SOAP OPTIMIZATION VIA PARAMETERIZED CLIENT-SIDE CACHING

Kiran Devaram and Daniel Andresen
Department of Computing and Information Sciences
234 Nichols Hall, Kansas State University
Manhattan, KS 66506, USA

ABSTRACT

The Simple Object Access Protocol (SOAP) is an emerging technology in the field of web services. Web services demand high performance, security and extensibility. SOAP, being based on Extensible Markup Language (XML), inherits not only the advantages of XML, but its relatively poor performance. This makes SOAP a poor choice for many high-performance web services. In this paper, we analyze the client side processing of a SOAP request and investigate the stages of this processing, where SOAP lags behind its peers in speed. We concentrate on the more popular RPC-style implementation of SOAP rather than the message-style. We then present an optimized design utilizing a caching mechanism at the client side for SOAP messages. We also describe our implementation based on the Apache Java SOAP client, which gives dramatically better performance (800%) over the original code.

KEY WORDS:

SOAP, XML, Network Optimization, Web Computing.

1. Introduction

Lately, there has been a tremendous development in the area of web services. SOAP [1] is one such development, which was conceived when there was a requirement for a standard, and is the standard binding for the emerging Web Services Description Language (WSDL) [3]. SOAP is based on XML [2] and thus achieves high interoperability when it comes to exchange of information in a distributed computing environment. SOAP, carrying the advantages that accrue with XML, has several disadvantages that restrict its usage. SOAP calls have great overhead due to the considerable execution time required to process XML messages. In this paper, we look at one negative side of SOAP: its speed of execution.

We optimize the client-side processing of a SOAP request to the server. We use the Java implementation of Apache [8] SOAP 1.2 and choose the most common model of SOAP that is used in distributed software, the RPC-style, rather than the message-style, which is less popular. This choice is obvious among Web developers, as it closely resembles the method-call model.

The study involves analyzing the SOAP request made by the client to the server when it requests a service from it. This involves profiling of a SOAP RPC client. The profiler that we have chosen is Hpjmeter. The profile data that is collected is then used to investigate the different stages of execution of the client using the profiler. Each of the stages is further examined to find out where the client is spending most of its time. Since SOAP requires messages to be in XML, a typical request from the client involves XML encoding (serialization and marshalling of the payload) before it is sent to the server.

The aim of this research is to make SOAP more efficient to cope with the requirements of a high-performance application or a web service, while still complying with the SOAP standard. After close examination of each phase of its execution, the client side is optimized by using a caching mechanism. This eliminates the need to regenerate the XML from scratch on every call. An experimental performance increase of around 800% is obtained by caching the client requests, which are small in size. A *partial caching* strategy was implemented for parameterized calls to improve the performance further. A secondary goal was to minimize overhead in modifying an existing SOAP application. Another objective was to have zero impact on the server-side code as our implementation mainly concentrates on the client side. We used Apache SOAP 1.2 with Tomcat 4.1 application server.

The rest of this paper discusses related work in section 2 and implementation details in section 3. Section 4 outlines the results of the study, and section 5 presents our conclusion.

2. Related Work

There have been several studies comparing SOAP with other protocols, mainly binary protocols such as Java RMI and CORBA. All of this research has proven that SOAP, because of its reliance on XML, is inefficient compared to its peers in distributed computing. In this section we examine studies [4] [5] [6] which explain where SOAP's slowness originates and consider various attempts to optimize it.

SOAP, relying heavily on XML, requires its wire format to be in ASCII text. This is the greatest advantage of using SOAP, as the applications need not have any knowledge about each other before they communicate. However, since the wire format is ASCII text, there is a cost of conversion from binary form to ASCII form before it is transmitted. Along with the encoding costs, there are substantially higher network-transmission costs, because the ASCII encoded record is larger than the binary original [4]. Reference [4] shows that there is a dramatic difference in the amount of encoding necessary for data transmission, when XML is compared with the binary encoding style followed in CORBA.

Other reasons for SOAP's inefficiency (from [5]) are the use of multiple system calls to send one logical message. Of course, the reason of concern to this paper, XML encoding/decoding, is also mentioned. Some suggestions made by [5] include HTTP chunking and binary XML encoding to optimize SOAP.

Extreme Lab at Indiana University [6] came up with an optimized version of SOAP, namely XSOAP. Its study of different stages of sending and receiving a SOAP call has resulted in building up of a new XML parser that is specialized for SOAP arrays, improving the deserialization routines. This study employs HTTP 1.1, which supports chunking and persistent connections.

Reference [7] states that XML is not sufficient to explain SOAP's poor performance. SOAP message compression was one attempt to optimize SOAP; it was later discarded because CPU time spent in compression and decompression outweighs any benefits [7]. Another attempt in [7] was to use compact XML tags to reduce the length of the XML tag names. This had negligible improvement on encoding, which suggests that the major cost of the XML encoding and decoding is in the structural complexity and syntactic elements, rather than message data [7].

In Reference [9], O. Azim and A. K. Hamid, describe client-side caching strategy for SOAP services using the Business Delegate and Cache Management design patterns. Each study addressed pinpoints an area where SOAP is slow compared to its alternatives. Some present optimized versions of SOAP using such mechanisms as making compact XML payload and binary encoding of XML. While said mechanisms achieved better efficiency, none could match Java RMI's speed and simultaneously preserve compliance to the SOAP standard.

3. Implementation

Our study focuses on the optimization of the client side of a SOAP service. We began by profiling a simple SOAP RPC-style client requesting service from a server. The profile data was studied and the client's job of requesting a service was broken into stages. Each of these stages was examined further and key areas where the client spends most of its execution time were identified. As expected, the client spends a considerable amount of its execution time in XML encoding. In some cases, such as a client application requesting the current stock-quote value of a company, converting binary data into ASCII format takes a significant amount of the computation on the client side while the rest of the client's task is simply to construct a query string requesting the stock-quote value. In such a scenario, XML encoding proves costly and will have a major effect on client performance.

Every client application has a finite set of different requests that are sent to the server over time. It happens quite often that the same request is generated again and again, which involves sending of the same SOAP payload. One such example is a stock application, which makes similar requests to the server querying the stock-quote values. Our study focuses on caching such requests at the client side. The first time the SOAP payload is generated by the client, it is cached in a file and is indexed by a key, which contains information about the type of request that generated this payload. Every time the client needs to send a request, it will first check the cache to see if the request was previously made and cached. If it is, then a simple file I/O operation can fetch the payload from the cache and send it to the server. This relieves the client application from using org.apache.soap.rpc.Call to create the payload again. This increases the execution speed manifold times because creating the SOAP request significantly impedes client performance.

We have implemented a caching mechanism using files. One important aspect to be considered is how the contents in the cache are indexed. The index should contain information about the type of the request that generated that particular SOAP payload. For example, for a client requesting stock-quote value, the index can be the company's name for which the stock-quote value is requested. In case the client needs to request the stock value of the same company, it can flip the payload from the cache using the company's name as a search key. The indexing can be made application-dependent.

We also considered a scenario in which the client may send to the server repeated requests that differ only by the values of a few XML tags in the SOAP payload generated. Consider a web service which provides flight information. The SOAP RPC client requests flight information between two cities by providing the city names as parameters to the server. We use HTTP as the underlying protocol for transporting SOAP XML payloads, though it is not mandatory according to the SOAP specification. Binding SOAP to HTTP provides the advantage of being able to use the formalism and decentralized flexibility of SOAP with the rich feature set of HTTP [1]. To send a request to the server, the SOAP RPC client creates an instance of org.apache.soap.rpc.Call, a java class that encapsulates a

SOAP RPC method call. After specifying the name of the service and the method being invoked, we set the parameters, which in this case are the names of the two cities, using the setParam() method of the Call object. The actual communication with the server is done with the use of the invoke() method of the Call object to make a method call to the server. Fig. 1 shows the SOAP payload that the client generates. Being in ASCII text, this message is very large compared to a similar request from a Java RMI client. Note that the source and the destination cities are stored in the <From> and the <To> tags of the SOAP payload.

Upon examination of the profile data of the SOAP RPC client, it is found that about 50% of the execution time is spent in XML encoding and creating a HTTP connection. XML encoding involves SOAP payload preparation, which is basically serializing and marshalling of the payload before it is transmitted to the server.

Comparing several such requests from the client, it is found that the SOAP payloads differ only in the values of the <From> and the <To> tags. For each such request, the client has to prepare the SOAP payload, which takes a significant amount of processing time involving XML encoding. From this observation, we discover that there are better ways to handle similar multiple calls made by the client. This is the area upon which our study mainly focuses and the area from which the notion of *partial caching* of SOAP payload stems.

In most web services, it is very common to have an interaction between the client and the server in which the client communicates with the server by only passing a few parameters. The SOAP payload generated by the client will be the same each time, except for the tag values of each parameter. In the stock-quote application, the client makes similar requests to the server by querying the stock-quote values using the company name as parameter. *Partial caching* can be employed here to cache such requests on the client side. The first time the SOAP payload is generated by the client using the Call object, it

```
POST /soap/servlet/rpcrouter HTTP/1.0
Host: hostname
Content-Type: text/xml;charset=utf-8
Content-Length:
SOAPAction: ""
Accept-Encoding: gzip

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Body>
<ns1:getFlightInfo xmlns:ns1="urn:FlightInfoService"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<From xsi:type="xsd:string">Madison</From>
<To xsi:type="xsd:string">Las Vegas</To>
</ns1:getFlightInfo>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Fig. 1. SOAP payload generated by a SOAP RPC client.

is cached in a file. During subsequent client requests, only a simple file I/O operation is necessary to fetch the payload from the cache. The client can then replace the values of the tags with the fresh values supplied to it. Fig. 2 shows SOAP client-server architecture in which, the client implements a *partial caching* mechanism. The client then establishes a socket connection at the port where the SOAP service is deployed on that particular host.

The response from the server is ASCII text, which is obtained by listening to the socket through which the request was sent. The response from the server will be in XML, from which the required element is searched by a simple string search, saving the extra time spent to parse the response, which is done by creating an instance of org.apache.soap.rpc.Response.

3.1 Limitations and Requirements

The idea of caching was conceived with the notion that the SOAP request of the client remains the same in most of the cases. However, there are few requirements and limitations for making use of this caching strategy. The primary requirement is that the client should have a fixed number of different types of requests that it can make to the server. Otherwise, for each request, the SOAP payload is saved in the cache, increasing its size. As the size of the cache increases, the time spent in file I/O for each of the following requests increase, which ultimately degrades its performance. We suggest a better caching mechanism as a countermeasure.

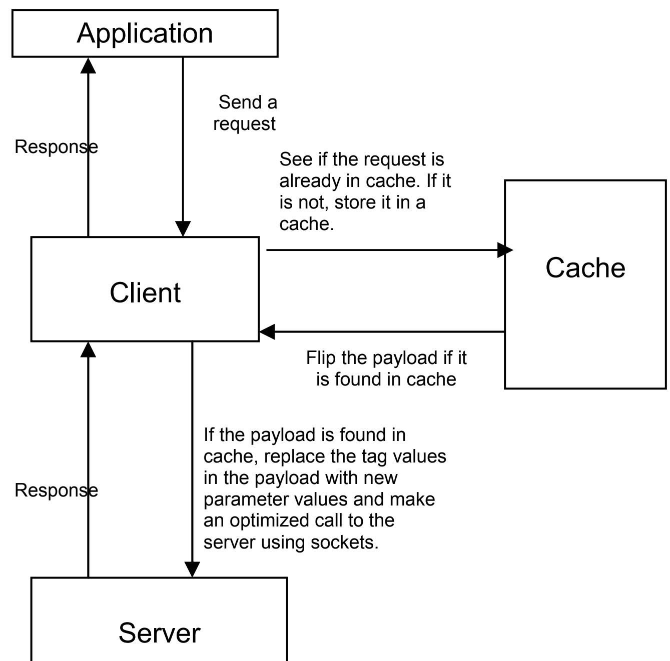

Fig. 2. SOAP RPC client–server using client side caching.

In cases where the client makes different requests all the time, our caching strategy does not improve the performance, rather, we anticipate a decrease in efficiency due to the time involved in cache lookup. Also, the growing size of cache will further hinder the performance. A solution to this is to determine the validity of the data in cache. This can be done using either time-based or notification-based approach [10]. Invalid cache contents can be flushed out. Several of the available techniques of flushing the cache are suggested for better cache implementation.

The *partial caching* mechanism that has been implemented has a better performance when the number of tag-values needing to be updated is small. If the client supplies many different parameters to the server, then using the cache and replacing the tag-values is not always efficient. This is because, as the number of tag-values increase, the time spent in replacing the values of parameters increases. Also, as the number of parameters increase, the size of cache increases, which in turn increases the file I/O computation lowering performance. Though the *partial caching* mechanism is designed to counter the degrading performance of complete caching due to huge cache size, it is more appropriate to use this strategy only for requests involving few parameters.

Another aspect that needs to be addressed here is SOAP fault handling. The SOAP fault element carries error and/or status information within a SOAP message [1]. The SOAP processor at the server side generates a client fault code when it receives an invalid message from the client. This means that the request from the client is improperly formed or does not contain enough information in order to succeed. This is an indication that the message should not be resent as it is and needs correction. These responses from the server require the cache to be flushed, the request to be freshly generated using SOAP libraries, and then recached.

4. Evaluation

This section first lists a series of experiments that we ran to compare SOAP with a binary protocol, Java RMI. These tests enabled us to focus on the stages of client-side processing we later developed. After SOAP RPC client was found to be spending a considerable amount of time encoding the XML payload, the notion of caching the frequently made requests was conceived. We then implemented a caching mechanism on the client side, where the complete SOAP payload is stored in cache and indexed. As discussed earlier, since most of the requests from the client are the same except for the values of the parameters supplied to the server, we also evaluated a *partial caching* mechanism at the client side. The final part of this section will provide a comparative study of the effect of the size of the data transmitted on the performance of each of the implemented strategies.

At first, simple applications of getting a string from the server were implemented in both Java implementations of Apache SOAP and RMI. We used Java 1.4 to test these applications on an Apache Tomcat 4.1 web server. Xerces was used as the XML parser for Apache SOAP 1.2. These applications were tested on SunOS 5.9 running on a 750 MHz, 2 GB RAM Sun Blade 1000 system. The results are shown in Fig. 3. The performance of Java RMI is far better than that of SOAP, and this is evident from the Fig. 3. For this example, Java RMI client spent around 92% of its total execution time for RMI naming look-up, while the SOAP RPC client spent over 52% of its execution time in encoding the XML payload that is sent to the server.

XML encoding, as mentioned in [5] is not the only reason for SOAP being slower than Java RMI. Another reason is making multiple system calls to send a message [5]. In order to optimize the client-side of SOAP RPC, frequently sent requests are stored in cache for future use. This will decrease the client side execution time, as there is no longer a need to create a SOAP payload using the class `org.apache.soap.rpc.Call`. Also, the SOAP payload is transmitted using sockets, saving the time required to establish HTTP connection. This logic was used to implement a modified SOAP RPC client, which now has a caching mechanism. Its performance is compared with both the traditional SOAP and Java RMI in Fig. 3. As the caching mechanism is implemented using files, there is an additional computation involving File I/O, replacing the encoding of XML. Every time the client needs to make a request, it first checks the cache to see if the SOAP payload corresponding to that request is found in the cache; if so the client will flip the payload and send it to the server using Java sockets. The file I/O for the above example took about 46 ms. This technique also involves establishment of Java socket connection with the host where the service is deployed. This cost is, however, meager.

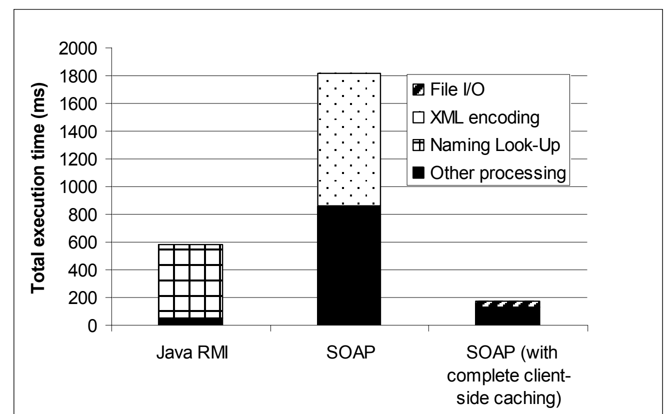

Fig. 3. Comparison of SOAP (with client-side caching) with Java RMI and the traditional SOAP.

Our client-side caching pushes the performance of this client, making it work faster than Java RMI. However, we wanted to evaluate its performance under high loads, i.e. when a large amount of complex data is sent to the server. Under high load the generated SOAP payload is very large, resulting in a huge cache. Combining large requests with a variety of possible client request types causes computation involving file I/O to be very expensive. For example, when the client sends a string array of size 20KB, the time taken for file I/O grows up to 300ms from the previous 46ms. However, it is still faster than traditional SOAP.

As observed earlier, in most cases, the SOAP payloads generated by the client for different requests differ only in the values of a few tags. These tag-values are the parameters supplied by the client to the server. Using this idea, we implemented a *partial caching* strategy on the client side. In this method, we cache the SOAP payload when it is first generated. From then on, every time the client has to make a request, we flip the payload from the cache and replace the values of the tags with the new parameter values and send it to the server using Java sockets. The previous mechanism of complete caching stores each payload even though the request that generated this payload differs only in parameters supplied. We did a comparative study on the effect of size of the data transmitted on the performance of each of the above strategies. The results are presented in Fig. 4. The graph shows that the performance of SOAP degrades as the size of the request increases. We see that SOAP with *partial caching* is more efficient than SOAP with complete caching. The difference in performance is attributed to the growing size of the cache for a client that implements complete payload caching. With the increase in cache size, computation involving file I/O increases, lowering overall performance. But for a client implementing *partial caching*, the size of cache does not grow over time, as it is limited to the number of different requests that the client can make. This limits the time spent for file I/O computation. Using better indexing on

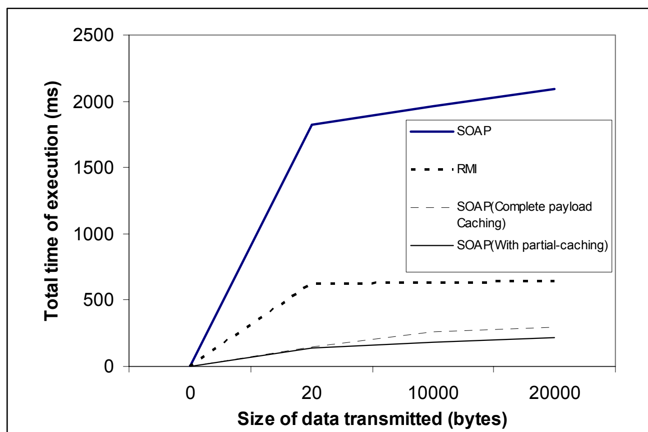

Fig. 4. Performance comparison when large and complex data is sent to the server.

cache increases the overall efficiency as it decreases the time taken to perform a lookup on the cache for the required payload.

Our notion of client side caching of the SOAP payload facilitates building of web services with better performance. Our study identifies areas where SOAP can improve its efficiency. Furthermore, our study was limited to the client side of SOAP. Similar problems do exist at the server side, which demand further study on server-side processing.

6. Conclusions and Future Work

In this paper, we have presented an idea of caching the SOAP payloads at the client side. We also have demonstrated this idea and implemented a variant of this mechanism, which is *partial caching,* which has better performance and a lesser effect on SOAP payload size. Our experiments imply the performance boost that we achieved using these strategies. However, several important issues still remain open for further research. We expect more research on improving the performance of SOAP considering the XML encoding. We are working toward making the caching mechanism more efficient by considering different approaches of caching data. The indexing of the cache contents is one other area that needs further refinement. As stated earlier, similar performance improvements are possible on the server side too. These advancements in SOAP can make it a good choice for not only web services, but also supercomputing.

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

7. References

[1] D. Box et al. “Simple Object Access Protocol 1.1”, Technical Report, W3C, 2000. http://www.w3.org/TR/2000/NOTE-SOAP-20000508/.

[2] World Wide Web Consortium. “Extensible Markup Language”, visited 04-02-03. http://www.xml.org.

[3] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana, “Web Services Description Language (WSDL) 1.1”, Technical Report, W3C, 2001. http://www.w3.org/TR/2001/NOTE-wsdl-20010315.

[4] F. E. Bustamante, G. Eisenhauer, K. Schwan, and P. Widener, “Efficient wire formats for high performance computing”. In *Proceedings of the 2000 conference on Supercomputing*, 2000.

[5] D. Davis and M. Parashar, “Latency Performance of SOAP Implementations”, *Proceedings of the 2nd*

IEEE/ACM International Symposium on Cluster Computing and the Grid, pages 407-412, 2002.
[6] K. Chiu, M. Govindaraju, and R. Bramley, "Investigating the Limits of SOAP Performance for Scientific Computing", Indiana University. Accepted for publication in the *Proceedings of HPDC 2002*. http://www.extreme.indiana.edu/xgws/index.html.
[7] C. Kohlhoff and R. Steele, "Evaluating SOAP for High Performance Business Applications: Real-Time Trading Systems", Proc. of WWW'03, Budapest, Hungary, 2003.
[8] Apache Software Foundation, http://xml.apache.org.
[9] O. Azim and A. K. Hamid, "Cache SOAP Services On The Client Side". http://www.javaworld.com/javaworld/jw-03-2002/jw-0308-soap.html?, March, 2002.
[10] "Caching Architecture Guide for .NET Framework Applications", http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/CachingArchch5.asp