 2, the overhead induced by the genericity of a template library resulted in slightly larger serialization times compared with the *Java String based Schema driven approach*. Despite this fact, the template based Schema driven approach performed 15-87% better than the default message serialization approaches available in Axis 1.

Independent of the type of SOAP services, the bean-based serialization technique employed in Axis 1 performed better than the XMLBeans-based serialization technique

```

private String serializeObj(ComplexObject temp){
    StringBuffer result_buffer = new StringBuffer();
    //now misc XML info
    String multiRefid = "id0";
    String xsiprefix = context.getPrefixForURI("http://www.w3.org/2001/XMLSchema-instance");
    String xsdprefix = context.getPrefixForURI("http://www.w3.org/2001/XMLSchema");
    String soapncprefix = context.getPrefixForURI("http://schemas.xmlsoap.org/soap/encoding/");
    String complexprefix = context.getPrefixForURI("http://service.timeTest");
    String soapencuri = "http://schemas.xmlsoap.org/soap/encoding/";
    String complexuri = "http://service.timeTest";
    //now the tags, "the bookends"
    String item_tag_start = "<item>";
    String item_tag_end = "</item>\n";
    StringBuffer varFloat_tag_start = new StringBuffer()
        .append("<varFloat ") .append(xsiprefix) .append(":type=\"") .append(xsdprefix) .append(":float\">");
    StringBuffer varFloat_tag_end = new StringBuffer("</varFloat>\n");
    StringBuffer varInt_tag_start = new StringBuffer()
        .append("<varInt ") .append(xsiprefix) .append(":type=\"") .append(xsdprefix) .append(":int\">");
    StringBuffer varInt_tag_end = new StringBuffer("</varInt>\n");
    StringBuffer varString_tag_start = new StringBuffer()
        .append("<varString ") .append(xsiprefix) .append(":type=\"") .append(xsdprefix) .append(":string\">");
    StringBuffer varString_tag_end = new StringBuffer("</varString>\n");
    StringBuffer type_header = new StringBuffer().append("<multiRef id=\"") .append(soapncprefix)
        .append(":root=\"0\" ") .append("soapenv:encodingStyle=\"http://schemas.xmlsoap.org/soap/encoding/\" ")
        .append(xsiprefix) .append(":type=\"") .append(complexprefix) .append("ComplexObject\"") .append("xmlns:")
        .append(complexprefix) .append("=\"") .append(complexuri) .append("\">\n");
    StringBuffer msg = type_header.append(varFloat_tag_start).append(temp.getvarFloat()).append(varFloat_tag_end)
        .append(varInt_tag_start).append(temp.getvarInt()).append(varInt_tag_end)
        .append(varString_tag_start).append(temp.getvarString()).append(varString_tag_end)
        .append("</multiRef>\n");
    return msg.toString();
}

```

Figure 5. Auto-generated pure Java based Serializer corresponding to the XML Schema in Figure 4

	Simple	Tuple	Array	Complex
Axis 1 (Bean)	72.17 (0)	100.02 (0)	2272.29 (0)	4030.7 (0)
Template	57.01 (21)	85.33 (14.69)	292.99 (87.11)	1188.16 (70.52)
Java string	26.1 (63.84)	45.11 (54.9)	245.83 (89.18)	754.81 (81.27)
Axis 2 (XMLBeans)	946.85 (-1212)	1015.01 (-914.77)	3276.99 (-44.22)	4428.65 (-9.87)

Table 1. Time spent in serialization in milliseconds for 1000 requests. The relative performance improvement over Axis 1 is provided within parenthesis.

used in Axis 2. Interestingly, the performance of the serialization technique used in Axis 2 was comparable to that used in Axis 1 when the complexity of the structure of the SOAP message increased in terms of the cardinality of composition and frequency of the elements of the same type.

6. Future Work

We plan to integrate and contribute the proposed approaches to the main code base of Apache Axis 1 and Axis 2. `WSDL2Ser` generates serializers that can plug directly into Axis 1. The same approach can be applied to generating Axis 2 serializers. However, Axis 2 does not support message serialization at the same logical point as in Axis 1. More specifically, in Axis 2, the serializer generates an OM tree and passes it to the Axis 2 engine to be rendered as a

string.

We also plan to extend `WSDL2Ser` to generate serializers that can handle messages containing `multirefs` and recursive data types.

Finally, we plan to conduct experiments to evaluate the combined benefits of the proposed schema driven approaches and caching techniques proposed in our earlier work [1, 7]. Further, it would be an interesting task to adapt the proposed approaches to be applicable to the XSD-conformant XML serialization problem.

7. Conclusion

In this paper, we validate our previous claim [7] that SOAP message serializers can be automatically synthesized

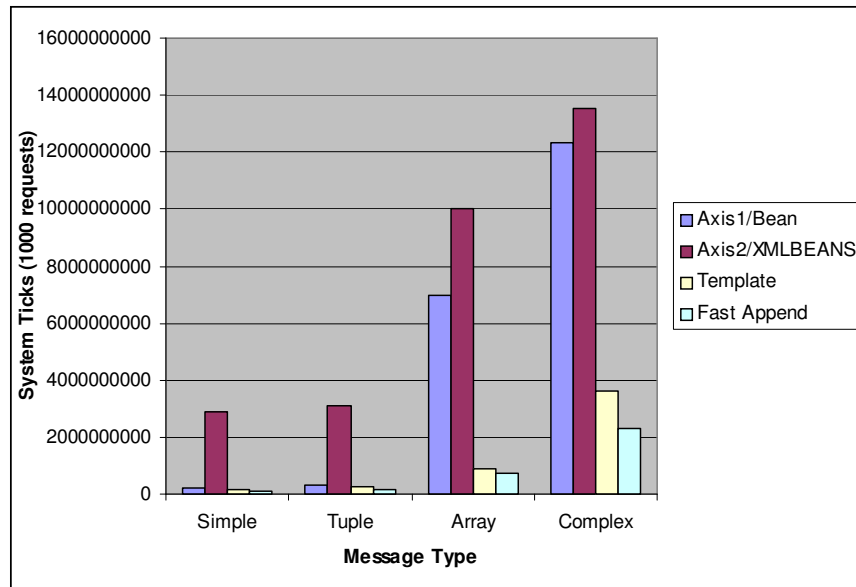


Figure 6. Serialization techniques vs. Message type

by leveraging the structure of various data types defined (via XML Schema) in SOAP service descriptors. In addition to the template based serialization technique, we introduce a trivial yet faster serialization technique based merely on the features of `java.lang.StringBuffer` class available in the Java runtime library. We empirically illustrate the proposed techniques perform up to 89% better than the default techniques employed in Apache Axis 1 and Axis 2.

Acknowledgments

This material is based in part upon work supported by the National Science Foundation under the award numbers CCR-0082667 and ACS-0092839. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

References

- [1] D. Andresen, D. Sexton, K. Devaram, and V. P. Ranganath. LYE: A High Performance Caching SOAP Implementation. In *Proceedings of the 2004 International Conference on Parallel Processing (ICPP-2004)*, pages 143–150, August 2004.
- [2] T. Bray, J. Paoli, and C. Sperberg-McQueen. Extensible Markup Language (XML) 1.0. *W3C*, Feb. 1998. <http://www.w3.org/TR/1998/REC-xml-19980210>.
- [3] F. E. Bustamante, G. Eisenhauer, K. Schwan, and P. Widener. Efficient wire formats for high performance computing. In *Proceedings of Supercomputing 2000*, pages 64–64, 2000.
- [4] K. Chiu, M. Govindaraju, and R. Bramley. Investigating the limits of SOAP performance for scientific computing. In

Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing HPDC-11 2002 (HPDC'02), page 246. IEEE Computer Society, 2002.

- [5] D. Davis and M. Parashar. Latency performance of SOAP implementations. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 407–412, 2002.
- [6] C. Kohlhoff and R. Steele. Evaluating SOAP for high performance business applications: Real-time trading systems. In *Proceedings of WWW2003*, Budapest, Hungary, 2003.
- [7] V. P. Ranganath, D. Sexton, and D. Andresen. LYE: high performance SOAP with multi-level caching. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems, PDCS 2004*, pages 743–748. IASTED, November 2004.
- [8] Simple object access protocol (soap) 1.1, Feb. 2003. <http://www.w3.org/TR/SOAP/>.
- [9] Web Services Description Language (WSDL), 2001. <http://www.w3.org/TR/wsdl>.