

Web Services Performance: A Survey of Issues and Solutions

Kenneth Chiu

Indiana University
chiuk@cs.indiana.edu

Abstract

With its goal of wide support for loosely-coupled interactions, Web services consciously deemphasize performance in favor of qualities such as readability and portability. While not enough by themselves, these qualities encourage the cooperation and collaboration necessary to establish wide-spread adoption across many domains. Despite this ordering of priorities, however, performance cannot be simply ignored. Performance still matters, and awareness of the performance costs of design decisions will benefit both practitioners and researchers alike. This paper surveys the current issues and state-of-the-art regarding Web services performance. We detail bottlenecks for various use cases, and examine current and proposed solutions.

Keywords

XML, SOAP, Web service

1 Introduction

The success of the World Wide Web has led many researchers to study the principles upon which it is based[8]. The application of these principles to distributed computing has resulted in Web services, which have emerged in recent years as a new approach to distributed computing.

The potential of Web services 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. The SOAP¹[10] protocol was conceived expressly to support such interoperability in-the-large, thus making it a natural *lingua franca*. Designed using principles learned from HTTP[9] and HTML[13], it facilitates interdependent interactions between otherwise independent entities. SOAP is also the standard binding for the emerging Web Services Description Language[3]. SOAP's interoperability arises partly from its use of the Extensible Markup Language (XML)[2], which has gained 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 common one for SOAP. Since SOAP can combine the strengths of XML and HTTP, it is an attractive candidate for Web services.

A common trade-off in computing is between the needs of universality and performance, and SOAP does not escape this principle. Concomitant with its advantages then is a degree of

inefficiency that may limit the applicability of Web services to some situations. The qualities of SOAP that make it universally usable and consequently highly interoperable tend to work against high performance. In particular, XML specifies a primarily ASCII format.

This paper examines the issues of Web services performance, and surveys some of the research approaches being used to address the issues. We analyze the processing of SOAP messages, and identify the issues of each stage. We also survey some of the approaches being used to improve Web services performance.

2 Web Services

No standard definition of Web services exists, but they are generally understood to be loosely coupled and based on Web standards. For our purposes we will further assume that a Web service uses SOAP with XML documents described by the XML Schema[15][1] definition language.

2.1 SOAP

SOAP provides a mechanism for exchanging messages in a distributed environment. 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.

The features that makes SOAP so attractive also causes potential performance problem. Tagged data is sent in a textual format, which results in a performance penalty.

The process of converting data from a machine native form to the transfer form is known as *serialization*. The reverse process is known as *deserialization*.

3 Sending SOAP Messages

We assume that SOAP messages are encoded using XML typed with XML Schema. This requires that all self-described data be sent as ASCII strings. The description takes the form of start and end tags which often constitute half or more of the message's bytes.

3.1 Stages

The sending of a SOAP message can be divided into several stages. These stages may not have a one-to-one mapping to

¹SOAP originally was an acronym for Simple Object Access Protocol, but this meaning was removed in version 1.2.

the actual source code implementation, but provide context useful for discussion and analysis. We discuss separately the performance issues and possible solutions to each stage.

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

Since bottlenecks can occur in different places depending on the application and implementation, performance profiling must be used to determine what performance optimizations to investigate.

3.1.1 Stage 1

A SOAP message begins as some kind of data structure in a program. Stage 1 traverses this structure to impart a corresponding structure to the SOAP XML. Generally speaking, 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. This suggests that for applications which primarily pass strings transparently from a received SOAP message out to a sent SOAP message, the string should be stored internally also in UTF-8/16 so as not to require conversion to and from UTF-8/16 to the internal representation.

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.

In fact, converting ASCII to IEEE-754, and vice versa, is surprisingly expensive[5]. Unfortunately, there seems to be no magic bullet for reducing this cost. Applications that use floating-point numbers extensively may need to adopt some kind of multiprotocol, binary XML approach, as proposed in Section 5.1.

The results of [5] also show, however, that there is some variability in the efficiency of different platforms when performing these conversions. This suggests that some limited improvements can be made by optimizing implementations. Significant improvement will depend on breakthroughs in the numerical aspects of these conversions, however.

One modification would be to allow a user-specified "tolerance" for doubles and floats. Conversions which require fewer than the full precision of a double-precision floating-point number are significantly faster. A major reason for the large performance drop near full precision (17 digits) is

because the IEEE standard specifies rounding modes, causing the conversion functions to use multiprecision libraries. The disadvantage of allowing users to specify lower tolerances is that it departs from strict adherence to the IEEE 754 numerical standard.

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 named 'el' is serialized with

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

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 may 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, 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 back-patch 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 concatenates 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.

The high cost of double-to-ASCII conversions imply that they should be avoided if possible. In particular, performing many conversions in a first pass to determine the length of the HTTP message is probably not cost-beneficial. An alternative is to simply fix the size of the converted double to its maximum possible size, and fill with spaces if the actual conversion requires less space. Thus, the calculated HTTP length may be slightly longer than necessary (because the actual message has been padded with spaces), but this slight overhead is normally more than balanced out by the elimination of the extra double-to-ASCII conversion.

Creating a connection per message can be another performance bottleneck. 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[7].

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. The actual benefit of persistent connections of course depend on the message size. Because the cost of establishing a connection increases with latency, the benefit of persistent connections will 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.

3.2 Pipelining

Previous work has shown that the memory usage of SOAP can be prodigious. A typical SOAP message may be 4-10 times the size of its corresponding machine representation. This can be particularly significant for large arrays, which are common objects in scientific computing. Besides being careful not to create extra copies, we chose to address the memory concern by using HTTP 1.1 instead of HTTP 1.0.

HTTP 1.1 supports a form of streaming called chunked encoding. The body is sent in chunks, with each chunk preceded by its 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 pro-

gram will occur. Of course, calling send too many times may cause the overhead of system calls to dominate.

Streaming has larger impact on 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 short.

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.

When receiving a chunked encoding message, it is important to decouple the chunk sizes from the size specified in each read/receive system call. The purpose of chunked encoding is not to specify the chunks in which the incoming HTTP message should be read, but rather to support two properties simultaneously: (1) a receiver can determine the end of the message without any kind of special marker, and (2) the sender can start transmission before knowing the length of the message.

Thus, the chunk sizes will be appropriate for the sender, and may be either much too small (too many system calls) or much too large (too much paging) for the receiver to operate efficiently.

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[16] (DOM) and the Simple API for XML[12] (SAX). 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 com-

pletely 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[14], 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 is more natural for many applications, because the does not cross module boundaries multiple times in one call stack, as it does with callbacks.

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.

Processing SOAP messages involves repeated matching of XML tags. 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 suggest using *tries*, which unambiguously determine whether or not a key is valid. A trie is essentially a table-driven deterministic finite automaton 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.

The C++ Standard Template Library (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.

Schema-Specific Parsing. For a application to interpret the data in a SOAP message, it must have some idea of its structure. This structure may be known in an ad hoc manner, or it may be formalized 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 (open coding). On some architectures, this can greatly improve performance, because instructions can be more effectively prefetched than data.

One can even imagine a parser-generator that directly generates machine-code (or Java byte-code) from a schema.

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 simple 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. We examine this issue more in Section 4.2

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 Pipelined Push-Pull

Pipelining can improve performance for two reasons. The first is that it can greatly reduce memory usage for large messages. A non-pipelined implementation will hold the unprocessed SOAP message and the final form of the data in memory at the same time. A pipelined implementation, on the other hand, will process the SOAP message in small chunks, thus greatly reducing the memory usage.

The other benefit of pipelining is that it improves memory locality, which improves cache utilization. As data flows through a pipeline, it is accessed by each successive stage in the pipeline while it is still in cache.

Another important precept is to 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 suggest interfacing 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.

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.

5 Other Techniques

In the previous sections we examined some approaches to improving performance on a stage-by-stage basis. We now examine approaches that span stages.

5.1 “Binary” XML

XML is a transfer syntax for the information. Implicit in the specification, however, is an abstract model for structuring data and metadata. Web services researchers soon realized that this abstract model was in itself useful, and developed the XML Infoset specification[6].

XML infosets suggest another approach to Web services performance. Once we abstract XML, we can then develop different concrete transfer syntaxes to realize that abstraction.

This is the idea behind what has sometimes been termed “binary” XML. The application is written to an API for XML that abstracts the actual syntax. This means that application no longer depends on how the XML is really represented, and can use different, perhaps more efficient representations.

Because of the high cost of converting IEEE-754 floating-point numbers to ASCII, a binary XML should represent the number in some kind of machine format. This implies, however, that the Infoset API must also present the application with typed data already converted to native form (such as `double` in C++), in order to assure that no ASCII to IEEE-754 conversions take place anywhere along the path from the original source of the data to the final destination.

An early “binary” XML effort is the WBXML[11] specification. Unfortunately, this format did not transmit doubles in IEEE-754, so would not improve applications that made heavy use of floating-point numbers.

5.1.1 Multiprotocol

To preserve interoperability, a binary XML approach should also be able to use regular XML. This suggests a multiprotocol approach[4], where less efficient but more interoperable formats are used to establish initial communications. A more efficient format is then used when supported by both ends.

5.2 Compression

XML is generally thought of as a verbose format for data. A double-precision floating point number that takes 8 bytes natively may easily take 25 bytes in XML. Consequently, some people have suggested using compression to reduce the size of the XML representation.

Whether or not compression is useful depends strongly on the environment. If available bandwidth is very low compared to the available processing power, then the bottleneck is the network, and compression can be helpful.

In LAN environments, however, we have observed that Web services communications are usually CPU-bound. That is, they do not saturate the available network bandwidth. In these situations, compression would only worsen the performance.

5.3 Extensions

Two basic approaches can then be taken to resolve the tension between SOAP’s universal lowest common denominator capabilities, and the need for high performance communications: (1) extend the SOAP protocol to provide for binary representation of floats, or (2) use SOAP as an initial mechanism that can then negotiate other (faster) shared protocols.

SOAP extensions might do something like encode IEEE-754 as base64. Though this can be useful in some situations, the SOAP is no longer conforms to standard encodings. We believe interoperability is preserved better, in the long run, if instead “vanilla” SOAP is used to negotiate faster shared protocols. Note that this faster shared protocol might actually be a SOAP extension.

Our recommendations are to follow the second approach, unless and until the community can persuade the W3C organization to extend SOAP to include binary protocols. In the event that two applications do not have a shared faster protocol, SOAP over HTTP can still be used as a fail-safe tool that assures communications can succeed.

6 Conclusion

We have divided the process of sending and receiving a SOAP call into various stages, and analyzed efficient ways to handle each phase. Various approaches that can be used to improve the different stages of the call along with inherent bottlenecks were presented and discussed.

For non-demanding applications, standard SOAP can provide adequate performance. Schema-specific parsing can provide additional performance boosts. For applications that use large numbers of floating-point numbers, however, we believe that a binary XML, multiprotocol approach will ultimately be necessary.

7 References

- [1] Paul V. Biron and Ashok Malhotra. *XML Schema Part 2: Datatypes*. May 2001.
- [2] Tim Bray, et al. *Extensible Markup Language (XML) 1.0 (Second Edition)*. October 2000. <<http://www.w3.org/TR/REC-xml>>.
- [3] Roberto Chinnici, et al. *Web Services Description Language (WSDL) Version 1.2*. March 2003. <<http://www.w3.org/TR/wsdl12>>.
- [4] Kenneth Chiu, Madhusudhan Govindaraju, and Dennis Gannon. The Proteus Multiprotocol Library. In *Proceedings of the 2002 Conference on Supercomputing*, November 2002.
- [5] Kenneth Chiu, Madhusudhan Govindaraju, and Randall Bramley. Investigating the Limits of SOAP Per-

formance for Scientific Computing. In *Proceedings of the Eleventh IEEE International Symposium on High Performance Distributed Computing (HPDC'02)*, July 2002.

- [6] John Cowan and Richard Tobin. *XML Information Set*. October 2001. <<http://www.w3.org/TR/xml-info>>.
- [7] 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.
- [8] Roy Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. Ph.D. Thesis. University of California, Irvine. 2000.
- [9] Roy Fielding, et al. *RFC 2616: Hypertext Transfer Protocol 1.1*. June 1999.
- [10] Marting Gudgin, et. al. *SOAP Version 1.2*. December 2002. <<http://www.w3.org/2000/xp/Group>>.
- [11] Bruce Martin and Bashar Jano. *WAP Binary XML Content Format*. June 1999. <<http://www.w3.org/TR/wbxml>>.
- [12] D. Megginson et al. SAX 2.0: The Simple API for XML, visited 07-01-00. <www.megginson.com/SAX/>.
- [13] Dave Raggett, Arnaud Le Hors, and Ian Jacobs. *HTML 4.01 Specification*. December 1999.
- [14] Aleksander Slominski. XML Pull Parser, visited 04-15-02. <<http://www.extreme.indiana.edu/xgws>>.
- [15] Henry S. Thompson, et al. *XML Schema Part 1: Structures*. May 2001.
- [16] World Wide Web consortium. Document Object Model, visited 7-15-99. <<http://www.w3c.org/DOM>>.