Real-Time Compression of SOAP Messages in a SOA Environment

Júlio Cezar Estrella
Universidade de São Paulo
Cx.Po. 668, 13560-970,
São Carlos, SP, Brazil
+55 16 3373 9655
jcezar@icmc.usp.br

Marcos José Santana
Universidade de São Paulo
Cx.Po. 668, 13560-970,
São Carlos, SP, Brazil
+55 16 3373 9655
mjs@icmc.usp.br

Regina H. C. Santana
Universidade de São Paulo
Cx.Po. 668, 13560-970,
São Carlos, SP, Brazil
+55 16 3373 9655
rcs@icmc.usp.br

Francisco José Monaco
Universidade de São Paulo
Cx.Po. 668, 13560-970,
São Carlos, SP, Brazil
+55 16 3373 9655
monaco@icmc.usp.br

ABSTRACT
This paper discusses how the use of compression techniques aimed at decreasing data transfer times over a communication network can influence the response time of an application that process SOAP messages in the context of a service-oriented architecture. Following an overview of the most known object models and comparing some of their features, the article presents an heuristic that can be used to decide whether a soap message should or should not be compressed. A simulated experiment shows that the proposed heuristic can help in reducing the service response time in a variety of scenarios.

Categories and Subject Descriptors
H.4 [Information Systems and Applications]: Communications Applications

General Terms
Algorithms, Performance.

Keywords
Web Services, SOAP, Compression, Real-Time communication.

1. INTRODUCTION

Real-Time compression of SOAP message is an active research area that in SOA field, mainly because large SOAP messages can degrade the response time of an application. A long response time may be experienced if the request is made to a heavily loaded web service, or if the service implemented in a web server consumes plenty of CPU time. In order to meet exigent quality of service requirements [7] [13], service providers should try to minimized the overhead associated to the processing of XML documents. In this extent, performance, often measured as throughput or response time, is the rate a service processes requests or the time taken between the invocation of a request and reception of a response respectively [22]. Several research works have been investigating how techniques can be used in order to minimize the overheads caused by XML processor [4], e.g. by optimizing the role of parameters, especially large arrays used in scientific computing [16]. A survey on the state of the art of binary encoding strategies for SOAP and a novel experimental concept for SOAP compression can be found in [21], while a discussion on how the performance bottleneck of SOAP messages can be eliminated by XML compression is given in [9]. Service performance can be degraded when extracting envelopes from SOAP messages, since parsing XML data is time-intensive. Another factor influencing the response time as seen by the client is the communication delay. This may account for a substantial impact on the service throughput specially when the service output is a large file that should be transferred over a low-bandwidth link. One important consideration is that SOAP over HTTP uses significantly more bandwidth than competitive technologies like Remote Procedure Calls (RPC). Not only SOAP but also HTTP itself adds significant overhead to the payload -- hence the interest in the development of efficient transport mechanisms for web services.

Experimental evaluation of SOAP performance using realistic business application message contents is carried out in [19]. Results can be compared with widely used existing protocols in order to get some indication on whether SOAP is appropriate for high performance applications [12].

This paper introduces an heuristic for deciding whether data compression can improve response time of web services in function of data content type and algorithm characteristics, current network performance and computational load in the end hosts. The approach is based on the estimation of the trade-off between the transfer time decrease in virtue of data volume reduction, on one hand, and the compression-decompression time overhead, on the other. Results show that not always compressing SOAP messages can eliminate performance bottleneck of the service throughput. The response time seen by the client can be even degraded when using XML compression in some cases. In other cases, compression-decompression techniques can sensibly improve the service performance.
2. WEB SERVICES AND SOAP

A service-oriented architecture (SOA) can be understood as a computer systems architectural style for creating and using business processes, packaged as services, throughout their life cycle. SOA resources are meant to allow for different applications to exchange data between distinct systems independently of programming language [14], making possible to separate functions into distinct units (services) [5] which can be distributed over a network, combined and reused to create business applications [18]. These services communicate with each other by passing data from one service to another, or by coordinating an activity between two or more services. A way of implementing a service-oriented architecture is by means of Web Services standards.

The W3C \(^1\) defines web services as an application identified by a URI \(^2\), whose interfaces and linkings are defined, described and discovered using a standard language as XML \(^1\). The interactions between web services typically occurs through SOAP \(^3\), an XML-based protocol [15]. SOAP is presented as a backbone for a new generation of applications of distributed computing, independent of platform and programming language. Using its facilities, the descriptions of interfaces of web services are express using a language called WDSL \(^4\).

Since all SOAP interactions between client and service provider are represented in XML [10], it is important consider how the information based in this format is processed at both ends. A parser is a program that interprets the structure of an XML text document while a document representation is the set of data structures used by a program to work with the document in memory. The document model can be handled by an API or library that supports working with a document representation. Parser, document representation and document model are critical parts that need to be optimized when processing XML documents.

To work with XML documents in memory the developer can choose among the various representation patterns. Out of these, DOM, JDOM, Xerces, and more recently the Axiom (Axis2 Object Model) [6] are well-known examples. The choice between them should to take into account ease of use and performance. DOM is the official W3C standard for XML documents representation. As its main features, DOM makes heavy use of interfaces and inheritance for different components of an XML document. The advantage is that developers can use a common interface to work with different types of components, while the disadvantage is the complexity involved. DOM is neutral regarding the programming language and makes no use of common components such as Java Collections Classes.

JDOM is object model based on Java language, allowing interaction with XML faster and more simpler than DOM based implementation. There are two differences between DOM and JDOM. The first is that JDOM uses classes instead of interfaces. This simplifies API in some aspects, although limiting flexibility. The second is that the JDOM API makes extensive use of Collections Class, simplifying the its use by Java developers who already have experience with these classes. JDOM itself does not include a parser. Usually SAX2 parser is employed to process and validate the XML document. This includes converting a JDOM representation to an output, with a SAX2 event, a DOM model, or as a text document in XML.

DOM4j is an open source library for working with XML on the Java platform using Java Collections Framework with full support for DOM, SAX and JAXP [2]. It incorporates a number of features beyond the basic processing of XML documents, such as support for XPath, XML Schema and processing based on events. DOM4j is based on an interface and an abstract base class, making extensive use of Class Collection. In many cases, it provides alternatives to allow improved performance as well as a more direct approach to achieve the codification. Flexibility is the advantage with regard to JDOM. However, the API is a little more complex, due to the features not presented in JDOM.

Axiom (Axis2 Object Model) is an information model based on XML, initially developed to Apache Axis2 (a framework for building Web Services). The object model is based on other models as DOM and JDOM. The difference between DOM/JDOM and Axis Object Model (OM) is the internal behavior. This is different and unique, and is based on StAX API (Streaming API for XML). Both JDOM/DOM are XML processing models based on document trees. With DOM, XML information is all stored in memory. Axiom has been engineered to be less memory-intensive, using deferred building [6].

The current approaches for XML processing can be categorized into two broad branches. The tree-based approach is followed by DOM. The whole XML file is loaded into memory, causing the size of the object model to be larger than the source XML. Therefore, this method is not appropriate for environments as J2ME or systems handling large documents [6]. In the alternative event-based approach, the XML file is processed in chunks and it is not necessary to build complex memory structures. The disadvantage is the control made by the developer. It is not necessarily easy to navigate forward and backward using an event-based approach. That is the reason why tree-based approach is preferred over event-based approach amongst developers, although it is memory inefficient [6].

One point in favor of Axiom is that it tries to get the best from both tree-based and an event-based approaches. Axiom depends on the StAX in order to input and output data. Axiom has deferred building support, which means that this model does not build the document until it is absolutely required by the application.

Another point is that in document model such as DOM/IDOMX, the control of the processing is done by the parser (SAX, for example). With StAX, the control of what must be processed is done by the client. The parser is controlled by the client deciding what type of information should be accessed in a XML document tree. All libraries based on pull parser are small in size and facilitates the filtering of elements since clients know that will occur a particular event.

\(^1\) World Wide Web Consortium.
\(^2\) Uniform Resource Identifier.
\(^3\) Extensible Markup Language.
\(^5\) Web Service Description Language.
2.1. The Axiom Object Architecture

The Axis Object Model access the XML stream through the STAX interface wrapped by the builder interface. Basically, a builder reads information from a XML parser and build the object model. The current implementation of Axiom has three builders: STAXOMBuilder (build an Axiom Tree) STAXSOAPModelBuilder (can read SOAP messagens and build a SOAP objetc model basead on Axiom) and MTOMSTAXSOAPModelBuilder (can read MTOM menssages). Axiom has also the option for STAX events with or without building the memory model for later access. This concept is named caching. With caching, the information can be accessed with or without the construction of an object model. The caching concept allows appropriate and controlled use of memory. For example, send messages SOAP does not require the construction of object model in memory. Yet, the Axiom API can be implemented using either a linked-list model, a tree-based model or a DOM-based model.

While the selection between different in-memory document representation patterns has influence in the system responsiveness, another factor impacting the service performance is the transferring of the generated XML reply through the communication network. It is easily comprehensible that data compression can be used to shorten the transmission time of a large file over a low-bandwidth network. Likewise, it is clear that the processing overhead can overcome this transmission time reduction if, instead, a small file is to be transferred over a high-bandwidth network. What happens between those extremes, on the other hand, may not be so obvious. The following sections address this issue.

3. The Compression Trade-Off

The theoretical time to send a volume \( V \) of data through a network link with constant throughput \( \phi \) and latency \( \lambda \), hereing referred to as transmission time, is given by Eq. (1).

\[
T_M = \frac{V}{\phi} + \lambda
\]

If, instead, \( V \) is reduced into a compressed volume \( V' \) prior to being sent through the network, the transmission time decreases by \( (V - V')/\phi \), therefore meaning a time gain proprotional to the volume diminution. The on-the-fly compressiodecompression process, nonetheless, adds an extra time to the total transfer time as given by Eq. (2)

\[
T_T = T_M + T_{CD}
\]

where \( T_{CD} \) is the time that it takes to compress the file prior to transmission plus the time it takes to decompress it after reception.

With Eq. (1) into Eq. (2) there comes the expression for the total transfer time in function of the data volume, network throughput and latency, and the compression-decompression time as stated in Eq. (3).

\[
T_{transf} = \frac{V'}{\phi} + \lambda + T_{CD}
\]

It is clear that \( V' \) depends on both the original data volume \( V \) and the compression algorithm employed \( \alpha \). It should not be disregarded, although, that another variable influencing the compression result is the data content type. For instance, some algorithms can be proven to produce different \( V/V' \) ratio for either text or image files due to the particular data patterns present in each kind of content data and how the way the given algorithms take advantage of them. Entropy-based compression algorithms, for example, explore redundancy in the information representation. The run-length technique [17], for example, works by replacing long sequences of repeated symbols (bytes or other data units) with a pair of data denoting the number of repetition and the repeated symbol. This scheme is well-suited for bitmap image files generated by paint-like software applications, which are characterized by regions of uniform solid colors. For landscape photo images produced by digital cameras, on the other hand, uniformly-colored adjacent pixels are not so common, resulting that run-length compression can be much less effective, if not useless at all. In this case, statistical compression techniques, such as Huffman coding [11], may be preferable. These work by replacing the usual uniformly-sized symbols (e.g. 8-bit bytes) with varying-sized alternate symbols, reserving the shortest ones for frequently appearing data unities. The LZW compression algorithm [20] takes advantage of both former principles to produce a dynamically-constructed translation table that assign indexes to repeated mixed symbols sequences, and has been used for text and 256-color image compression. Another class of algorithms diverse from the former entropy-based one is that founded on source-encoding. Differential-encoding is well-suited for general audio compression and it takes the advantage of the fact that the sequence of data points produced by periodic sampling is somehow temporally related. In voice compression applications, the sequence of sample values can be replaced with the relative difference between consecutive samples because it takes less bits to represent the (positive or negative) variation between successive data values than to represent the absolute value of the data points itself. Other source-encoding compression algorithms include frequency spectral analysis (e.g. Fourier) and vectorial quantization employed in multimedia contents. The use of wrong combination of algorithms and data content type can be ineffective or even cause volume inflation.

That said, if, by a judicious hypothesis, which is indeed valid for an important class of algorithms, compressed and original volumes are constantly proportional, than \( V \) and \( V' \) can be related by a compression factor \( F \) as in Eq. (4),

\[
V' = F^{\alpha \tau}
\]

where \( F \) is the compression factor of the algorithm \( \alpha \) applied to a data set of type \( \tau \).
As for $T_{cd}$, this also depends on the original data volume, the compression algorithm and the data content type. Diversely from the compression factor, however, the compression-decompression time depends as well on the processing power and the current workload present in the host. Even worse, compression and decompression take place in different hosts: the former in the sender, and the latter in the receiver. It is convenient, thus, to note it explicitly making $T_c$ the time it takes for the sender to compress the volume of data, and $T_d$ the time it takes for the receiver to decompress the compressed volume, both for an algorithm $\alpha$ and data content type $\tau$ as summarized in Eq. (5).

$$T_{cd\alpha\tau} = T_{C\alpha\tau} + T_{D\alpha\tau}$$  \hspace{1cm} (5)

Actually, both compression and decompression are also time-dependent, since they are affected by the instantaneous processing capacity at the end-hosts; in these equations, $T_c$ and $T_d$ stand for the processing times at the moment the transfer takes place and is considered constant in that period. Anyway, with the same approach adopted so far, it is possible to formulate the second hypothesis for $T_c$ and $T_d$, considering compression and decompression times constantly proportional to the data volume that is being processed -- the experimental data provided ahead show that this is valid for the essayed algorithm. This makes it possible to write $T_c$ and $T_d$ as in Eq. (6)

$$T_{C\alpha\tau} = \frac{V}{K_{C\alpha\tau}}$$ \hspace{1cm} (6)

$$T_{D\alpha\tau} = \frac{V}{K_{D\alpha\tau}}$$

where $K_{C\alpha\tau}$ is the compression rate at the sender host and $K_{C\alpha\tau}$ is the compression rate at the receiver host, both for an algorithm $\alpha$ and data content type $\tau$.

Now with Eq. (4) and Eq. (6) it is possible to rewrite Eq. (3) as Eq. (7) below$^6$.

$$T_{trasf} = \frac{F}{\phi} + \lambda + \left( \frac{V}{K_{C\alpha\tau}} + \frac{V}{K_{D\alpha\tau}} \right)$$  \hspace{1cm} (7)

If one wants to know when the transfer for compressed data is shorter than that for the raw data, it suffices to calculate the difference between the transfer time $T_1$ with compression and $T_2$ without compression, as in Eq. (8).

$$T_1 - T_2 = \left[ \frac{F \cdot V}{\phi} + V \cdot \frac{V}{K_{C\alpha\tau}} + \frac{V}{K_{D\alpha\tau}} \right] - \left[ \frac{V}{\phi} + \lambda \right]$$  \hspace{1cm} (9)\hspace{1cm} which, simplified, turns into Eq. (9):

$$\Delta T = V \left[ \frac{1 - F}{\phi} \cdot \frac{K_{C\alpha\tau} + K_{D\alpha\tau}}{K_{C\alpha\tau} \cdot K_{D\alpha\tau}} \right]$$ \hspace{1cm} (9)

The result of the above equation informs whether the transfer time is either increased or decreased by using compression-decompression in the end hosts. If $\Delta T$ is positive, than the tread of between transmission time decrease versus compression time increase bends to the side of compression-decompression, meaning that transfer time is shortened by applying this technique. On the contrary, if $\Delta T$ is negative, it means that the compression-decompression time is longer than the amount of time corresponding to that saved during the transmission. A value of $\Delta T = 0$ simply means no gain, no loss.

4. Experimental Essays

In order to assess the practical value of the elicited hypotheses, some essays were carried out. There follows a summary of the experiment and its results.

A widely known and extensively used compression algorithm is that implemented by the gzip software application, usually available as a system utility in Unix-like operating systems. It is no named after the GNU Project, to which it was originally intended for use, and possibly the popular proprietary PKZIP commercial application for which it offers an open source alternative. Gzip [8], [3] is based on the DEFLATE [1] algorithm, a combination of LZ77 and Huffman coding intended as replacement for the patented LZW compression technique.

Since the compression factor is influenced also by the data content type, it is convenient to specify which kind of symbol patterns are present in the volume of data to be transferred. Since SOAP conversation among Web services is based on XML format, text-files were selected for the experiment. XML text files are not expected to contain either long sequences of repeated symbols (such as in "paint" images) or smooth transitions between adjacent (multi)byte symbols (such as in synthetic audio). Instead, text-files comprise short heterogeneous character strings separated by blank spaces, usually forming varying-size phrases.

In order to prepare the experiment input data a pseudotext generator software was coded in C language. The small program, genfile, produces text-like files of exactly specified byte lengths. Generated pseudo-texts bear the referred text-files characteristics, except by some minor differences, such as the fact that in human-readable texts the whole alphabet is not equally frequent -- capital letters are much rarer, lowercase letters have occurrence patterns -- and that in XML files < and > signs are expected to

---

$^6$ Here the compression rate (in bps) is given in function of the decompressed volume.
appear widely. These, however, are unlikely to have influence on the end result for this particular experiment.

As the first step, genfile was used to produce pseudo-text files of sizes 1KB, 10KB, 100KB, 1MB, 10MB, 25MB, 50MB, 75MB and 100MB. These files were then compressed and subsequently decompressed with gzip. The compression and decompression times were measured with the aid of the time software utility often available in GNU/Linux distributions, which informs the total (wall clock) time of any program and the run time (only the time the program has really processed). The latter value was noted.

The calculated compression factor $F$ and decompression rates $K_C$, $K_D$ were noted out on the Table 1. These values were obtained by running experiment on a dual-core 1.66MHz x86 computer platform with 1GB of RAM, on top of a 2.6.24 Linux kernel, with low CPU and memory loads. As stated, $\alpha$ and $\tau$ are for gzip algorithm and pseudo-text files.

<table>
<thead>
<tr>
<th>$V$ (MB)</th>
<th>$F$</th>
<th>$K_C$ (Mbps)</th>
<th>$K_D$ (Mbps)</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>0.64</td>
<td>6.82</td>
<td>33.78</td>
</tr>
<tr>
<td>50</td>
<td>0.64</td>
<td>6.96</td>
<td>31.81</td>
</tr>
<tr>
<td>75</td>
<td>0.64</td>
<td>6.77</td>
<td>34.40</td>
</tr>
<tr>
<td>100</td>
<td>0.64</td>
<td>6.85</td>
<td>35.10</td>
</tr>
</tbody>
</table>

As shown in the table, the hypotheses that $F$, $K_C$ and $K_D$ were constant are valid -- at least for the range between 24MB and 100MB (timings for smaller volumes seems to fall off the precision of the time utility).

5.A Decision Making Heuristic

With the values of $F$, $K_C$ and $K_D$ of Table 1 it is possible to fill in Eq. (9) to decide whether there is some benefit in using the compression-decompression technique to decrease transfer time. Since there is some computational overhead in starting the compression and decompression processes at the end hosts, too small positive values of $\Delta T$ can be overcome by varying latencies at both sides. Therefore, to cope with this, a threshold was used to trigger compression-decompression.

Table 2 brings the result of the heuristic for an empiric threshold of 5s (reasonably larger the expected overhead in pratical systems).

<table>
<thead>
<tr>
<th>$V$ (MB)</th>
<th>$\phi$ (Mbps)</th>
<th>$\Delta T$ (s)</th>
<th>Decision</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>1.0</td>
<td>3.6</td>
<td>don't compr</td>
</tr>
<tr>
<td>22</td>
<td>1.0</td>
<td>4.0</td>
<td>don't compr</td>
</tr>
<tr>
<td>26</td>
<td>1.0</td>
<td>4.4</td>
<td>don't compr</td>
</tr>
<tr>
<td>24</td>
<td>1.0</td>
<td>4.7</td>
<td>don't compr</td>
</tr>
<tr>
<td>28</td>
<td>1.0</td>
<td>5.1</td>
<td>compr</td>
</tr>
</tbody>
</table>

With the measured compression factor and rates and for a throughput of 1MBps, compression-decompression is worthy only for volumes above 25MB approximately. If the instantaneous throughput falls to 90% of that value, however, beyond 21MB compression-decompression is already interesting. The results are consistent with the rather intuitive notion that compression is most likely to be advantageous for transferring large files over low-bandwidth links.

6.A Conceptual Architecture

In order to illustrate how the heuristic can be used in a service-based ensemble, Figure 1 shows a brief outline of the architecture that is being developed at our research group.

![Figure 1: Conceptual architecture cal threshold of 5s](image)
7. Conclusions

This paper presents an heuristic for compressing SOAP messages in a SOA environment. Results show how data compressing techniques can help to minimize the response time perceived by the user in function of network throughput, file size, content-type, algorithm properties and client-server performance parameters. XML compressing is an area where many problems remain open. Although techniques are being discussed in the literature, no heuristic is encapsulated with a compression technique of a soap message engine such as Apache Axis2.

Although the essays reported in the article were carried out in a small-scale test bed, the overall conclusions apply for enterprise-level systems. Further refinement of the heuristics to include other environmental effects and real-world experiments for practical validation of the results needed to assess the method under different conditions are being investigated under the framework of this research project. Meanwhile practitioners are referred to the simplified architecture outlined in the paper.

It should be noted that the decrease in transfer time enabled by the introduce technique is effective only if the object-content is compressible. Still image files and multimedia content are usually found or produced already compressed in the server to save storage space - no gain is obtained from re-compressing them, since no significant file size reduction is usually achieved with standard compression algorithms. The efficiency of an operational mechanism based on the presented heuristic can be improved by adding a module able to identify the file content type and select a suitable compression algorithm (including no compression at all).

Finally, while the paper focuses on the usage of compression algorithms in a Web service environment, the addressed problem and the main rationales supporting the proposed heuristic can also model other application areas, such as mobile (e.g. cellular phone) Web access which operate over limited-bandwidth networks and under power consumption constraints.

8. Acknowledgment

The authors are grateful for the financial support offered by FAPESP (Fundação de Amparo à Pesquisa do Estado de São Paulo), CNPq (Conselho Nacional de Pesquisa e Desenvolvimento) e CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior).

9. References

[1] Deflate - a lossless data compression algorithm


[3] gzip (gnu zip) compression utility


