

SOAP for High Performance Computing

Kenneth Chiu(chiuk@cs.indiana.edu) *

Madhusudhan Govindaraju (mgovinda@cs.indiana.edu)

Randall Bramley (bramley@cs.indiana.edu)

Abstract

The growing synergy between Web Services and Grid-based technologies [10] will potentially enable profound, dynamic interactions between scientific applications dispersed in geographic, institutional, and conceptual space. Such deep interoperability requires the simplicity, robustness, and extensibility for which SOAP [7, 6] was conceived, thus making it a natural *lingua franca*. Concomitant with these advantages, however, is a degree of inefficiency that may limit the applicability of SOAP to some situations. In this paper, we investigate the limitations of SOAP for high-performance scientific computing. We analyze the processing of SOAP messages, and identify the issues of each stage. We also present a high-performance SOAP implementation and a schema-specific parser based on the results of our investigation. Our results assist the assessment of SOAP's ability to meet a given performance requirement. Our implementation is in C++, but we believe that the results have wider application.

Keywords: SOAP, Web Services, Grid, scientific computing, high performance, schema, parser.

1 Introduction

The growing synergy between Web Services and Grid computing potentially heralds the realization of distributed computing's promise. This promise depends in part on interoperability that is semantically deep but syntactically shallow. That is, the system must be loosely-coupled, but when coupling does occur, it must be at a deeper level than that typified by applications such as SETI@Home. Like natural languages, minor variations in syntax should not change the essential contents of a message.

The SOAP protocol¹ was conceived expressly to support such interoperability in-the-large. Designed using principles learned from HTTP and HTML, it facilitates interdependent interactions between otherwise independent entities. SOAP is also the standard binding for the the emerging Web Services Description Language [5].

*Department of Computer Science, 150 S Woodlawn Avenue, Bloomington, IN 47405. Ph: 812 855 8305
Fax: 812 855 4829

¹Though SOAP was initially an acronym for Simple Object Access Protocol, the current specification states that it is no longer.

The qualities that make it ideal for Grid services, however, also make it less suitable for scientific computing. Scientific computations demand reliable transfer of data in distributed heterogeneous environments. These can consist of parallel programs sending large, complex and rapidly changing data objects or self-contained modules sending events to steer other modules. Scientific systems also have complex run-time systems designed for heterogeneous environments with dynamically varying loads, multiple communication protocols, and differing Quality of Service (QoS) requirements. This diversity presents a unique set of challenges. The communication medium plays a crucial role in the success of engineering and computational systems, and needs to handle high-performance messaging and leverage the benefits of high-speed networks like Abilene, MREN, or ESNET.

To combine the strengths of different systems, a user may want to develop an application that has components from several of them, using the strength of each wherever appropriate. Since these systems can be connected via more than one protocol, a common denominator protocol is needed to negotiate a more specialized protocols. The criteria for an effective communication protocol include reliability, robustness, readability, ease of use, seamless integration with existing computational code and interoperability. In earlier work [11] we showed that SOAP [7, 6] can be used to build a reliable multi-protocol RMI system and meets this criteria.

The ease and degree of interoperability between components is dependent on the format of data exchanged between systems. The Extensible Markup Language (XML) [4] has been gaining acceptance as a canonical data representation. HTTP is a ubiquitous network protocol used extensively over the Internet. SOAP does not mandate an underlying transport protocol, but HTTP has emerged as the most widely used one for SOAP. Since SOAP can combine the strengths of XML and HTTP, it is an attractive candidate for communication between scientific computations on the Grid.

SOAP can also be used for providing fail-over implementations for distributed scientific systems. Although many implementations of SOAP exist, few have made inroads into exploring the performance limits achievable for scientific applications. A clear understanding of the bottlenecks and areas of improvement in implementation will result in better integration with Web Services and Grid based frameworks.

When operating in fail-over mode the SOAP implementation may have to handle all types of data. Thus, a study of SOAP performance for scientific data is imperative. While it is known that the performance of SOAP based systems is not efficient for high performance computing, it is important to learn the exact performance penalty for using it. A clear understanding of the inherent bottlenecks and areas of improvement can provide better integration with Web Services and Grid based frameworks.

In this paper we propose the design of a SOAP implementation suitable for systems with stringent memory and bandwidth requirements. We present a thorough analysis of all performance issues and the results that can be used to examine and develop solution strategies for other systems. Some of the important issues that we have examined include the use of streaming, “string” operations, conversion routines between strings and primitive data types, memory management and schema-specific parsing.

This paper addresses the following questions:

- How does one design an efficient implementation of SOAP?

- Are there any bottlenecks inherent to SOAP and if so how do they affect performance?
- What is the upper limit of performance that can be achieved with SOAP?
- How can one design a high-performance schema-specific parser?

In this paper we state the upper limits of SOAP performance and identify techniques that can be used to achieve this performance. We introduce a high performance XML parser specialized for SOAP and discuss possible ways of improving some existing SOAP based systems.

2 SOAP for Grid and Web Services

SOAP provides mechanism for messaging between a service provider and a service requester. It is an XML based protocol that consists of three parts, an envelope that defines a framework for interpreting and processing the message, a set of encoding rules for defining data-types and a convention for representing remote procedure calls and messages.

Projects like GSI SOAP [18] have designed an implementation that provides a GSI enabled HTTP protocol with delegation and authentication capabilities as part of the security infrastructure for computational grids. Several research efforts have been focusing on using XML Schemas [2] and SOAP as an effective data format for event bases systems on the Grid. The Grid Forum Performance Monitoring Group [9] has been studying the performance of the Grid Monitoring Architecture [1] that has events defined as XML Schemas. In an earlier work we discussed a SOAP based event system for the Grid [16].

3 Sending SOAP Messages

3.1 Stages

The sending of a SOAP message can be divided into several stages. These stages may not be directly represented in the code, but still provide context useful for discussion and analysis.

1. Traverse data structures representing message.
2. Convert machine representation of data to ASCII.
3. Write ASCII to buffer.
4. Initiate network transmission.

3.1.1 Stage 1

A SOAP message begins as some kind of data structure. Stage 1 traverses this structure to impart a corresponding structure to the SOAP XML. This traversal is not a significant part of the serialization, because each leaf element can be traversed with simple operations such as member offset calculations, array indexing, or pointer following.

3.1.2 Stage 2

The strings or numbers that comprise the actual data are usually in machine representation, and are converted in Stage 2 to the ASCII form required by XML. For strings already in ASCII, conversion is simple and fast. Strings in UNICODE may require some processing to convert to UTF-8/16. The majority of characters used in scientific computing fall within the ASCII range, and therefore require minimal processing to convert to UTF-8/16.

Integers are usually in two's-complement representation. Conversion to ASCII involves a binary-to-decimal conversion. Floating-point numbers are usually in IEEE-754 representation. Conversion to ASCII also requires a binary-to-decimal conversion, but the floating-point conversion is considerably more complex than the integer conversion.

3.1.3 Stage 3

The ASCII form of the data is stored, along with the appropriate XML tags, to a memory buffer in Stage 3. Exactly how the ASCII is stored can affect the number of memory operations required. For example, if an integer element is serialized with

```
printf("<integer>%d</integer>", i);
```

each character of the start tag must first be read from memory, then written to the buffer. However, if the start tag is created with a sequence of statements such as

```
*buf = '<';  
*buf = 'i';  
*buf = 'n';
```

the characters comprising the start tag will likely already be in the instruction stream as immediate operands.

3.1.4 Stage 4

Finally, in Stage 4 the operating system transmits the contents of the memory buffer. This requires a system call, which is relatively expensive, so the buffer should be flushed sparingly. Using a buffer that is too large to fit in cache, however, may increase cache misses.

When using HTTP 1.0 [12], the length of the body must be specified in a `Content-Length` header field. Because this value cannot be determined until the SOAP message is serialized, the straightforward implementation would (1) use two separate buffers, one for the header and one for the body, and (2) serialize the complete SOAP message before completing the header. This can require two system calls, and consume much memory if the message is large.

The first issue can be resolved by either back-patching or vectored sends. If we insert spaces for the content length during the initial header generation, we can later replace the spaces with the actual content length once the SOAP message has been processed. Alternatively, we can use a vectored send (available on both UNIX and Win32 machines) that will concatenate multiple memory buffers in one send call.

The second issue can be resolved by either using HTTP 1.1 (discussed below), or using a two-step serialization. In the first step the length of the body is calculated without actually

storing to the memory buffer. The header can then be completed and the body serialized into the memory buffer. Because the memory buffer does not need to hold the entire body at one time, memory usage is reduced.

The actual transmission is most commonly over TCP/IP. Since TCP/IP requires one packet exchange before transmission can begin, establishing a separate connection for each message adds a round-trip delay to each message.

In addition to the delay, creating a connection per message consumes operating system resources. The TCP/IP protocol requires that one end of a closed connection remain in `TIME_WAIT` state for twice the maximum segment lifetime. This period can be as long as four minutes. During this period, a certain amount of memory must be maintained, and the port cannot be reused for the same remote host and port[17].

3.2 Design

Our previous experience revealed that the memory usage of SOAP can be prodigious. A typical SOAP message may be 4-10 times the size of the corresponding machine representation. This can be significant for large arrays. Besides being careful not to create extra copies, we chose to address the memory concern by using HTTP 1.1 [13] instead of HTTP 1.0.

HTTP 1.1 also supports chunked encoding. This allows a body to be sent in chunks, with each chunk preceded by the chunk size. The content length is no longer necessary, because a receiver can determine the end of the body by processing the chunks. The elimination of the content length means that the sender does not need to buffer the whole message before transmission, which also allows overlap between network transmission and serialization.

Most operating systems copy the user-space memory buffer to a kernel-space buffer during the send system call, and immediately return. The actual transmission occurs in the background. By calling send multiple times for a large message, overlap between the transmission of the previous buffer by the kernel and the preparation of the next buffer by the program will occur. Of course, calling send too many times will cause the overhead system calls to dominate.

HTTP 1.1 also supports persistent connections, which remain open for multiple messages. This reduces the overhead of creating a new connection for every message.

4 Receiving SOAP Messages

4.1 Stages

Though in some sense receiving a SOAP message is the inverse of sending a SOAP message, the issues are somewhat different. Conceptually, we divide the receiving process into four stages:

1. Read from network into memory buffer.
2. Parse XML.

3. Handle elements.
4. Convert ASCII to machine representation.

4.1.1 Stage 1

The SOAP message is read from the operating system into a memory buffer in Stage 1. This requires a system call, so reading as much data as possible will minimize the number of system calls. If the amount of data is larger than will fit into cache, however, the number of cache misses will increase.

4.1.2 Stage 2

In this stage, the XML is parsed to identify its syntactic constructs. Comments are stripped, start tags located, etc. This parsing normally involves a state machine of some kind. Possible choices are coding the transitions in a switch-statement, or using a table-driven approach.

There are currently two popular paradigms for processing XML, the Document Object Model (DOM) [3] and the Simple API for XML (SAX) [8]. DOM builds a complete object representation of the XML document in memory. This can be memory intensive for large documents, and entails making at least two passes through the data. During the first pass, the object model is constructed. Only after the document is completely parsed can the application interpret the data in another pass.

SAX operates at one level lower. Rather than actually constructing a model in memory, it informs the application of elements through callbacks. This also requires at least two passes through the data. The first pass is performed inside the SAX implementation, and is required to identify tags and element content. The second pass is performed by the application after the XML constructs are pushed to the application through callbacks.

Pull parsing, as exemplified by the XML Pull Parser [15], is an efficient paradigm similar to SAX in that it does not build a complete object model in memory. It differs in that the tags and content are returned directly to the application from calls to the parser, rather than indirectly in the form of callbacks. This provides a more natural API for many applications.

A number of dual-mode models also exist. Progressive DOM, for example, switches between a SAX model and a DOM model depending on the needs of the application at that point in the XML processing.

4.1.3 Stage 3

Once the tags are identified, the content of each tag is interpreted. For the parsing paradigms described above, the parser presents the tag and the content as simply text. So the actual interpretation of an element is completely delegated to the application. This means that even if the application uses an efficient data structure like a red-black tree to match actions to tags, it still must examine the tag after the parser has already made one pass through it.

4.1.4 Stage 4

Ultimately the ASCII text must be converted to the machine representation. For numerical data, this will involve a decimal-to-binary conversion. For string data, it may involve a UTF-8/16 to wide string conversion.

4.2 Design

Our design for receiving SOAP messages rests on two interdependent principles. The first is that we read from the network only when we run out of data, and once we have issued a read, we process until we do run out of data. This means that the stages need to be fully pipelined.

The second principle is that we never examine data more than once. Thus, stages 1-4 should all be performed with one pass.

The fundamental reason the popular parsing paradigms require two passes is that they present a data-centric interface to the application. Thus, for example, the parser must first make one pass to syntactically identify the end of the content. The application then makes another pass to interpret the content. Likewise, the parser first makes one pass to demark the end of a start tag. The application must then examine the tag again to decide how to handle it.

To avoid this we interface to the application through a streamed, push-pull model. Like SAX, we make callbacks to the application. Unlike SAX, however, the parser has already matched the tag to a specific callback. The application therefore does not need to examine the tag again, and in fact the tag may not have ever existed as a complete string anywhere in memory.

One candidate for tag matching is perfect hashing. Perfect hashing, however, may generate a valid hash code for an item not in the hash table. Thus, the item must be reexamined after the hash code has been computed, which requires two passes over the tag.

We therefore elected to use *tries*, which unambiguously determine whether or not a key is valid. A trie is essentially a table-driven DFA for a fixed set of strings. As the parser encounters each character of a tag, it simultaneously feeds it into the trie. Upon reaching the end of the tag, the trie has already matched the tag to a specific handler.

When the handler is called, the content of an element is not presented as data, but as a function that the application can call for the next character. The function parses the XML on-the-fly as the application requests each character of the content. Thus, the parser does not need to make an initial pass through the XML to identify the end of the character data.

Attributes are handled similarly.²

4.3 Schema-Specific Parsers

For a scientific application to interpret the data in a SOAP message, it must have some idea of its structure. This structure may be known in ad hoc manner, or it may be formalized

²Our XML parser does not currently support the full XML specification. For example, CDATA sections are not supported. Namespaces are parsed, but we do not yet have an API for tag handlers within namespaces.

through an XML Schema.

In either case, performance can be improved by using this information to generate a schema-specific parser. For example, rather than representing the tag-matching tries as tables, they can be represented directly in the program itself as code. One can even imagine a parser-generator that directly generates machine-code (or Java byte-code) from a schema.

For our tests, we implemented a limited form of a schema-specific parser. The SOAP encoding rules specify an array form that is easier to parse than general XML. A parser written just for this form can be faster than a general XML parser. By using the schema to direct the parsing, we can use the array parser whenever we are parsing an array.

5 Performance Measurements

We conducted experiments to measure the performance of SOAP for large arrays as well as two different small objects. We varied the size of the double arrays from 10 to 1,000,000. One of the small objects contained a short string and the other contained sixty-four integers as separate elements. We included the smaller objects to verify that our improvements for arrays do not adversely affect performance for other objects. We disabled various features selectively to study the effect of each on performance.

All Solaris machines were running Solaris 8. The code was compiled with the Workshop 5.3 C++ compiler.

All Linux machines were running the Linux 2.4 kernel with Redhat 7.1. The code was compiled with g++ 3.0.3.

All machines were connected via 100 Mbps Ethernet.

All performance profiles were produced with the Sun Forte 6 Update 2 performance tools.

5.1 End-To-End

The end-to-end results include the cost of serialization, deserialization and communication over the network for sending a message between two nodes.

5.1.1 Persistent Connections and Streaming

As can be seen from Figures 1–4, persistent connections significantly improve performance for messages that have less than 100,000 doubles. Because the cost of establishing a connection increases with latency, the benefit of persistent connections would be especially pronounced for a high-latency, high-bandwidth network. However, for larger messages, the cost of establishing socket connections is amortized over many doubles, and not significant.

Streaming improves the performance for large messages, because it allows overlap between communication and deserialization that would otherwise not be possible. This overlap is not significant for small messages since the communication time is very short. The test shows, however, that streaming code does not add a significant amount of overhead for small messages.

Analysis with the Sun Forte 6 tools on a Blade 1000 showed that for messages of 1,000,000 doubles, streaming (via chunking) reduced time stalled on L2 cache misses from .147 to .009

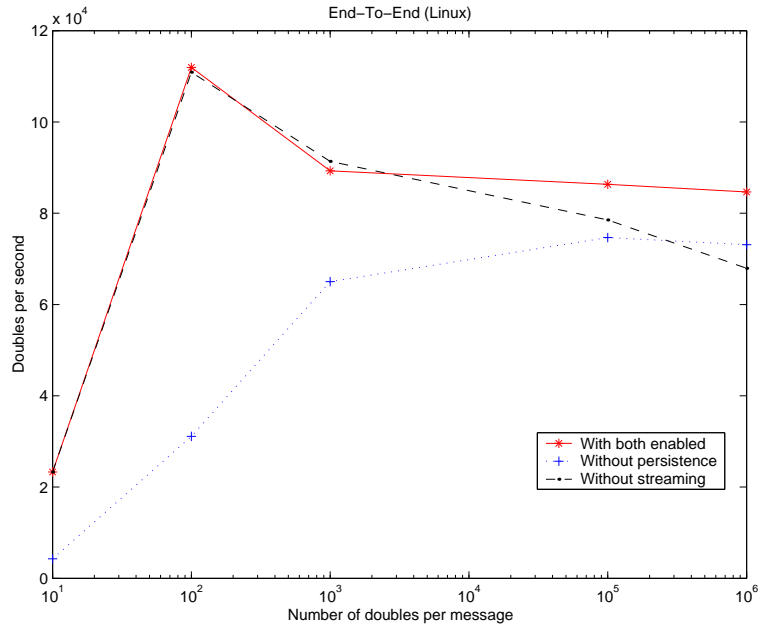


Figure 1: Effect of persistent connections and streaming for array of doubles on a Linux machine. The sender was a Pentium III (Coppermine) with 256 KB of on-die Level 2 cache. The receiver was a Pentium III (Katmai) with 512 KB of off-die Level 2 cache. The peak for arrays of 100 doubles is likely due to cache effects.

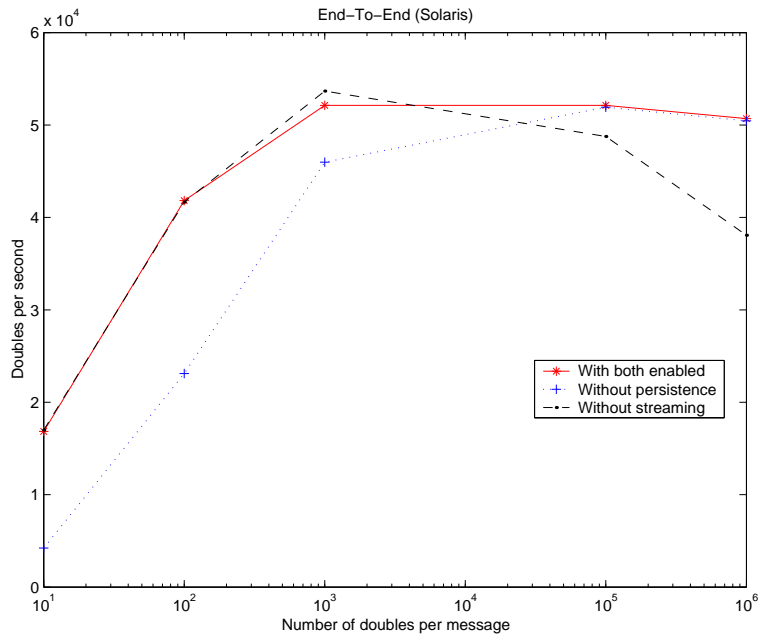


Figure 2: Effect of persistent connections and streaming for an array of doubles on Solaris machines. The sender and receiver were both Sun Blade 100 machines. Each machine had a 500 MHz UltraSPARC-IIe.

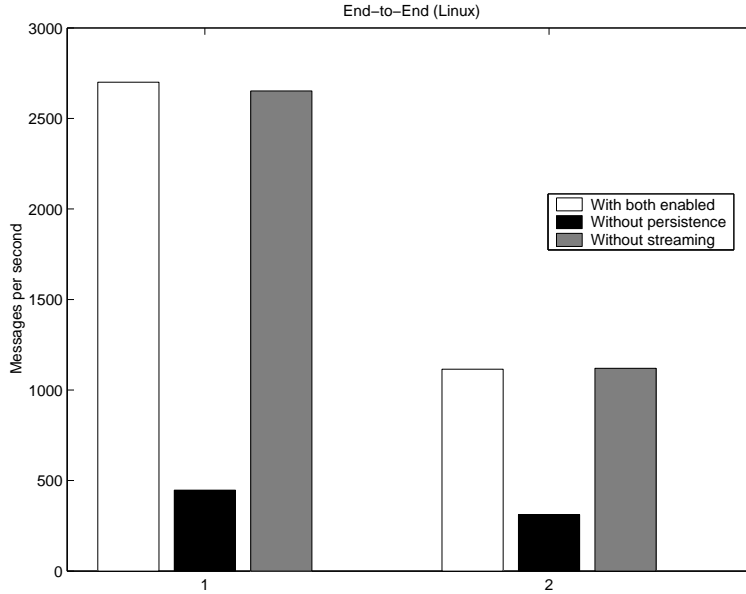


Figure 3: Effect of persistent connections and streaming for (1) a string and (2) an object with 64 integer elements, tested Linux machines. Both machines were 600 MHz Pentium IIIs.

seconds, but this is insignificant compared to the total execution time of 35 and 26 seconds, respectively.

The large peak for messages with 100 doubles in the Linux chart is likely due to a cache effect that manifests in the presence of 16 KB of L1 cache and full-speed L2 cache. The sender in this case was a Pentium III with 256 KB of full-speed L2 cache, while the receiver was a Pentium III with 512 KB of half-speed L2 cache. When the roles were reversed, the peak was not as pronounced. Furthermore, when we used a Pentium 4, there was no peak at all. The Pentium 4 has 256 KB of full-speed L2 cache, but only 8 KB of L1 cache.

For Solaris, the peak is not present at all. This is due to the cache effect being masked by other factors. Analysis with the Sun Forte 6 tools on a Blade 1000 showed that the function that consumed the most CPU cycles for message sizes from 10 to 1000 (a Solaris internal function named `multiply_base_2_vector()`) was actually incurring no cache misses.

5.2 Deserialization

Some of our performance enhancements affected deserialization only. Since our deserialization was no slower than our serialization, the deserialization was isolated to reveal the effects. A dummy sender repeatedly transmitted a pre-composed SOAP message to the SOAP receiver.

5.2.1 Trie

Processing of SOAP messages involves repeated matching of XML tags. Figures 5 and 6 show that using the trie structure instead of the STL map class can enhance performance

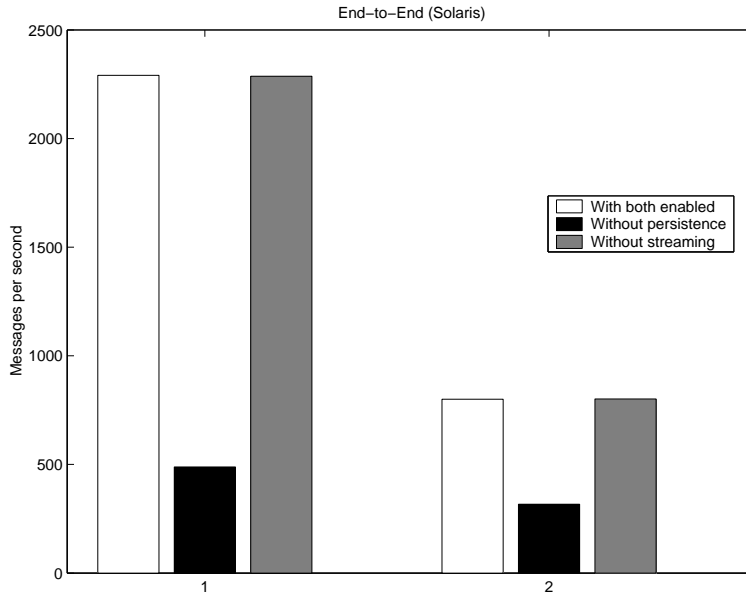


Figure 4: Effect of persistent connections and streaming for (1) a string and (2) an object with 64 integer elements, tested Solaris machines. The sender and receiver were both Sun Blade 100 machines. Each machine had a 500 MHz UltraSPARC-IIe.

when the number of tags is significant. The STL `map` is implemented as a balanced binary tree, for which lookups are $O(\lg n)$, compared to the trie for which lookups are $O(1)$. The trie also has a lower constant because the matching is done as the parser scans the tag. The binary tree, on the other hand, needs to make repeated comparisons of the tag against keys stored at the nodes.

5.2.2 Array Parsing

We use knowledge of the SOAP message schema to enable an array-specific parser when parsing SOAP arrays. Figures 7 and 8 show that the array parser significantly improves performance.

5.3 gSOAP

To corroborate our results, we also compared our performance against gSOAP. The gSOAP [14] system provides a language binding for deploying SOAP in applications using C/C++ code. Since the gSOAP system only supports request-response messaging, we estimated the double/second rate by sending one array of 100,000 doubles. The test was between two Sun Blade 100 machines with 500 Mhz CPUs. The one-way network latency for an established TCP/IP connection was 0.2 milliseconds. For this configuration, gSOAP achieved 17,000 doubles/second.

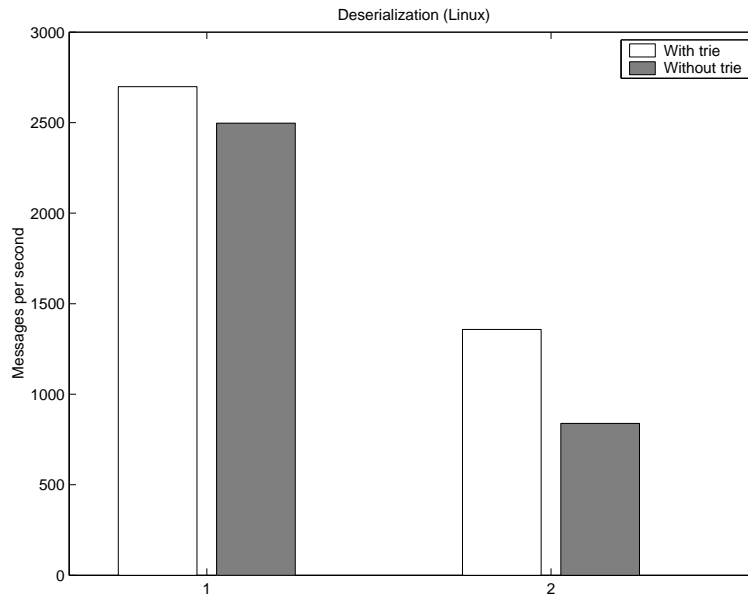


Figure 5: Effect of trie data structure on (1) a string and (2) an object with 64 integer elements, tested on Linux machines. As expected, the trie has little effect when the number of elements is very small. Both machines were 600 MHz Pentium IIIs.

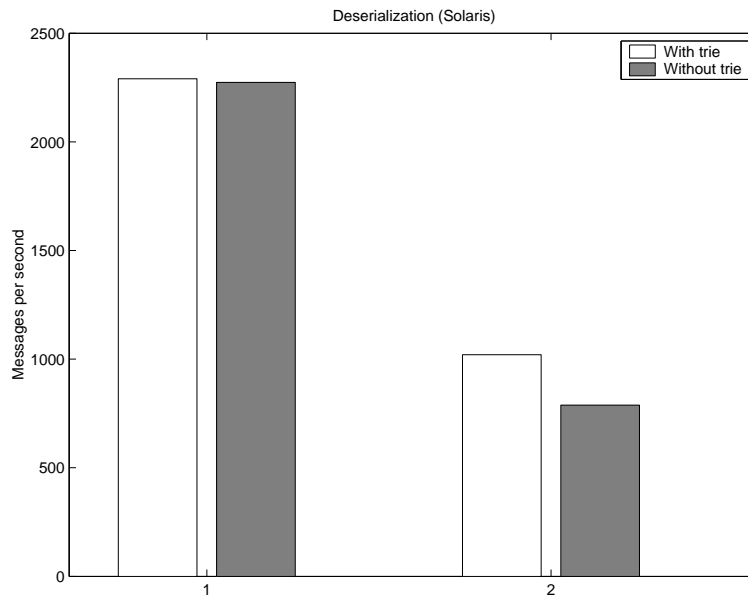


Figure 6: Effect of trie data structure on (1) a string and (2) an object with 100 integers, tested on Solaris machines. As expected, the trie has little effect when the number of elements is very small. The sender and receiver were both Sun Blade 100 machines. Each machine had a 500 MHz UltraSPARC-IIe.

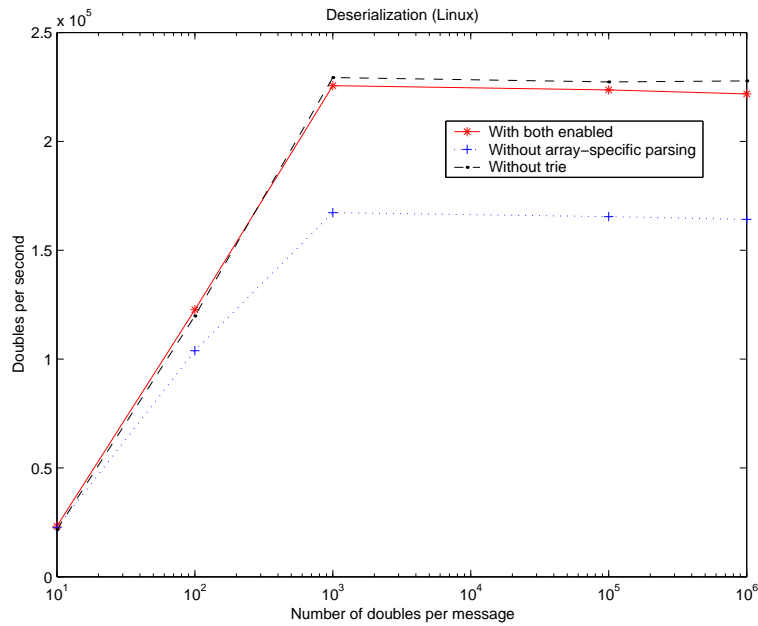


Figure 7: Effect of array-specific parsing and trie data structure on deserialization of array of doubles, tested Linux machines. Both sender and receiver were 600 MHz Pentium IIIs.

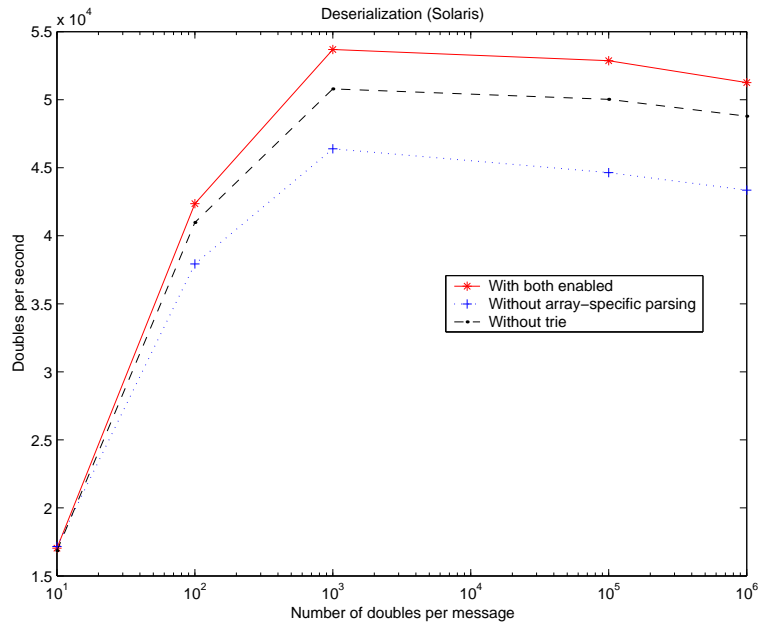


Figure 8: Effect of array-specific parsing and trie data structure on deserialization of array of doubles, tested on a Solaris machines. The sender and receiver were both Sun Blade 100 machines. Each machine had a 500 MHz UltraSPARC-IIe.

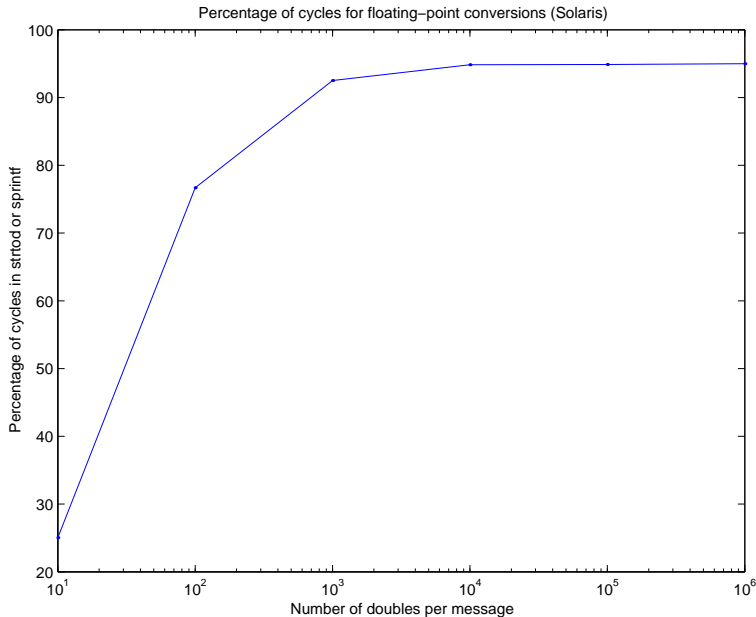


Figure 9: Percentage of CPU cycles spent performing double-to-ASCII or ASCII-to-double conversions as a function of message size. The test was conducted on a Sun Blade 1000 with 2x750 MHz UltraSPARC-IIIs.

6 Discussion

For a high-performance SOAP implementation, the two most costly operations are the conversion of ASCII to double and vice versa. In Figure 9 we have plotted the fraction of the total CPU cycles that is spent in either the `sprintf()` or the `strtod()` function as a function of the message size. This graph shows any further improvements to SOAP message processing will have little effect on the efficiency of sending arrays of doubles, unless the conversion of doubles is specifically addressed. In other words, with a high-performance SOAP implementation, an upper bound on the performance of arrays of doubles can be obtained simply by measuring the performance of `sprintf()` and `strtod()`.

If the full 18 digits of double precision are not required, some performance enhancement, especially on the Solaris platform, can be obtained by reducing the digits of precision. Figure 10 and 11 show the performance of the respective conversions as a function of the number of digits of precision. The Solaris platform exhibits sharp drops in performance between 14 and 17 digits. Profiling showed that somewhere in this range the library calls switch to costly high-precision integer operations.

Note that the IRIX platform performs remarkably well. By any measure, the IRIX machine is significantly slower than the other two, yet it performs almost as well as the 600 MHz Pentium III.

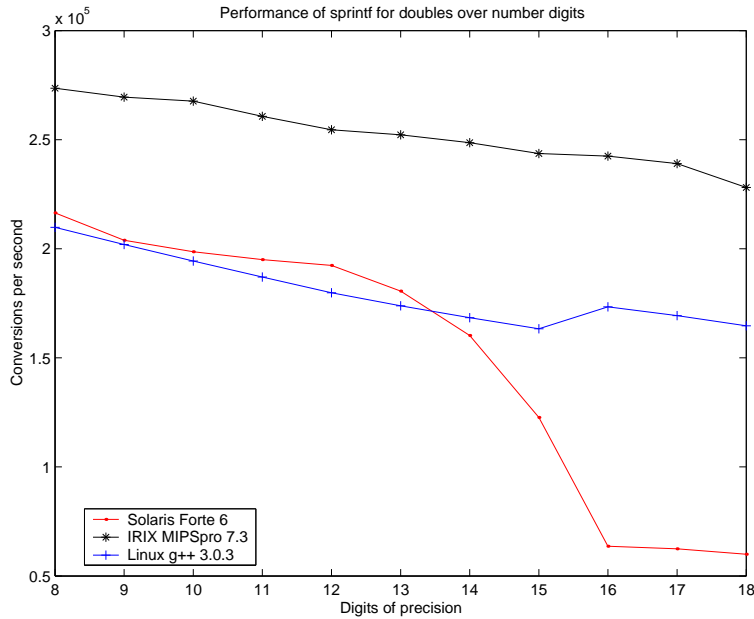


Figure 10: Performance of double to ASCII conversion over digits of precision. The Solaris machine was a Sun Blade 100 with a 500 Mhz UltraSPARC-IIe. The IRIX machine was an SGI Octane with a 195 Mhz R10000. The Linux machine was a 600 MHz Pentium III (Katmai).

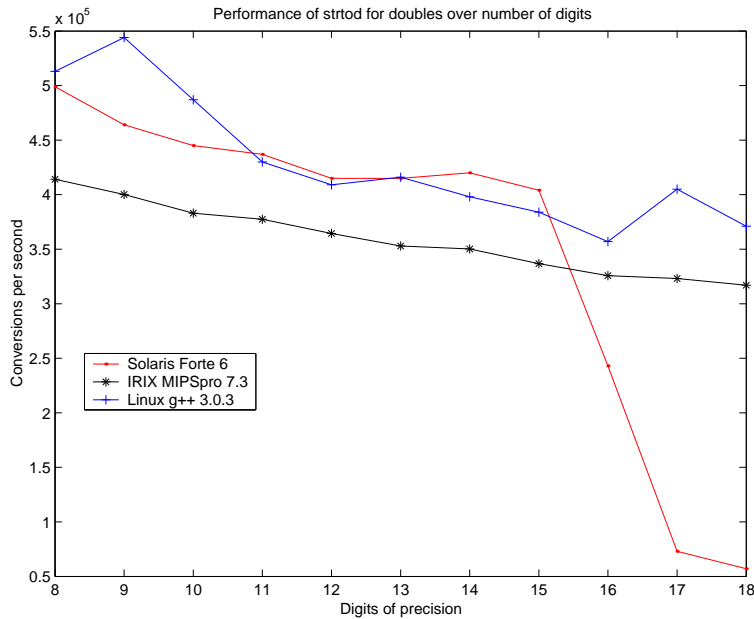


Figure 11: Performance of ASCII to double conversion over digits of precision. The Solaris machine was a Sun Blade 100 with a 500 Mhz UltraSPARC-IIe. The IRIX machine was an SGI Octane with a 195 Mhz R10000. The Linux machine was a 600 MHz Pentium III (Katmai).

6.1 gSOAP

In the experiment, gSOAP spent approximately 90% of the total time in the `sprintf()` function. We further observed that gSOAP calls `sprintf()` function twice for each serialization cycle. This is because it calculates the buffer length in the first iteration and then in the second iteration it sends the data. From our tests we have learned that it is better for gSOAP to switch to a one step process so that `sprintf()` is called only once. This will result in much greater gain than what streaming can provide with a two-step process.

7 Conclusion

We divided the process of sending and receiving a SOAP call into various stages and discussed an efficient way to handle each one. We introduced an XML parser that is specialized for SOAP and lends itself to high performance deserialization routines. The various techniques that can be used to improve the different stages of the call along with inherent bottlenecks were presented and analyzed their performance. We discussed the different parameters that govern the efficiency of converting routines between doubles and strings and stated the upper limit of performance that can be achieved with SOAP.

With these enhancements, over 90% of the CPU cycles were spent performing double-to-ASCII conversions for messages of 1000 doubles or more. Further improvements will come from improving the numerical algorithms for these conversions, rather than from improving the parsing of the SOAP message per se.

8 Further Work

Our results show that on the IRIX platform the conversion from binary to ASCII is very efficient. This suggests that the algorithms used by Solaris and Linux are not optimal.

Schema-specific parsers may prove to be beneficial for more than just large arrays. Generating C code or even machine code from a schema may greatly enhance performance.

We are also interested in studying the different ways in which the concepts we learned in this project can be applied to Java based systems.

References

- [1] Brian Tierney et al. White paper: A grid monitoring service architecture (draft), visited 03-10-01. <http://www-didc.lbl.gov/GridPerf/papers/GMA.pdf>.
- [2] World Wide Web Consortium. XML-Schemas, visited 2-17-00. <http://www.w3.org:80/TR/NOTE-xml-schema-req>.
- [3] World Wide Web consortium. Document object model, visited 7-15-99. <http://www.w3c.org/DOM>.
- [4] World Wide Web Consortium. XML, visited 7-20-99. <http://www.xml.org>.

- [5] Erik Christensen, Francisco Curbera, Greg Meredith and Sanjiva Weerawarana. Web Services Definition Language, visited 03-01-02. <http://www.w3.org/TR/2001/NOTE-wsdl-20010315>.
- [6] D. Box et al. Simple Object Access Protocol. Technical report, IETF, 1999. <http://www.ietf.org/internet-drafts/draft-box-http-soap-01.txt>.
- [7] D. Box et al. Simple Object Access Protocol 1.1. Technical report, W3C, 2000. <http://www.w3.org/TR/2000/NOTE-SOAP-20000508/>.
- [8] D. Megginson et al. SAX 2.0: The Simple API for XML, visited 07-01-00. www.megginson.com/SAX/.
- [9] Grid Forum. Grid performance working group, visited 03-01-01. [http://www-didc.lbl.gov/GridPerf/](http://www.didc.lbl.gov/GridPerf/).
- [10] Ian Foster and Carl Kesselman. *The GRID: Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann, 1998.
- [11] Madhusudhan Govindaraju, Aleksander Slominski, Venkatesh Choppella, Randall Bramley, Dennis Gannon. Requirements for and Evaluation of RMI Protocols for Scientific Computing. In *Proceedings of SuperComputing 2000, Dallas TX, 2000*, November 2000.
- [12] Network Working Group . Hypertext Transfer Protocol 1.0 , visited 03-04-02. <http://www.ietf.org/rfc/rfc1941.txt>.
- [13] Network Working Group . Hypertext Transfer Protocol 1.1 , visited 03-04-02. <http://www.ietf.org/rfc/rfc2616.txt>.
- [14] Robert A. van Engelen and Kyle A. Gallivan. The gSOAP Toolkit for Web Services and Peer-To-Peer Computing Networks. In *Proceedings of IEEE CC Grid Conference, 2002*.
- [15] Aleksander Slominski. XML Pull Parser, visited 02-15-01. <http://www.extreme.indiana.edu/soap>.
- [16] Aleksander Slominski, Madhusudhan Govindaraju, Dennis Gannon, and Randall Bramley. SoapRMI C++/Java 1.1: Design and Implementation. Technical Report TR-548, Indiana University, May 2001.
- [17] T. Faber, J. Touch, W. Yue. The TIME-WAIT State in TCP and Its Effect on Busy Servers,. In *Proceedings of IEEE INFOCOM*, March 1999.
- [18] The Apache Software Foundation. GSI SOAP, visited 03-01-02. <http://www-itg.lbl.gov/Grid/projects/soap/>.