SOAP Optimization via Client-side Caching

Kiran Devaram and Daniel Andresen
Department of Computing and Information Sciences
234 Nichols Hall, Kansas State University
Manhattan, KS 66506, USA
{devaram, dan}@cis.ksu.edu
Office: (785) 532-6350, Fax: (785) 532-7353

Abstract

The Simple Object Access Protocol (SOAP) [1] 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 [2], together with the advantages of XML, however, has a relatively poor performance, which makes SOAP a bad choice for high performance demanding 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. Our concentration is on the more popular RPC-style implementation of SOAP rather than the message-style. We then present an optimized Java implementation of Apache SOAP [8] client, showing experimental improvements in performance (800%), which have been achieved by implementing caching mechanism at the client side.

Keywords: SOAP, XML, Performance study.

1. INTRODUCTION

Lately, there has been a tremendous development in the area of web services. SOAP 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 [3]. SOAP is based on XML 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 few disadvantages, which restrict its usage. As SOAP requires messages to be in XML, processing of these messages takes considerable amount of execution time, which is a great overhead in computation of a SOAP call. In this paper, we look at one such negative side of SOAP: its speed of execution.

In this paper, we analyze the client side processing of a SOAP request to the server. We use the Java implementation of Apache 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. As SOAP requires messages to be in XML, a typical request from the client involves XML encoding, which is basically 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. The client side, after close examination of each phase of its execution, is optimized by using a caching mechanism. An experimental performance increase of around 800% is obtained by caching the client requests, which are of small size. Secondary goal was, to have minimum overhead to modify an existing SOAP application to achieve this performance. Another objective was to have zero impact on the server side code as our implementation mainly concentrates on the client side. We used the Apache SOAP 1.2 with Tomcat 3.02 application server.

The rest of this paper talks about the related work in section 2, and puts the implementation details in section 3. Section 4 presents the results of the study and conclusion follows it in section 5.

2. RELATED WORK

There have been several studies comparing SOAP with other protocols, mainly binary protocols like JavaRMI and CORBA. All of this research has proved that SOAP, because it relies on XML, is inefficient compared to its peers in distributed computing. In this paper we look at some of those studies [4] [5] [6] which explained where SOAP is getting slower and look at various attempts made to optimize it in different ways. All of these studies have targeted to increase the performance of SOAP.

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 its transmitted. Together 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 binary encoding style followed in CORBA. But SOAP, by definition is based on XML.

There have been various other studies, which compared SOAP with binary protocols like JavaRMI and CORBA. Reference [5] does one such study, which finds out reasons why XML causes SOAP to be inefficient. The research in [5] finds out that one source of inefficiency in SOAP is 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. Their 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. They employ HTTP 1.1, which supports chunking and persistent connections.

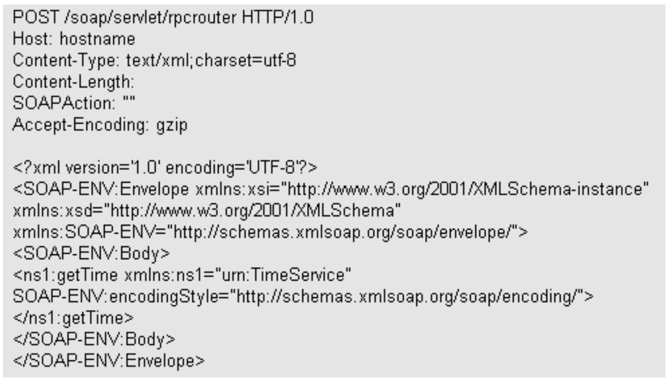
Reference [7] says that XML is not sufficient to explain the SOAP's poor performance. SOAP message compression was one attempt to optimize SOAP, which was later discarded as CPU time spent in compression and decompression the messages, outweighs any benefits [7]. Another attempt in [7] was to use compact 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 the data contained in the message [7].

Each of these studies pinpoints the area where SOAP gets slower by comparing it with its alternatives. Some of them also present optimized versions of SOAP which were conceived as a result of different mechanisms like making compact XML payload, binary encoding of XML etc and achieved a better efficiency. But none of these solutions could make SOAP close to JavaRMI in speed while preserving the software to comply with the SOAP standard.

3. IMPLEMENTATION

Our study focuses on the optimization of the client side of a SOAP service. The work starts with the profiling of a simple SOAP RPC-style client requesting service from a server. The profile data that is collected is studied and the client's job of requesting a service is broken into stages. Each of these stages is further studied and the key areas where the client is spends more time are identified. As expected, the client spends a considerable amount of its execution time in XML encoding. In some cases, like a client application requesting the current stock quote value of a company, this conversion of binary data into ASCII format, takes significant amount of the computation that occurs at the client side as the rest of the client's task is to simply construct a query string requesting the stock quote value. In such a scenario, XML encoding can prove costly and will have a major effect on the performance of the application as far as client side is concerned.

Consider a simple SOAP RPC client requesting the Time-of-day service from a server. We use HTTP as the underlying protocol for transporting SOAP XML payloads though it's 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 use the `invoke()` method of the `Call` object to make a method call to the server, passing the required parameters. Fig. 1 shows the SOAP payload that the client generates. This message is very large in size when compared to a similar request of a JavaRMI client.



```
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:getTime xmlns:ns1="urn:TimeService"
SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
</ns1:getTime>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

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

Upon examination of the profile data of this SOAP RPC client, it is found that, around 40% of the execution time is spent in XML encoding. This involves preparation of the

SOAP payload, which is basically serializing and marshalling of the payload before it is transmitted to the server.

Consider the scenario where the client application sends the same kind of request to the server over and over again. For each request, the client has to prepare the same SOAP payload, which takes significant amount of processing time involving XML encoding. From this observation, we figure out that, there can be a better way to handle similar multiple calls made by the client. This is the area, our study mainly focuses on. This is where the notion of caching the SOAP payload comes up.

Every client application has a finite set of different requests that are sent to the server over time. It happens quite often that 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. The idea that is presented in this paper is to cache 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 the 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, which is then sent to the server. This relieves the client application from creating the payload again using org.apache.soap.rpc.Call, increasing the execution speed manifold times. Fig. 2 shows a SOAP client-server architecture in which, the client implements such a caching mechanism.

We have implemented caching mechanism using files. An 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 stock quote value requesting client, 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, then it can flip the payload from the cache using the company's name as search key. The indexing can be made application dependent.

Once it is found that the present request was already sent and has been stored in the cache, the client application, using the search key, flips the XML payload and sends it to the server using Java Sockets. A socket connection must be set open to the port at which the SOAP service is deployed on that particular host. The response from the server is an ASCII text, which is obtained by listening to the socket through

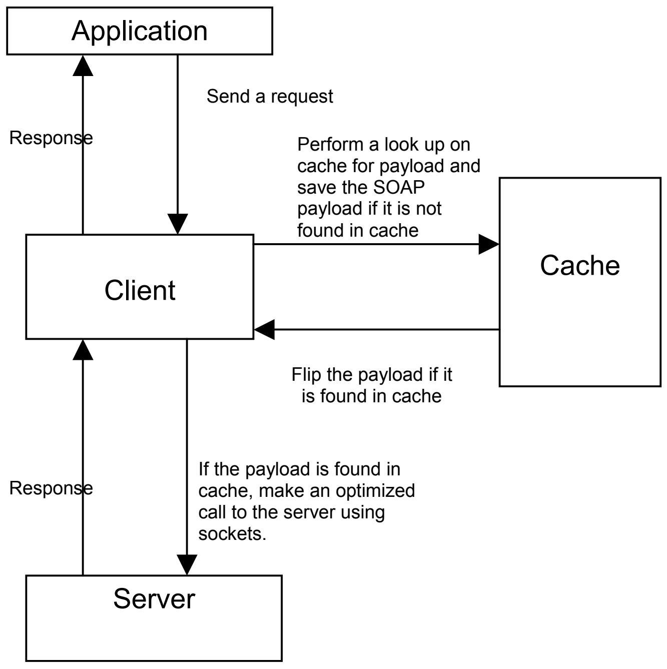

Application
Send a request
Response
Perform a look up on cache for payload and save the SOAP payload if it is not found in cache
Cache
Client
Flip the payload if it is found in cache
Response
If the payload is found in cache, make an optimized call to the server using sockets.
Server

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

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.

4. EVALUATION

This section first lists a series of experiments that we ran to compare SOAP with a binary protocol, JavaRMI. These tests enabled us to focus on the stages of client side processing we later worked on. 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. The later part of this section presents the comparative study between the performance of SOAP and SOAP with client side caching. The effect of passing large and complex data types to the server on performance of SOAP with client side caching has also been evaluated.

At first, simple applications of getting a string from the server were implemented in both Java implementation of Apache SOAP and JavaRMI. We used Java 1.4 to test these applications on Apache Tomcat 3.02 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, 1GB main memory Sun Blade 1000 system. The results are shown in Fig. 3. The performance of JavaRMI is far better than that of SOAP and this is evident from the Fig. 3. For this example, JavaRMI spent around 47% of round trip

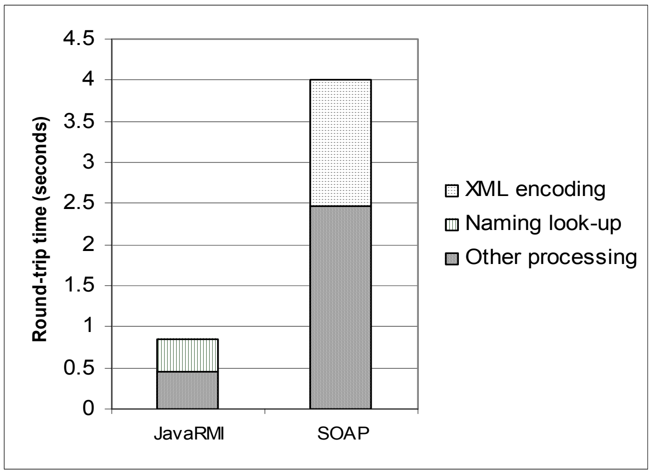


Fig. 3. Comparison of RMI with SOAP.

time for RMI naming look-up, while the SOAP RPC client spent over 39% of its round trip time in encoding the XML payload that is sent to the server.

XML encoding, as said in [5] is not the only reason for SOAP being slower than JavaRMI. Another reason is making of 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 JavaRMI in Fig. 4. As the caching mechanism is implemented using files, there is an additional computation involving File I/O, replacing the encoding of XML. It also involves establishment of Java socket connection with the host where the service is deployed. This cost is, however, meager. The performance of SOAP using client side caching is over 800% more than that of SOAP, which uses XML encoding.

Our client side caching pushes the performance of this client further, making it work faster than JavaRMI. We, however, wanted to evaluate its performance under high loads, i.e. when large amount of complex data is sent to the server. For this, we implemented clients in all three ways, JavaRMI, SOAP and SOAP with client-side caching, which sent 20KB of string array. The results of this experiment are shown in Fig. 5. The sending of a complex data type, like an array, involves more XML encoding as the SOAP payload now contains many more tag value pairs. This further degrades the performance of SOAP which is now over 5 times worse than that of JavaRMI. However, the performance of SOAP with client-side caching persists to be lot better as it involves only File I/O and socket connection establishment. The increase in the time spent in File I/O is minimal making SOAP with client side caching perform better even for large SOAP payload transmissions.

Earlier studies in this topic focused on making SOAP faster by different means like modifying the XML parser, compressing the XML payload etc, but none of them worked on the idea of reusing the payload that is already generated. The caching mechanism works great and usage of better indexing on cache increases the overall efficiency further 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 can facilitate building of web services with better performance. Our study shows areas of SOAP to work on, to improve its

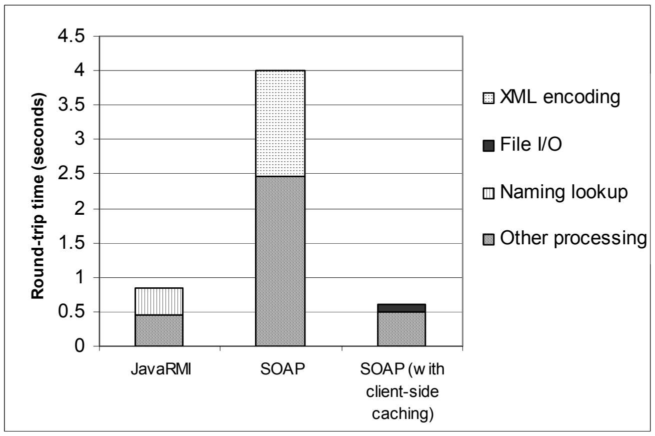


Fig. 4. Comparison of SOAP (with client-side caching) with JavaRMI and the traditional SOAP.

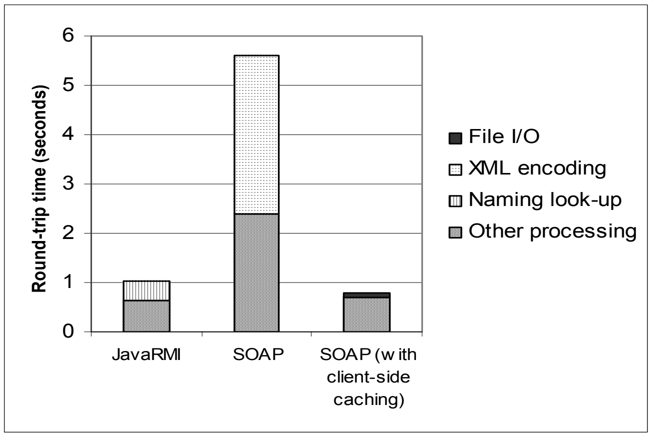


Fig. 5. Performance comparison when large and complex data is sent to the server

efficiency. Furthermore, our study was limited to the client side of SOAP. Similar problems do exist at the server side, which demand study on the 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 various other issues that effect its applicability for real life applications involving high speed demanding web services. Our experiments imply the performance boost that we achieved using this mechanism. 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 towards making the caching mechanism work better, considering different approaches of caching data. The indexing of the cache contents is one other aspect, which needs further refinement. More often than not, it happens that only a few of the XML elements of the payload change, while the rest of the nodes and values remain the same. With this finding, future work also includes usage of DOM for large payloads facilitating modification of the XML elements that need to be updated and keeping the rest of the document the same, which might save valuable time of creating the XML payload each time.

Acknowledgments: This material is based in part upon work supported by the National Science Foundation under award numbers ITR-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”, accepted to WWW2003, Budapest, Hungary, 2003.

[8] Apache Software Foundation, http://xml.apache.org.