LYE: HIGH-PERFORMANCE SOAP WITH MULTI-LEVEL CACHING

Venkatesh Prasad Ranganath, David M. Sexton, Daniel A. Andresen
Department of Computing and Information Sciences
Kansas State University
234 Nichols Hall, Manhattan KS, USA
{rvprasad, dms3333, dan}@cis.ksu.edu

ABSTRACT
Simple Object Access Protocol (SOAP) is a dominant enabling 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 present new approaches to leverage multiple levels of caching and template-based customized response generation in a SOAP server to improve performance while maintaining complete protocol compliance. We demonstrate its practicality by implementing a demonstration system under Linux that provided speedups of over 600% for sample applications.

KEY WORDS
SOAP, eCommerce, protocols, WWW, web services

1 Introduction

Recently, there has been tremendous development in the area of web services in eCommerce, high-performance computing, and Computational Grid. In response to the need for a standard to support web services, SOAP became the standard binding for the emerging Web Services Description Language (WSDL) [1, 2]. SOAP is based on XML [3] and thus achieves high interoperability when it comes to exchange of information in a distributed computing environment. While carrying the advantages that accrue with XML, it has several disadvantages that restrict its usage. SOAP calls have a large overhead due to the considerable execution time required to process XML messages. In this paper, we partially mitigate a primary negative of SOAP: its speed of execution. We do this through the selective implementation of caching on the server side assuming that a number of applications send the same information, repetitively, in a structured form. Examples might include "stock tickers," game broadcasts, or airline ticket pricing. Each of these is likely to send the same information multiple times, yet the information is also continuously changing, so a simple reverse proxy cache [4] is inadequate. Furthermore, we also note that the response format is fixed, so we demonstrate how a customized response template can (a) be automatically generated, and (b) show this can speed up response generation by a very significant amount (over 300% in some test cases).

In our previous work, we implemented caching on the client-side and server SOAP engines, and achieved speedups of over 800% for the client and 70% for the server compared with the standard Apache implementations [5, 6, 7].

In this paper, we optimize the server-side processing of a SOAP request, achieving speedups of 2500% for structured datatypes and achieving at least a small optimization for all "interesting" requests. In the following section, we will discuss work related to optimizing SOAP. In Section 3 we shall present the caching and template-based serialization strategies in detail. Cost analysis and experimental results of the proposed strategies are presented in Section 4. We conclude the paper with future work and summary of our contributions.

2 Background

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 [8, 9, 10] which explain where SOAPs slowness originates and consider various attempts to optimize it.

In SOAP, objects are encoded in a structured form as elements in an XML documents. This simplifies interoperability between various services that use different binary formats to represents objects internally. However, as the wire format is ASCII text-based, there is a cost associated with encoding/serializing[1] and decoding/deserializing the objects to and from the wire format. Bustamante et.al. [8] have shown that the cost of encoding/decoding of data as text in XML is relatively high when compared to encoding/decoding of data in a custom binary format as done in Java RMI and CORBA (IIOP). They also observe that ASCII encoded data incurs higher network-transmission costs as it is larger than binary encoded data. Another reason for SOAPs inefficiency is the relatively high number of system calls required to send one logical message [9]. Some suggestions made in [9] include HTTP chunking and binary XML encoding to optimize SOAP.

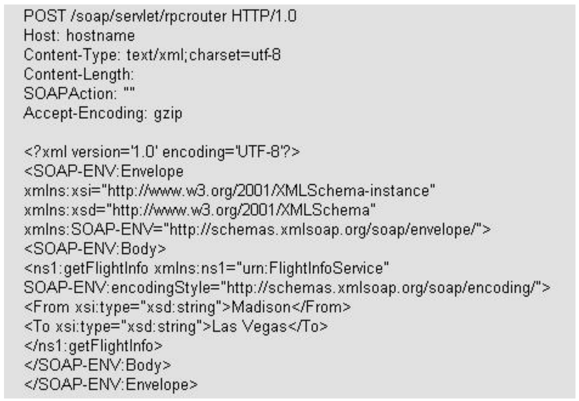
[1] We shall use the terms encoding and serializing interchangeably.

Extreme Lab at Indiana University developed an optimized version of SOAP, namely XSOAP [10]. 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.

Kohlhoff et.al. [11] state that XML is not sufficient to explain SOAPs poor performance. SOAP message compression was one attempt to optimize SOAP; it was later discarded because CPU time spent in compression and decompression outweighed any benefits [11]. Another approach 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 [11].

O. Azim and A. K. Hamid [12] 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 RMIs speed and simultaneously preserve compliance to the SOAP standard.

In previous efforts, upon examination of the profile data of an SOAP RPC client, it was found that, approximately 50% of the execution time is spent in XML encoding and creating a HTTP connection[6]. Similar effort is expended on the server-side to encode the response [7].



```
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>
```

Figure 1. SOAP payload generated by SOAP RPC client. Server responses are similar.

Comparing several such requests from the client, it was found that the SOAP payloads differ only in the data embedded in XML elements. In our example of airline ticket pricing, the data embedded in elements *From* and *To* (Figure 1) change for each request. For each such request, the client has to prepare the SOAP payload and in response the server has to prepare a similar SOAP payload as well. In the due process, XML encoding takes up significant amount of time at both ends of the communication. Hence, reducing the time taken for XML encoding is a way to improve performance.

3 Details

In this section we present a brief overview of Axis, an apache implementation of SOAP, followed by description of the strategy for caching and template-based serialization.

3.1 Apache Axis Overview

Axis [13] is a SOAP implementation available from Apache. A bird's eye view of control flow on the server-side in Axis is illustrated in Figure 2. Upon reception of a *request message*[2] at the server, it is decoded/deserialized into binary form. The request object (request[3]) then traverses through a *chain* of *handlers*. The handlers accept a request and either return a modified request object that shall traverse the chain further down or return a response object (request[4]) that shall trigger the backtracking of the chain. A handler that causes the backtracking of the chain is called the *pivot* and it realizes the actual service-related functionality. In other words, the actual service logic occurs in the *pivot*. When the response object reaches the beginning of the chain, it is encoded/serialized by a *serializer* and the response message is sent to the client.

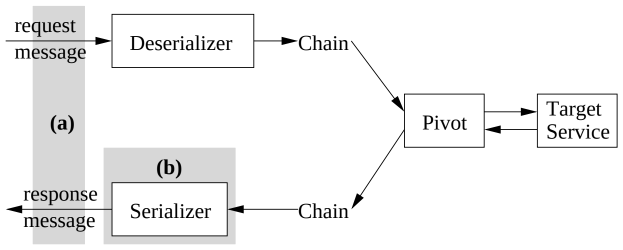


Figure 2. Server-side control flow in Axis.

3.2 Caching

Based on the above architecture, we have identified two locations in the control-flow at which caches can be placed to improve performance under various circumstances. We shall describe the intuition and the strategy the following sub-sections.

3.2.1 Request Caching

In applications such as stock tickers and game broadcasts, identical request messages to a service results in identical

[2]By request message we mean the SOAP part of the message.
[3]From hereon, we shall refer to the request object as "request".
[4]From hereon, we shall refer to the response object as "response".

response messages during a period of time independent of the clients. In other words, for some applications, a *time-to-live (TTL)* value can be associated with responses. Hence, if the value of TTL is available at the time of service deployment then it can be leveraged to enabling caching.

The strategy is to cache the response messages with the request messages as the key to the cache. A response message cache is installed at location (a) in Figure 2. Upon reception of a request message, the cache consulted. If a response message exists and the TTL of the service has not expired, then the cached message is returned. If either the cached response message exists but the TTL has expired or there is no cached response message, the process to service a request message as described in Section 3.1 is executed and the returned response message is cached. In case of cache hits, circumventing deserialization, chain traversal, service invocation, and serialization improves service response time, and in case of cache misses, a small overhead of cache lookup is incurred.

Any request messages that may result in client specific response messages will contain client specific information. Hence, the difference between similar request messages from different clients will ensure that a client will always receive a cached response message targetted for it. The simple strategy assumes a simple TTL cycle in which the TTL period starts when the service is deployed and repeats itself till the service is undeployed with no changes in between. For more interesting ttl cycles, the proposed strategy will need more support from the service to keep the cache "clean".

As part of implementation, services can be deployed into an Axis server via *Web Service Deployment Descriptor (WSDD)*. As the language used in this descriptor is extensible, it is possible for the user to specify caching and the TTL for a service via the WSDD. Hence, existing SOAP services can utilize caching with minor changes to their deployment descriptors.

3.2.2 Element Caching

The encoded form of a data value will be identical independent of the service that generated it. Hence, if it is assumed that all services use a general pool of serializers then caching the encoded form of data values in the serializer can improve performance.

In this strategy, each serializer provided with Axis is extended with a cache (at location (b) in Figure 2). During serialization, the cache is consulted. If the encoded form of the given data value exists, the cached encoded form of the data is returned. If not, the value is encoded, the encoded form is cached, and it is returned as the result.

In contrast with the previous strategy, this strategy can be employed in existing deployment of SOAP servers with no change to the services. The caches are serializer-specific and common across a server. Also, this strategy applies well in situations where the response messages change across each SOAP call.

3.3 Template-based Serialization

As in any protocol, the structure of the messages in SOAP are fixed while describing the service via WSDL. Moreover, as SOAP uses XML to represent messages, each piece of data in a message is represented in an XML element or as an attribute of an XML element. Hence, a message in it's textual form contains text that imposes the structure on the data (*tags*) and text that represents the actual data. Also, the tags in a message do not change while the data changes. In terms of template, the tags is the static part of a template while the data is the dynamic part of a template. Hence, it is be possible to use templates to generate the textual form of messages or serialize data into a message.

Using templates to generate HTML/XML documents on the fly is not novel. Java Server Pages (JSP) is a good example in which the concept of templates has been used extensively. From the perspective of SOAP, Soaplet[14] provides a framework to generate SOAP requests using templates and extract data from SOAP responses using XPath. Auto-generation of templates is also partially supported in Soaplet.

In the light of optimization, we explored the use of templates on server side to generate responses and its implications. The basic idea is to process the WSDL specification and generate a template that can be used in a generated serializer to serialize the results of a service within Apache SOAP framework. We hand coded templates and serializers simulating a automated generator. In doing so, we discovered the subtleties the generator would need to handle and also the benefits of having such a generator.

```
<schema>
  <complexType name="ComplexObject">
    <sequence>
      <element name="childObject" nillable="true"
        type="tns1:ComplexObject"/>
      <element name="intArray"
        type="impl:ArrayOf_xsd_int"/>
      <element name="intNum" type="xsd:int"/>
    </sequence>
  </complexType>
</schema>
```

XML schema

```
<co>
#if (${childObject1})
${childObject1}
#end
<intArray xsi:type="soapenc:Array"
  soapenc:arrayType="xsd:string[${intArray.size()}">
#foreach ($item in ${intArray})
  <item>$item</item>
#end
</intArray>
#end
<intNum xsi:type="xsd:double">\${intNum}</intNum>
</co>
```

Template

Figure 3. XML schema for a message and the generated template.

Given a WSDL specification of the service, we can determine the structure of the response messages based on the corresponding schema definition. Figure 3 contains a schema for a complex type and the template to generate an instance of that type. Apache SOAP implementation provides a WSDL-to-Java compiler. This compiler generates Java classes corresponding to the complex types used in the messages in a WSDL file. The service implementation can then use instances of these classes as response objects. As the structure of the response object to be serialized is known, we can use this information to generate a walker/visitor that walks/visits the containment hierarchy of a particular language (Java) type and encode each element in the hierarchy. These results of encoding fill particular holes in the generated template. Hence, the template and the walker can be combined to realize a serializer in the SOAP framework. As an implementation detail, Apache SOAP implementation comes with a set of default serializers which are used unless specified otherwise via a WSDD. The WSDD fragment to use a customized generator can also be automatically generated.

As for the automation of this process, from our experience in translators, as most service artifacts draw from WSDL and as WSDL specification is rich, we believe that automation is possible. Although we did initially attempt to implement such a generator, the sheer possibilities in XML Schema proved to be overwhelming. However, the hand coding experience proved to be rather simple and easy as we better understood the implications of the possibilities in XML Schema on the structures (not the values) of the messages. Hence, we believe that further understanding of these implications will simplify the implementation of the generator.

The primary benefit of this approach is it decouples the serialization of the message from the assembling of the message data. This provides opportunity to optimize the serialization process independently. Hence, server-side performance can be improved by using a faster template instantiation technique that draws from information available in WSDL or some other ancillary specification. We have also found that while using this approach the performance improves as the structure of the message becomes more complex.

4 Experimental Results

In this section, we present a cost analysis for the strategies to gain insight into the expected performance gains followed by a description of the conducted experiments along with the empirical data collected to verify the outcome of the cost analysis.

4.1 Cost Analysis

In general, each request is deserialized into an object, the object is passed as argument to a function invoked on the service, and the response object is serialized. Let *d*, *i*, and *s* be the time taken for these operations, respectively.

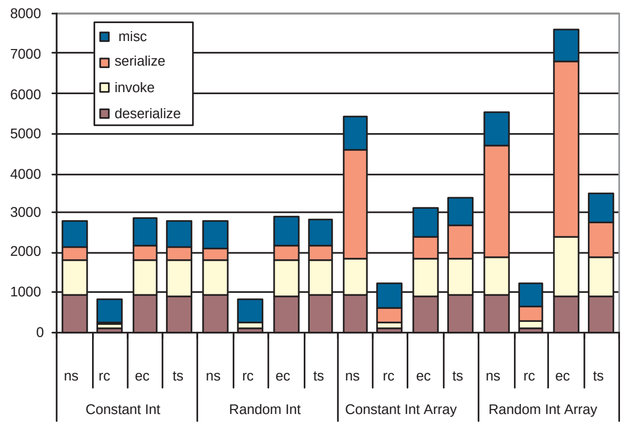


Figure 4. Comparison of response times for simple objects.

During serialization in Axis, each element in the containment hierarchy of the response object is visited, converted to a string (s_{str}), and appended to a serialization buffer (s_{app}). Even the tags of the element are appended to the buffer. However, we ignore their contribution in the analysis as it a small constant for each element. If *n* elements occur in the containment hierarchy, then $s = (s_{str} + s_{app}) \times n$. In short, the time taken to respond to a request in the Apache SOAP server is $T_{apache} = d + i + (s_{str} + s_{app}) \times n$.

In case of services that return the same response for a request repeated over a period time called *time-to-live (TTL)*, we can use request caching approach. Suppose r requests occur per second and t_{ttl} is the TTL in seconds. Then, $r \times t_{ttl}$ calls occur in t_{ttl} seconds. Of these calls, the cost of the first call in the TTL interval will be $t_1 = d+i+(s_{str} + s_{app}) \times n + c$ where c is the time taken to cache the response in it's serialized form. Subsequent, $r \times t_{ttl} - 1$ calls will be serviced based on the cache, the cost for these calls will be $t_j = l$. Hence, the total cost to respond to each call will be $t_{RC} = ((d+i+s+c)+((r \times t_{ttl}-1) \times l))/(r \times t_{ttl})$. In other words, the time taken to respond to each message reduces approximately by a factor of $(r \times t_{ttl})$.

If sub-element caching is used, then the time taken to generate the string representation of an element will be $l + p_m \times s_str$ where l is the cache lookup time and p_m is the probability of a cache miss. Hence, $T_{SC} = d + i + (l + p_m \times s_{str} + s_{app}) \times n$. In comparison with the approach in Axis implementation, this approach will perform better if $p_m + l/s_{str} < 1$.

If template-based serialization is used, then the relation between s and s_{TS} (time to serialize the response using a template) controls the performance cost. As the structure of the message gets complex, s_{TS} reduces as the number of append operations decreases. Furthermore, if the length of the elements as strings is known then s_{TS} can be further reduced. Hence, $T_{TS} < T_{apache}$ when $s_{TS} < s$ and this should occur is many cases.

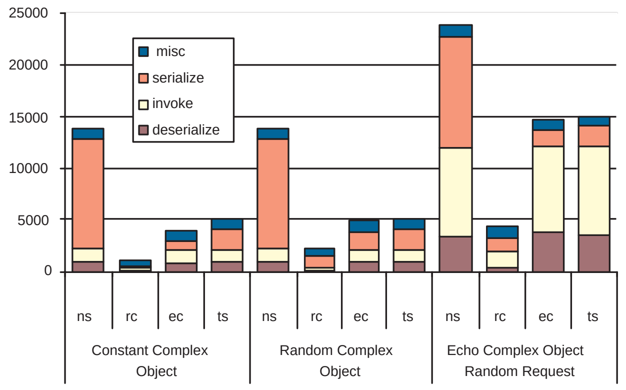


Figure 5. Comparison of response times for complex objects.

The amount of available memory, the cache size, and cache flushing policy will influence *l*, *c*, and p_m. This will influence the response time and scalability when caching is used. Hence, appropriate configuration holds the key to scalability. In the case of template-based serialization, a small amount of memory per type and usually small number of types on a SOAP server should provide excellent scalability with minimum extra memory requirements.

4.2 Experiments

The experimentation platform was a Linux 2.4.x server containing two Athlon 1900+ processors and 2GB of RAM. The Axis SOAP server (v1.1) and the services were hosted inside a Apache Tomcat server v5.0.19 run on SUN JVM (build1.4.2_04-b05). We used Velocity Template Engine [15] in experiments related to template-based serialization. For the purpose of timing, we used `sun.misc.Perf` class included in the j2sdk 1.4.2 with a resolution of 1M ticks per second. All results are averaged over multiple runs.

We used five test cases to measure the performance improvements provided by each strategy. In each test case, a client invoked 100 identical SOAP calls on the server. The test cases differed by the value and the structure of the data returned in response. In the first test case, a integer was returned as response and the same integer was returned in all the 100 calls. The second test case was similar to the first test case except that a random integer was returned for each call. In the third and fourth test cases, integer arrays were returned. The content of the array remained constant across calls in the third test case while it was randomized in the fourth test case. In the fifth test case, a complex object whose structure was similar to the structure described in Figure 3 was returned. The value of the complex object remained constant across the calls.

Each test case was executed with no strategies (*ns*), with request caching (*rc*), with element caching (*ec*), and using template-based serializers (*ts*). In the case of request caching, a TTL value of ***15*** seconds was used. The cumulative timing information for the test cases under different settings are illustrated in Figures 4 and 5. Specific cases for constant simple and complex objects are broken down in Tables 1 and 2.

For most cases, a significant reduction in serialization costs were achieved through caching and a template-based encoding strategy. More specifically, for simple objects, little to no speedup was found except for the *rs* case, which avoids the cost of serialization completely. This is to be expected, given the minimal costs of encoding something like a single integer. On the other hand, more complex objects like arrays often took one-third to one-half of the time for unmodified Axis calls.

For distinctly cache-unfriendly scenarios, such as the random array (which requires the overhead of a cache lookup and insertion for each call), element caching proved ineffectual, but template-based serialization provided a major performance improvement. Choosing the correct strategy for a service is left to the service deployer.

5 Future Work and Summary

In the future we plan to study the impact of memory on the proposed strategies and various cache management options, as well as develop strategies to mitigate head-of-line (HOL) blocking effects in the cache. From the developer's perspective, we plan to explore the automated generation of response (and request) templates, whose practicality was clearly shown in Section 4. As the experiments were executed on Axis, we plan to submit our code for inclusion into the primary Axis code base.

In this paper we have presented a new approach for accelerating the performance of a vital portion of the eCommerce infrastructure through the multi-level caching of responses and customized template generation. Our analysis predicts, and experimental results confirm, that this approach can give substantial speedups (600+%) for many practical applications, while exacting little to no performance penalty for applications unsuited to the architecture beyond the memory devoted to the cache. The latency added by the cache is minimal for most types of requests, and the user has the option of turning off caching for services which are known to be cache-unfriendly.

Acknowledgments

We wish to thank Kiran Devaram for his initial implementation of SOAP caching. Additionally, 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.

	total	d	i	c	l	s	misc
ns	5421	935	907	0	0	2763	816
rc hit	568	0	0	0	19	0	549
rc miss	6202	941	1340	18	205	2824	879
ec	3064	915	927	.14	126.9	531.3	764
ts	3412	931	921	0	0	832	728

Table 1. Average time information for “constant integer array” test case.

	total	d	i	c	l	s	misc
ns	13887	928	1323	0	0	10610	1026
rc hit	589	0	0	0	19	0	570
rc miss	14539	933	1642	219	18	10630	1097
ec	3793	896	1189	.37	403	931.6	873
ts	5068	912	1281	0	0	1897	978

Table 2. Average time information for “constant complex object” test case.

References

[1] Simple object access protocol (soap) 1.1, Feb. 2003. http://www.w3.org/TR/SOAP/.

[2] Web Services Description Language (WSDL), 2001. http://www.w3.org/TR/wsdl.

[3] T. Bray, J. Paoli, and C. Sperberg-McQueen. Extensible Markup Language (XML) 1.0. *W3C*, Feb. 1998. http://www.w3.org/TR/1998/REC-xml-19980210.

[4] D. Wessels and K. Claffy. ICP and the Squid Web cache. *IEEE Journal on Selected Areas in Communication*, 16(3):345–357, 1998.

[5] K. Devaram and D. Andresen. SOAP optimization via parameterized client-side caching. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS 2003)*, pages 785–790, Marina Del Rey, CA, Nov. 2003.

[6] K. Devaram and D. Andresen. SOAP optimization via client-side caching. In *Proceedings of the First International Conference on Web Services (ICWS 2003)*, pages 520–524, Las Vegas, NV, June 2003.

[7] D. Andresen, K. Devaram, and V. P. Ranganath. LYE: a high-performance caching SOAP implementation. In *Proceedings of the 2004 International Conference on Parallel Processing (ICPP-2004)*, Montreal, Canada, Aug. 2004.

[8] F. E. Bustamante, G. Eisenhauer, K. Schwan, and P. Widener. Efficient wire formats for high performance computing. In *Proceedings of Supercomputing 2000*, pages 64–64, 2000.

[9] D. Davis and M. Parashar. Latency performance of SOAP implementations. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 407–412, 2002.

[10] K. Chiu, M. Govindaraju, and R. Bramley. Investigating the limits of SOAP performance for scientific computing. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing HPDC-11 2002 (HPDC'02)*, page 246. IEEE Computer Society, 2002.

[11] C. Kohlhoff and R. Steele. Evaluating SOAP for high performance business applications: Real-time trading systems. In *Proceedings of WWW2003*, Budapest, Hungary, 2003.

[12] O. Azim and A. Hamid. Cache SOAP services on the client side. *JavaWorld: IDG's magazine for the Java community*, Mar. 2002. http://www.javaworld.com/javaworld/jw-03-2002/jw-0308-soap.html.

[13] Apache axis: An implementation of the soap. This software is available at http://ws.apache.org/axis/.

[14] Soaplet, June 2004. http://www.codehaus.org/ dandiep/soaplet/.

[15] Velocity: A java-based template engine. This software is available at http://jakarta.apache.org/velocity/.