Accepted for Publication in Procceedings of *5th IEEE/ACM International Workshop on Grid Computing*, Pittsburgh, November 2004.

Toward Characterizing the Performance of SOAP Toolkits

Madhusudhan Govindaraju[1], Aleksander Slominski[2], Kenneth Chiu[1],
Pu Liu[1], Robert van Engelen[3], Michael J. Lewis[1]
1. Department of Computer Science, State University of New York (SUNY) at Binghamton
2. Computer Science Department, Indiana University
3. Department of Computer Science, Florida State University
{mgovinda, kchiu, pliu1, mlewis}@binghamton.edu, aslom@cs.indiana.edu, engelen@cs.fsu.edu

Abstract

The SOAP protocol underpins Web services as the standard mechanism for exchanging information in a distributed environment. The XML-based protocol offers advantages including extensibility, interoperability, and robustness. The merger of Web services and grid computing promotes SOAP into a standard protocol for the large-scale scientific applications that computational grids promise to support, further elevating the protocol's importance and requiring high-performance implementations. Various SOAP implementations differ in their implementation language, invocation model and API, and supported performance optimizations. In this paper we compare and contrast the performance of widely used SOAP toolkits and draw conclusions about their current performance characteristics. We also provide insights into various design features that can lead to optimized SOAP implementations. The SOAP implementations included in our study are gSOAP 2.4, AxisC++ CVS May 28, AxisJava 1.2, .NET 1.1.4322 and XSOAP4/XSUL 1.1. [1]

Key Words*: Grid Computing, Web Services, SOAP, Scientific Computing, Communication Performance*

1 Introduction

In recent years, Web services have been adopted as the standard underlying architecture for grid systems [8, 10]. Web services based specifications are now widely used to represent, discover and communicate with grid services. The synergy between Web services and grid computing has the potential to greatly simplify the design, development, and deployment of distributed applications over wide area networks with heterogeneous environments and diverse resource characteristics.

[1]This research is supported by NSF Career Award ACI-0133838 and DOE Grant DE-FG02-02ER25526.

SOAP [12] is the most commonly used communication protocol for Web services. SOAP can support a variety of message exchange patterns, including request-response, one way messages, RPC, and peer-to-peer interaction. HTTP and HTTPS are SOAP's most popular transport protocols. XML and HTTP have made SOAP robust, extensible, platform and language independent, and appropriate as an underlying protocol to achieve interoperability among disparate platforms. The expressiveness and extensibility properties of SOAP make the protocol particularly well-suited for heterogeneous environments supporting eclectic applications with diverse characteristics and requirements. SOAP Web service endpoints can decide and customize how the protocol is used, without making limiting assumptions about the capabilities and configuration of potential receivers of SOAP messages.

Increased interoperability and expressiveness comes at the potential expense of performance. Embedding information in text-based XML with descriptive tags requires considerable processing time to generate the meta-data tags and write them into the message, and to read and interpret them at the receiving endpoint. Furthermore, the conversion of in-memory data to and from ASCII-based string format, which SOAP requires, can be an expensive step, especially for scientific data. Together, this functionality (reading and writing tags and converting in-memory data into and out of ASCII format) comprises *serialization* and *deserialization* of SOAP messages, which have been shown to be significant components of SOAP communication overhead [4].

SOAP implementations are interesting and important to compare, contrast, and study for three different reasons:

1. Web services based grid applications place disparate requirements on their communication substrate. The requirements can include features such as high end-to-end performance, serialization or deserialization efficiency, small memory footprints, specific security requirements, chunking and streaming capability, minimal toolkit overhead, scalability, and support for opti-

mized handling of scientific data structures. Diverse application requirements lead to a wide range of different implementation choices.

2. Various individual features of SOAP require clever implementation techniques to achieve improved performance. Often, the naive implementation leads to considerable processing time. We discuss several examples in Section 2.

3. The number of SOAP implementations and toolkits is both large and growing. SOAP toolkits [15] exist in languages such as C, C++, Java, C#, Perl and Python. Many efforts are currently underway to develop new ones [3, 9, 16].

In this paper, we describe the features of SOAP for which interesting implementation strategies exist, or are required. We provide an overview of SOAP's characteristics and implementation issues in Section 2. Section 3 then characterizes the performance of several different current SOAP implementations. We focus on SOAP's performance on scientific data, which will become increasingly important for grid applications. Our results show that compliance with the multi-ref specification, as done by gSOAP, incurs a significant performance penalty for large arrays of strings. For small array sizes, XSOAP4 performs better than the .NET framework. For scientific data, such as arrays of integers and doubles, gSOAP is faster than the other toolkits we tested. XSOAP4 outperforms AxisJava and AxisJava-Streaming for all data types that we tested.

2 Designing a SOAP Toolkit

In this section we describe SOAP features that affect the performance of SOAP toolkits. Realizing these features can be achieved in a variety of different ways. We also describe a representative set of implementation techniques that are included in some of the SOAP implementations we study.

2.1 An Introduction to SOAP

SOAP is a lightweight protocol that provides an extensible XML framework for message exchange in a distributed environment. SOAP is not tied to any programming language and can be used over a number of transport protocols. In recent years, SOAP has emerged as the standard communication protocol for Web services.

The current specification of SOAP, version 1.2, is a W3C recommendation. A SOAP message is formally specified as an XML Infoset [6], which is an abstract description of the contents of the SOAP message. XML 1.0, a tree-oriented data representation language, is the most commonly used on-the-wire representation of the infoset. Consequently, the data model for SOAP can use XML to encode data types used in the message. The SOAP-Envelope, which contains namespace information, provides a vocabulary for message content of the SOAP payload. The SOAP specification allows for the use of any transport protocol. Due to the ubiquitous use of HTTP on the Internet, an explicit binding of SOAP messages with HTTP is available with the SOAP specification. The SOAP specification defines a convention for mapping RPC calls over the XML messaging framework. This convention includes mapping of method signatures into request response structures specified in XML. SOAP encoding rules define precisely how commonly used data structures can be represented in XML. This allows SOAP toolkits to automatically translate SOAP calls to method invocations at runtime. Apart from RPC, SOAP supports various programming models including request-response and one-way messaging. The XML-based messages can be packaged in one of the following two ways: *RPC* and *document* style. The packaging style is also referred to as SOAP binding. In RPC style, the name of the operation, to be invoked on the service, appears in the SOAP message. The receiver can dispatch the message to the appropriate implementation of the method. In document style, the SOAP message does not indicate which method it is intended for. It is expected that an a priori agreement exists between the sender and receiver on how to interpret the XML message.

A WSDL document specifies the use of SOAP binding to be either *encoded* or *literal*. The use of *encoded* implies that the XML constructs defined in the WSDL document are an abstract specification of the structure of the SOAP message, and the SOAP encoding rules in the message should be used to interpret the XML constructs. The use of *literal* indicates that the XML constructs in the WSDL document refer to the exact XML format of the SOAP body. In the former case, a SOAP toolkit has to process the encoding rules at run-time to serialize and deserialize in accordance with the rules. The *encoded* style has been made optional in SOAP 1.2. It is expected that the *literal* style will be most widely used in the future.

2.2 The Role of HTTP

HTTP is the most widely used transport protocol with SOAP. HTTP 1.0 requires that the size of the payload (XML encoded data used in SOAP) be specified as part of the *Content-Length* field of the HTTP header. Some HTTP 1.0 servers can calculate the content length from other fields in the message, but most SOAP based HTTP 1.0 servers return an error if the content length is not specified. A simple implementation of SOAP with HTTP 1.0 uses separate buffers for the HTTP header and for the serialized data in XML format. Once the XML payload has been formed, its length is determined and the value is placed in the HTTP header. At this

point, the two buffers could be concatenated to create one large buffer, but this would invoke expensive memory operations. bSOAP [1, 2, 3] addresses this by providing the option of using a *vectored send* call, if available, which allows multiple buffers to be sent via a single system call. Also, sending this buffer over the network incurs a significant performance penalty if the size of the buffer is large and exceeds the size of the system cache. In earlier work, we showed that the SOAP protocol increases message sizes (as compared to the corresponding binary representation) by a factor of four to ten [11]. As a result, the memory requirement for SOAP can have a significant effect on performance, particularly for scientific data such as large arrays. gSOAP avoids expensive buffer copying operations by using a two-iteration algorithm; the first iteration traverses the data structures and calculates the length of the required buffer and in the second iteration it outputs the HTTP header and serializes the SOAP message directly over TCP/IP.

One way to address the performance limitations of HTTP 1.0 is to use the HTTP 1.1 specification. HTTP 1.1 provides explicit support for chunked encoding of messages, which enables overlapping the serialization process with the network transmission of buffers [18]. HTTP 1.1 also has support for *persistent connections* (HTTP keep-alive) which complements the use of chunked encoding. This feature allows the reuse of the same TCP/IP connection to send all the chunks of a single message. It also eliminates the overhead of establishing a new network connection for each call.

HTTP 1.0 requires the content length of the entire XML payload to be specified in the header of the message, but HTTP 1.1 requires that each chunk be preceded by its own size. The size of the chunk is typically a configurable parameter that depends on the size of system-cache and TCP packet size. If the size of the chunk is too small, then many system calls are required to send the chunks over the network. When the size is too big, the delay to fill the chunk buffer increases and the effectiveness of overlapping communication and communication diminishes. Since system calls are expensive, the SOAP payload should not be divided into too many small sized chunks. However, if a large buffer is used for each chunk, then it may result in cache misses. So, there is tradeoff between benefits of ensuring cache-hits and the cost of invoking many system calls.

2.3 Parsing XML

SAX and DOM: The two most popular ways of parsing XML are DOM and SAX. DOM is a tree structured API, and is implemented by building an object representation of the XML document in memory. The DOM model is ideally suited for cases when the XML document needs to be traversed and modified. In contrast, SAX (Simple API for XML) based XML parsers, use the callback model to send out events as the parser encounters various elements of the document. For large documents, that do not fit in memory, the SAX model is more efficient. Also, the SAX model is suited for applications that are interested only in a few specific elements of the document.

Pull Parsing: The pull parsing technique is optimized for cases when the XML elements need to be accessed in succession, and elements that have been parsed before, do not need to be visited again. SOAP toolkits that use a pull-model based parser can efficiently split an XML stream into small sized chunks. The pull parser can build a partial XML Infoset tree in memory in an incremental manner, allowing applications to start processing the XML content even before the entire document has been parsed. An implementation of this technique can be found in the XML Pull Parser (XPP3) [13].

Buffering: Look-aside buffering schemes can be used to optimize the parsing and storage of frequently encountered XML constructs. gSOAP uses look-aside buffers to speed up XML parsing by reusing the memory allocated for storing attribute name/value pairs. This improves the performance of parsing the `xsi:type` attribute which may be present in every XML element of the SOAP payload. Similarly, XPP3 caches parsed strings and avoids strings allocations for processing XML input with values that repeat frequently, such as in the case of arrays.

Namespaces: XML namespace handling is an important issue. XML Namespaces allow tags with identical names to be distinguished by placing them in separate namespaces. Each namespace is associated with a defining namespace name (URI). In the XML payload, tag names are qualified in XML through the use of a prefix. The prefix is bound to the URI by declaring the namespace binding with a special attribute `xmlns`. Parsing namespace info requires maintaining a namespace stack to store namespace prefix/URI pairs. The number of defining *xmlns* namespace bindings in an XML message is typically much smaller than the number of uses of this namespace prefix. Therefore, to optimize the expensive maintenance of a stack, gSOAP parses and processes each `xmlns` attribute with one table lookup to find the namespace prefix that gSOAP internally uses for qualified tags. These internal prefixes are pre-determined from the XML schemas of the SOAP messages. The namespace stack simply records the translated prefix to enable efficient matching of qualified tags without having to store and compare namespace URIs.

Multi-ref: Parsing multi-ref elements is required by SOAP. To efficiently represent data structures, such as cyclic graphs, SOAP encoding rules allow the use of multi-reference accessors, with *id-ref* attributes. The SOAP 1.1 encoding rules place all multi-ref accessors at the end of a message. This means that all references in an XML message are *forward pointing*. An efficient streaming SOAP processor must wait until the multi-ref elements are parsed at the end of a message to deserialize data structures exhibiting co-

referenced objects. Note that a SOAP processor based on a DOM can make multiple passes over the DOM to resolve the pointers. However, a DOM introduces significant overhead to store and process SOAP messages. A naive implementation of the multi-ref feature can hurt the scalability of the serialization process.

2.4 Generating XML

Serialization: Serialization (and deserialization) of scientific data via SOAP can account for 90% of the end-to-end time [4]. bSOAP [1, 2, 3] addresses this performance bottleneck by saving a template of an outgoing message, and in subsequent calls serializes only those values that have changed since the previous send. This optimization, called *differential serialization*, results in significant performance improvements for clients that repeatedly send similar requests to the same Web service.

To produce relatively simple SOAP messages, implementations may use the Libc `printf` family of functions to generate XML output using a template-based approach. However, this family of functions can be computationally expensive to generate large XML documents, and the overhead can be higher than the cost of producing XML output from a DOM, due to internal buffering [19].

Namespaces and tags: Namespace-qualified element and attribute tags can be emitted without namespace table lookup when the namespace prefixes associated with namespace names (URIs) can be determined in advance. This strategy is used by gSOAP [20]. gSOAP serialization code includes fixed qualified XML element tag and attribute names to speed up XML generation. XSOAP4, on the other hand, uses XPP3's specialized XML writer/serializer class to generate valid XML by maintaining list of current namespace prefixes and URIs.

Memory Footprint: Memory footprint can be a deciding factor for the deployment of Web service applications on embedded devices and on high-demand servers. A large memory footprint impacts the performance of a multi-threaded server that has to serve many concurrent calls. In addition, any increase in dynamic memory usage can negatively affect the effectiveness of data cache. While Java's garbage collection alleviates memory allocation from a programmer's point of view, it can also be an adversary to performance-critical applications.

gSOAP employs various techniques to reduce memory footprint [18], including the use of a schema-specific parsing techniques to reduce memory requirements for parsing XML using one pre-allocated buffer for I/O, look-aside buffers to store XML content, and HTTP compression to reduce message size. For example, a Google API client call can be implemented with only 2.4K of dynamically allocated memory [18].

2.5 Performance Tradeoffs

Performance versus portability: portability requirements often conflict with performance requirements. Certain machine features and platform-dependent library functions cannot be used to increase the overall performance of portable software. For C/C++ toolkits, the use of `sprintf`, `writev`, and even `strtod` is not portable to a small Operating Systems such as Palm OS or WinCE. Though Java code is portable, low-level socket programming in Java to develop efficient Java SOAP toolkits may take a performance hit. Therefore it is important to design SOAP toolkits with pluggable network transport modules.

Performance versus interoperability: the SOAP protocol has many nuances with respect to serialization formats, especially with respect to the encoding style. The same data content can have slightly different XML representations, due to use of multi-ref accessors, nil elements, position attributes in sparse SOAP encoded arrays, `xsi:type` attributes, and minor differences in XSD types of different XML Schema specifications.

Performance versus dynamic invocation: SOAP toolkits with a dynamic invocation interface (DII), such as those that use Java's *dynamic proxy* feature [14], are inherently slower compared to SOAP toolkits that statically generate code. However, the DII approach allows clients to adapt more easily with changes in the service definitions, such as when service endpoints and operations are changed.

2.6 Compressing SOAP Payloads

HTTP compression: Real-time compression seems like a good candidate to reduce bandwidth requirements, because HTTP gzip compression can reduce the SOAP message length to a size that is comparable to the size of the compressed form of the original binary data [17]. However, our performance study [17] indicates that real-time gzip compression is computationally expensive and exceeds the combined cost of XML serialization and data transport over LANs. Therefore, compression of SOAP messages is only useful when the network bandwidth is very limited. However, even low-bandwidth networks such as those including modem pools do not yield significant performance gains with compressed HTTP transfers because v.90 modems already apply compression.

XML compression: Compressed XML formats can yield a performance gain if the overhead of the compression algorithm is relatively low compared to XML serialization. XML compression tools, such as XMill [22], use methods that exceed the compression rate of gzip on XML. The use of XMill's compression methods for real-time compression may improve the performance of SOAP. More efforts in the

XML community are under way to address XML compression standardization [21].

2.7 SOAP Attachments: DIME and MIME

SOAP DIME attachments provide an efficient means to exchange raw binary data. While binary data transfers are often best utilized for platform-independent streaming media types, such as images and sound, binary numerical data exchange is platform dependent due to the differences between big- and little-endian architectures and internal floating point representations. DIME attachments can also be streamed [17], which means that the content of DIME attachments does not require persistent storage. This approach enables data transfers rates that exceed Java RMI and IIOP. SOAP MIME attachments cannot be streamed and require a significant amount of pre-processing to determine MIME boundaries at both the sending an receiving sides. Therefore, MIME attachments are less suited for performance-critical data transfers.

2.8 Support for Scientific Data

The SOAP specification itself does not have inherent support for scientific data. Our earlier study showed that the standard and interoperable way of translating numeric data into decimal text formats can account for upto 90% of the end-to-end time for scientific data [4]. This overhead of XML encoding can be addressed by exploiting XML schema extensibility to define optimized XML data representations for numerical data [17], e.g. base64 encoded IEEE754, or by using attachments. However, attachments place the scientific data outside the XML framework of SOAP requiring another data format description to deal with it. In earlier work, we presented some techniques to improve performance of SOAP toolkits for scientific data, including *differential serialization*bSOAP [1, 2, 3], use of *tries* [4], and schema specific parsing [5, 19].

2.9 Concurrency Control

Multi-threading: SOAP Web service applications are usually concurrent servers that are based on Apache's httpd or IIS. Some SOAP toolkits include stand-alone Web service capabilities for supporting the software as a service (SaaS) paradigm [7]. When a service is deployed as a stand-alone application, optimized threading strategies can be employed, such as thread pooling and task farming to improve performance. Thread pooling avoids the overhead of spawning and joining threads for every SOAP request by reusing the thread after it has completed a previous request.

Scheduling: Servers that must operate under high loads can use task scheduling policies, such as, improving performance by first serving requests with a smaller load. The load

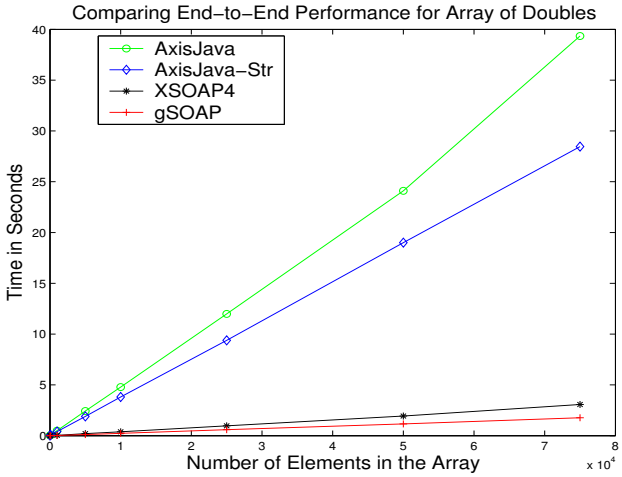

Figure 1. End-to-End performance for arrays of doubles. AxisJava is slower than AxisJava-Streaming by a factor of 1.2 to 1.4. XSOAP4 is faster than AxisJava-Streaming by a factor of 9 to 13. gSOAP is faster than XSOAP4 by a factor of 1.24 to 1.76.

can be determined from the HTTP content length of the request when there is a strong correlation between the SOAP message size and the complexity of the request (HTTP headers include message content length). Or it can be determined from the type of operation requested.

3 Performance

We conducted experiments to study the performance of the following toolkits: gSOAP 2.4 [17, 18, 20], XSOAP4/XSUL [11, 13, 14], AxisJava 1.2 (with and without streaming enabled) [16], AxisC++ 1.1.1 [16] and .NET 1.1.4322. The .NET framework was run on a Dell Dimension 4500 with Intel Pentium 4 2.26GHz processor, 1GB of DDR SDRAM and 80GB Ultra ATA/100 hard drive. The other toolkits were run on dual processor 2.0 GHz Pentium 4 Xeon machines with 1 GB DDR SDRAM and a 15K RPM 18GB Ultra-160 SCSI drive running Debian Linux version 2.4.24. The machines were connected by a gigabit ethernet switch. gSOAP and AxisC++ were compiled with gcc 2.95.4 with optimization flag "-O2." AxisJava and XSOAP4 were compiled with Java 1.4.2_04.

Experiments

We measured the performance of SOAP toolkits for various workloads commonly used in scientific computing, such as varying array sizes of doubles, integers and strings. Figure 1 compares the performance for arrays of doubles. AxisJava-Streaming (labeled AxisJava-Str) uses persistent

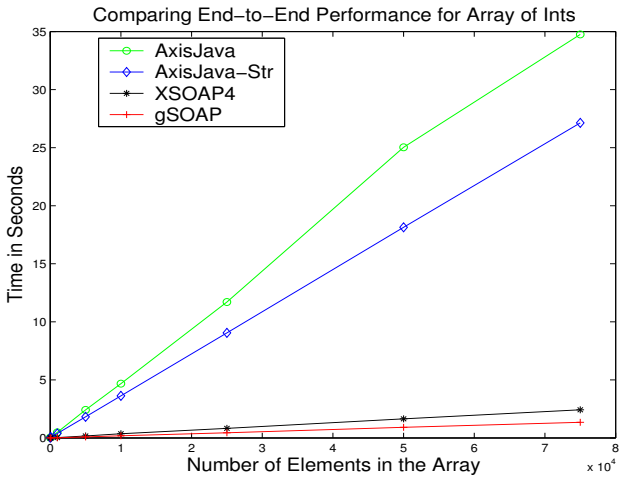

Figure 2. End-to-End performance for int arrays. AxisJava-Streaming is faster than AxisJava by a factor of 1.06 to 1.33. The difference between gSOAP and XSOAP4 increases with increase in the size of the array.

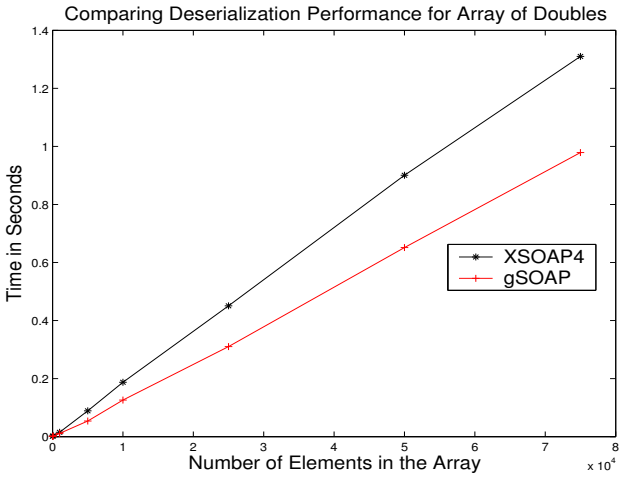

Figure 3. Performance for deserialization of XSOAP4 and gSOAP for double arrays. For array sizes close to 10 elements the difference is 3%, but increases to 32% for larger arrays.

connections along with chunking and streaming to enhance performance, and as a result is faster than AxisJava by a factor of 1.2 to 1.4. However, XSOAP4, which is also a Java based toolkit, is faster than AxisJava-Streaming by a factor of 9 to 13. gSOAP, which is a C/C++ based toolkit, is faster than XSOAP4 by a factor of 1.2 to 1.76.

Toolkit	gSOAP	XSOAP4	AxisC++	.NET	AxisJava
Latency	0.0013	0.0016	0.0027	0.0034	0.0101

Table 1. Time in seconds for the *void echoVoid()* method. The performance shows the latency imposed by each toolkit for a SOAP call.

Table 1 shows the overhead imposed by each toolkit to send and receive a SOAP call that does not pass any parameters (referred to as latency in the table). For applications that that use SOAP to send small amount of data, such as events and error reporting, latency is an important factor in deciding which toolkit to use.

In Table 2, we compare the performance of AxisC++ with gSOAP, another C/C++ based toolkit. We had stability problems with AxisC++ which prevented us from completing the tests for arrays of more than 1000 elements. Table 2 shows that for end-to-end performance (labelled as *echoDoubles* in the table) gSOAP is faster than AxisC++ by 44% to 51%. For deserialization, gSOAP performs faster by 48% to 59%.

Figure 2 compares the end-to-end performance for arrays of integers. AxisJava-Streaming is faster than AxisJava by a factor of 1.06 to 1.33. gSOAP is faster than XSOAP4 by a factor of 1.1 to 1.8. XSOAP4's performance decreases as compared to gSOAP, with increase in the array size. This trend can be observed even in Figure 3, which compares the deserialization performance of gSOAP and XSOAP4. For small array sizes (close to 10), gSOAP is faster by 3%, while for array sizes close to 75,000, gSOAP is faster by 32%. XSOAP4 uses a pull-based parser (XPP) that is optimized for parsing small sized data whose structure is known in advance. Currently, XSOAP4 does not support streaming and instead loads data into memory creating a DOM like tree for conversion between ASCII and binary format of the data

Toolkit ⇓	array-size ⇒	10	100	1000
AxisC++	echoDoubles	0.0031	0.0057	0.0366
	deserDoubles	0.0028	0.0047	0.0215
gSOAP	echoDoubles	0.0015	0.0029	0.0204
	deserDoubles	0.0013	0.0019	0.0111

Table 2. The table shows performance in seconds for AxisC++ and gSOAP, for array sizes 10, 100 and 1000. The two features tested were end-to-end performance for double arrays and deserialization of double arrays.

Toolkit ⇓	array-size ⇒	100	10000	50000
.NET	echoStrings	0.0052	0.1904	1.0836
	receiveInts	0.0046	0.1085	0.5467
XSOAP4	echoStrings	0.0045	0.2738	1.3549
	receiveInts	0.0032	0.1207	0.6061

Table 3. The table shows performance in seconds for two Java based SOAP implementations: .NET and XSOAP4. For array sizes less than 100, XSOAP4 performs better than .NET.

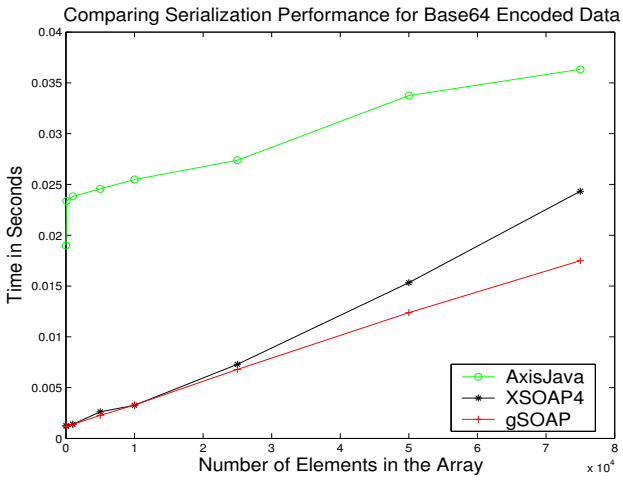

Figure 4. Serialization performance for array of base64 encoded data. XSOAP4 is faster than AxisJava by a factor of 1.6 to 6. For large array sizes, gSOAP performs better than XSOAP4 by upto 38%.

types. For large array sizes, this design has a significant effect on the performance.

In Table 3, we compare the performance of two SOAP implementations on Windows platform: .NET and XSOAP4. For large array sizes, .NET performs better, while for array sizes less than 100, XSOAP4 performs slightly better.

One way to address performance limitations of SOAP is to serialize binary data using Base64 encoding. Serialization routines for base64 encoded data need to keep one large string in memory and write it to the socket layer. Figure 4 shows performance for serialization of base64 encoded data. AxisJava is slower than XSOAP by a factor of 6 for small array sizes, but for vary large array sizes the difference is a factor of 1.6. For small array sizes, gSOAP is faster than XSOAP4 by 3%. However, as the array size increases, gSOAP's performance is better by upto 38%.

In Figure 5, we compare the performance of serialization of string arrays. For array sizes greater than 50,000, the performance of gSOAP degrades due to the the multi-ref encoding algorithm. gSOAP's string serializer checks for co-referenced strings using a hash table for every string to be serialized. Even though the hash table is fast, it does not necessarily require constant time per check. Depending on the string alignments for the memory allocated using *malloc()*, it can affect the results due to the hash table's overflow chains. This is an example of an interoperability/performance tradeoff choice. gSOAP guarantees that the logical coherence of graph structures is preserved during serialization. This observation is consistent with the scalability of gSOAP for end-to-end performance of doubles and integers, which do not require multi-ref serialization.

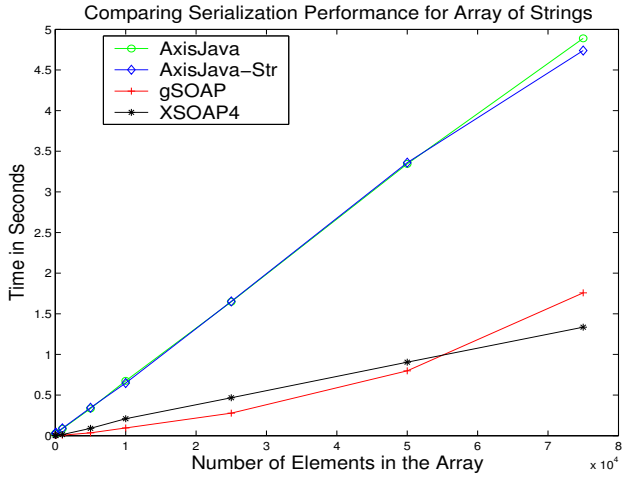

Figure 5. Serialization Performance for string arrays. gSOAP checks for co-referenced objects, to enforce multi-ref rules of SOAP, and so its performance drops for large array sizes.

4 Summary and Future Work

In this paper we provided insights into various features of SOAP that affect its performance. We studied the performance of some representative SOAP toolkits for scientific data structures. These results will aid in the design and development of new SOAP toolkits, and guide users in choosing the appropriate toolkit for their current application requirements.

We did not test and compare memory footprints of different toolkits. Memory footprint can be a deciding factor for the deployment of Web service applications on embedded devices and on high-demand servers. A large memory footprint impacts the performance of a multi-threaded server that has to serve many concurrent calls. In the near future, we plan to study the effect of memory footprint on performance.

We also plan to test the performance of SOAP for emerging security standards.

We are currently working on the design of a benchmark suite to test and assess the performance and scalability of different SOAP toolkits. The benchmark suite consists of a set of workloads that are designed to exercise features discussed in Section 2. We plan to make the benchmarks and associated drivers publicly available to the SOAP community.

References

[1] N. Abu-Ghazaleh, M. Govindaraju, and M. J. Lewis. Optimizing Performance of Web Services with Chunk-Overlaying and Pipelined-Send. *Proceedings of the International Conference on Internet Computing (ICIC)*, pages 482–485, June 2004.

[2] N. Abu-Ghazaleh, M. J. Lewis, and M. Govindaraju. Performance of Dynamic Resizing of Message Fields for Differential Serialization of SOAP Messages. *Proceedings of the International Symposium on Web Services and Applications*, pages 783–789, June 2004.

[3] N. Abu-Ghazaleh, M. J. Lewis, and M. Govindaraju. Differential Serialization for Optimized SOAP Performance. *Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing (HPDC-13)*, pages 55–64, June 2004, Honolulu, Hawaii.

[4] K. Chiu, M. Govindaraju, and R. Bramley. Investigating the Limits of SOAP Performance for Scientific Computing. In *Proceedings of HPDC-11*, pages 246–254, Edinburgh, Scotland, July 23-26, 2002.

[5] K. Chiu and W. Lu. A Compiler-Based Approach to Schema-Specific XML Parsing. In *First International Workshop on High Performance XML Processing (Satellite of WWW2004)*.

[6] J. Cowan and R. Tobin. XML Information Set, W3C Recommendation, October 2001. `http://www.w3.org/TR/xml-infoset`.

[7] K. B. et al. An architectural model for service-based software with ultra rapid evolution. In *proceedings of IEEE International Conference on Software Maintenance (ICSM 01)*, pages 292–300. IEEE CS Press, 2001.

[8] I. Foster, C. Kesselman, J. Nick, and S. Tuecke. Grid Services for Distributed System Integration. *Computer 35(6)*, 2002.

[9] Globus Alliance. ScOAP. `http://www-unix.globus.org/scoap/index.html`.

[10] Globus Alliance, IBM and HP . The WS-Resource Framework, 2004. `http://www.globus.org/wsrf/`.

[11] M. Govindaraju, A. Slominski, V. Choppella, R. Bramley, and D. Gannon. Requirements for and Evaluation of RMI Protocols for Scientific Computing. In *Proceedings of SuperComputing 2000*, November 2000.

[12] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, Canon, and H. F. Nielsen. Simple Object Access Protocol 1.1, June 2003. `http://www.w3.org/TR/SOAP`.

[13] Indiana University, Extreme! Computing Lab. Grid Web Services. `http://www.extreme.indiana.edu/xgws/`.

[14] A. Slominski, M. Govindaraju, D. Gannon, and R. Bramley. Design of an XML based Interoperable RMI System : SoapRMI C++/Java 1.1. In *Proceedings of PDPTA*, pages 1661–1667, June 25-28, 2001.

[15] SoapWare.Org. The Leading Directory for SOAP 1.1 Developers. `http://www.soapware.org/directory/4/implementations`.

[16] The Apache Project. Axis Java. `http://ws.apache.org/axis/`.

[17] R. van Engelen. Pushing the SOAP envelope with Web services for scientific computing. In *proceedings of the International Conference on Web Services (ICWS)*, pages 346–352, Las Vegas, 2003.

[18] R. van Engelen. Code generation techniques for developing light-weight efficient XML Web services for embedded devices. In *proceedings of 9th ACM Symposium on Applied Computing SAC 2004*, Nicosia, Cyprus, 2004.

[19] R. van Engelen. Constructing finite state automata for high performance XML web services. In *Proceedings of the International Symposium on Web Services (ISWS)*, Las Vegas, 2004.

[20] R. A. van Engelen and K. Gallivan. The gSOAP Toolkit for Web Services and Peer-To-Peer Computing Networks. In *The Proceedings of the 2nd IEEE International Symposium on Cluster Computing and the Grid (CCGrid2002)*, pages 128-135, May 21-24, 2002, Berlin, Germany.

[21] W3C. Xml binary characterization. http://www.w3.org/XML/Binary.

[22] H. Westra. XMill. http://sourceforge.net/projects/xmill.