SOAP OPTIMIZATION VIA PARAMETERIZED CACHING

by

KIRAN KUMAR DEVARAM

B. E., Osmania University, India, 2001

A REPORT

submitted in partial fulfillment of the

requirements for the degree

MASTER OF SCIENCE

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY

Manhattan, Kansas

2003

Approved by:

Major Professor
Dr. Daniel Andresen

ABSTRACT

The Simple Object Access Protocol (SOAP) is an emerging technology in the field of Web services. Due to its dependence on Extensible Markup Language (XML), SOAP achieves platform-independence and high interoperability when it comes to exchange of information in a distributed computing environment. SOAP, together with the advantages of XML, inherits its relatively poor performance. This makes SOAP a wrong choice for many high-performance Web services. This report aims at optimizing the SOAP protocol by proposing a parameterized caching technique at both the client and the server sides. The optimization done on the client-side involves profiling of the client side processing of a SOAP request and employing different caching strategies for improved performance. The server-side optimization proposes a modification to the Apache SOAP engine to enable caching.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I am very grateful for having the opportunity to be involved in research with Dr. Daniel Andresen, my major professor. I am thankful for his enormous patience to guide me at every milestone in the project and to respond to the countless emails that I sent.

I also thank Dr. Gurdip Singh and Dr. Mitchel Neilsen for serving as my committee.

I thank all the faculty members, friends and staff of the Department of Computing and Information Sciences, especially Ms. Delores Winfough, for their help and guidance through out my MS program.

DEDICATION

to

my Dad,

for everything I am.

1 Introduction

1.1 What are Web services?

A *Web service,* as defined by IBM, "*is a software interface that describes a collection of operations that can be accessed over the network through standardized XML messaging*." It uses protocols based on the XML [2] language to describe an operation to execute or data to exchange with another Web service. Web services are being implemented in a wide variety of technologies but will provide services that can be accessed using a standardized protocol.

1.2 SOAP Overview

SOAP [1], an acronym for *Simple Object Access Protocol*, is one of the recent developments in the area of distributed computing. It is one of the core technologies of Web services. World Wide Web consortium [11] defines SOAP in its specification as "*a lightweight protocol for exchange of information in a decentralized, distributed environment.*" It mainly consists of three parts: an envelope that is used to describe the content of a message and some clues on how to process it, a set of encoding rules for specifying custom defined data-types and, a convention which describes the application of the envelope and the data encoding rules for representing Remote Procedure Calls and responses.

SOAP is the standard binding for the emerging Web Services Description Language (WSDL) [3] and is for communication between loosely coupled systems. SOAP achieves this goal by specifying the wire format of the information that is exchanged between the loosely coupled systems. SOAP uses Extensible Markup Language (XML) for this and is so a wire-level specification.

Due to its dependence on XML, SOAP achieves platform-independence and 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. This report analyses one such negative side of SOAP, its performance and suggests some techniques to improve it.

1.3 SOAP: Performance

The factor that inevitably comes up in a discussion about SOAP is its performance. When sending data over a network, the data must comply with the underlying transmission protocol, and be formatted in such a way that both the sending and receiving parties make sense out of it. As mentioned earlier, SOAP specifies its encoding based on XML data encoding. This implies that SOAP protocol tends to generate verbose requests and responses. In addition to the high transmission costs due to heavy requests and responses, SOAP calls have great overhead due the considerable execution time required to process the XML messages embedded in a SOAP envelope. This makes SOAP a poor choice for many high performance Web services. Since from its inception, SOAP has been compared with other protocols, mainly binary protocols such

as RMI and CORBA. SOAP is a natural choice when the three characteristics – language independence, platform-independence and interoperability are important in designing a client-server model. But its usage is questioned when performance is one of the requirements. Due to the latency associated with the transmission of the SOAP messages and the computation involved at both the client and the server ends, a high performance and high scalability Web service is not achievable.

1.4 Solution?

This report studies the different processing stages at both the client and server side, which involves building and breaking of the SOAP envelope at the respective ends of the client-server architecture. The study reveals the areas of processing where the client and the server spend most of their computation time. These are the areas, which need maximum optimization for an over-all increase in the performance of the Web service. The solution to the problem associated with building of a SOAP envelope at both the client and server ends, is to use "*Caching*".

1.5 Outline

The rest of the report discusses related work in this area of research in section 2 and implementation details in section 3. Section 3 talks about work carried on both the client and the server side of a SOAP Web service. It discusses the processing stages at client and server, and the approach followed to optimize them. It also explains the different caching techniques used on the client side together with the limitations and requirements for this strategy to function effectively and, finally, it lists the technologies used for this

implementation. Section 4 outlines the results of the study. It does a comparative study of the different optimization techniques employed on the client and server side of the Web service. Section 5 presents the conclusion of this report.

2 Background and Related Work

Since the time Web services are thought of as a replacement to distributed computing on the web, the usage of SOAP for high performance Web services and distributed computing is argued. Previously RMI or CORBA were chosen for programming distributed systems. Java's RMI provides an elegant solution for a java developer. But Remote Method Invocation technology lacks the openness to other distributed protocols, which hinders its interoperability. The Common Object Request Broker Architecture (CORBA) [12], introduced by Object Management Group (OMG), is a specification for client/server architecture. It specifies the remote invocation of objects using Internet Inter-ORB Protocol (IIOP). CORBA offers the support of language independence, but is generally not considered for distributed applications for its complex specification, which requires considerable infrastructure. Also, CORBA calls are not firewall friendly. None of these technologies are designed to scale up to the web.

The popularity of XML led to the development of XML-RPC, which is a precursor of SOAP. XML-RPC offers the support which all of its peers – RMI, CORBA, DCOM, could not provide. It offers set of implementations that allow software running on disparate operating systems, running in different environments to make procedure calls over the Internet. It is remote procedure calling using HTTP as the transport and XML as

the encoding. XML-RPC is designed to be as simple as possible, while allowing complex data structures to be transmitted, processed and returned. SOAP picked up from where XML-RPC was left frozen. Since then, 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.

In client-server model, or in reality in any peer-to-peer model of communication, the data that is sent over a network must comply with the underlying transmission protocol and should be formatted in such a way that both the sending and receiving parties make sense out of it. This is called *data encoding*. 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. This provides the luxuries of language independence and high interoperability. However, since the wire format is ASCII text, there is a cost of conversion from binary form to ASCII form before it is transmitted. This process of conversion of binaries objects to XML is called as serialization. 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] mainly illustrates the impact of wire-format on the performance and 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 report, 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 (previously called SOAPRMI). 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 de-serialization 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

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. With this in mind, no attempt was made to achieve efficiency by XML compression or modifying the existing XML parser. A secondary goal was to provide a solution for the client-side (or server-side), which has zero impact on the server-side (or client-side) code. Another objective was to minimize overhead in modifying an existing SOAP application to employ the suggested optimization techniques.

3.1 Web Service Invocation: Overview

This section gives an overview of the Web services invocation model. It illustrates the different stages of the request-response scenario that typically happens in the client-server model. Fig. 1 shows the Web service invocation model. From the figure, we can understand what actually happens when a client wants to send a request to a Web service. The whole process of invocation of a Web service can be broken down into several stages. First, the client that needs a service from a Web service builds a SOAP request. Due to the fact that SOAP specifies its wire format to be in XML, the building of the

SOAP envelope involves serialization of binary values into ASCII text. The request may contain few parameters that the service needs to complete the request. The client then sends the request, which is now in the form of ASCII text onto the Internet using Hyper Text Transfer Protocol (HTTP). 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].

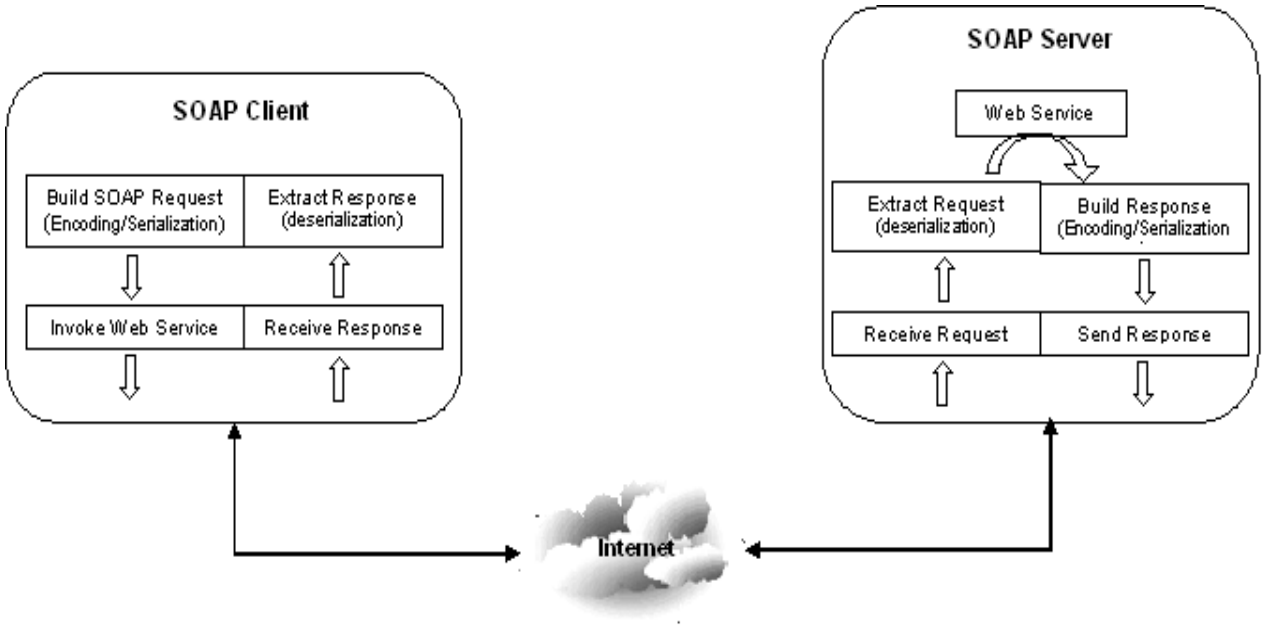

Figure 1. Web services invocation model

The SOAP server (or engine), which receives this request will de-serialize it and forwards it to the Web service to which it is addressed. The architecture of the SOAP engine is explained in detail in the later part of this section. The return value from the Web service is then used to build a SOAP response. This again involves serialization of

binary values into XML format, to meet the SOAP specification. After the data is encoded into XML format, it is sent back to the client over HTTP. The client, upon receiving the response, de-serializes it and extracts the actual return value from the Web service.

The total time required for invoking a Web service can be broken depending on the different stages involved in the process of invocation of a Web service.

$$\mathbf{T} = \mathbf{T}_{buildRequest} + \mathbf{T}_{send} + \mathbf{T}_{extractRequest} + \mathbf{T}_{WebService} + \mathbf{T}_{buildResponse} + \mathbf{T}_{receive} + \mathbf{T}_{extractResponse}$$

Where,

$\mathbf{T}$ = Total time taken by a client for invoking a Web service and getting the response from it.

$\mathbf{T}_{buildRequest}$ = Time taken by the client to build the SOAP request.

$\mathbf{T}_{send}$ = Time taken for sending the client's request to the server.

$\mathbf{T}_{extractRequest}$ = Time taken by the SOAP engine to parse the SOAP envelope and extract the parameters and make a method call to the Web service.

$\mathbf{T}_{WebService}$ = Time taken by the Web service to process the request.

$\mathbf{T}_{buildResponse}$ = Time taken by the SOAP engine to build the SOAP response with the values returned by the Web service.

$\mathbf{T}_{receive}$ = Time taken by the response to reach the client.

$\mathbf{T}_{extractResponse}$ = Time taken by the client to parse the response (de-serialize) and extract the actual response from the Web service.

In this research, we are not concerned with $\mathbf{T}_{send}$ and $\mathbf{T}_{receive}$, which are the times taken for the messages to transfer from the client to the server and vice versa, via the Internet. The optimization of SOAP protocol is divided into two stages – client-side and server-side. This section explains the implementation carried out on both sides. The client-side optimization details the analysis performed on the computation that is taking place on the client-side and the approach used to optimize it. Two different techniques of optimization are implemented, which have been evaluated later. The implementation on the server-side starts with the understanding of the architecture of the SOAP engine. This helps in breaking the sequence of flow of operation on the server-side, to include a caching logic.

3.2 Client-Side Optimization

3.2.1 Profiling a SOAP RPC Client

The previous subsection described the process of invocation of a Web service. It also lists the various stages of client-side processing taking place. In our implementation, we have chosen the Apache [8] implementation of SOAP version 2.3. Also, we chose 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 starts with an analysis of the SOAP requests generated by the client when it invokes a Web service. This involves profiling of a SOAP RPC client. The profiler that is chosen is Hpjmeter. To study the client-side processing, a simple Web service was deployed on Tomcat 5.0 web container, which does nothing and returns nothing to the

client, when it requests its service. The client, on the other hand, makes a method-call, *doNothing()*, to the Web service. The SOAP RPC client was running on a 360MHz, 128MB RAM, Ultra SPARC –III system on SunOS 5.9 version platform. This puts $\mathbf{T}_{buildRequest}$, $\mathbf{T}_{send}$, $\mathbf{T}_{extractRequest}$, $\mathbf{T}_{WebService}$, $\mathbf{T}_{buildResponse}$, $\mathbf{T}_{receive}$ and $\mathbf{T}_{extractResponse}$ to the minimum. Our concern mainly is in $\mathbf{T}_{buildRequest}$, which is the time taken by the client to build the SOAP request.

The client is run to invoke the Web service and the profile data generated is collected. The client takes a total of 47 seconds to complete its invocation. The collected profile data is then used to investigate the different stages of execution of the client, using a profiler. Hpjmeter displays the CPU utilization of the client using a call-graph tree as seen in Fig.2.

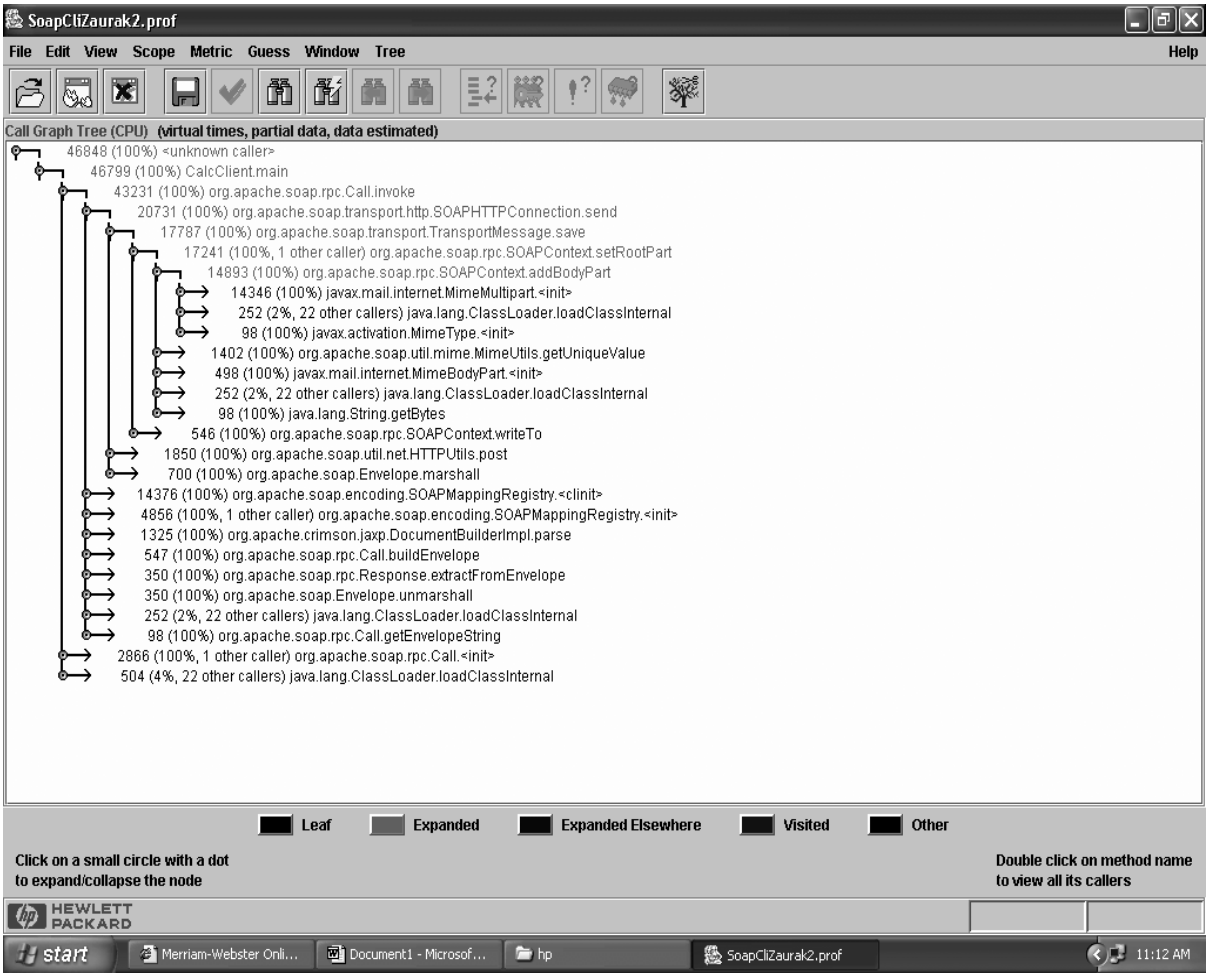

Figure 2. Profile of a SOAP RPC client

Fig. 2 also lists the different methods called in the process, together with the time taken by each method-call in milliseconds. This analysis determines the major stages of client-side processing which take-up maximum time of execution.

Each of the stages is further examined to find out the key areas where the client is spending most of its time. From the information obtained by profiling a simple SOAP client, we can easily make out how much percentage of total run-time does the client spend in serialization and de-serialization. It is observed that the SOAP client spends

around 18 seconds in building the SOAP envelope, which also includes establishment of HTTP connection to POST the SOAP messages on to the wire. This accounts to around 38% of the total time of execution of the client. The building of envelope involves creation of header, body and other tags. The other significant portion of the execution time is taken in serialization. The SOAP client took 20 seconds in serializing the binary information into XML format. This time also involves the time spent for a look-up performed on the SOAP mapping registry, which lists the serializers and de-serializers that can be used for respective classes. The total execution time also includes 3 to 4 seconds of marshalling and un-marshalling. The time taken for the message in transit and the time taken by the SOAP engine to service the request together will constitute the rest of the total time.

The result of the analysis of the client-side processing of a SOAP request testifies the amount of time spent by the client on 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 to simply construct a query string requesting the stock-quote value. In such a scenario, XML encoding proves critical and will have a major effect on performance of the client.

3.2.2 Solution

The latency caused because of the XML encoding and serialization has a major effect on the efficiency of a SOAP client. This in turn affects the total performance of the Web

service. It is observed that Web services are used in scenarios where a client sends the same request to the Web service over and over again. This obviously results in a call being made to the same Web service, by the SOAP engine. Sending of the same request message involves creation of the same SOAP payload. A close look at the client-side execution indicated that the creation of a SOAP payload has got the associated latency due to the encoding costs. Upon identifying that every client application has a finite set of different requests that are sent to the server over time, we can think of the solution, which avoids repeated generation of the same request.

One such example is a Time-of-Day Web service. The SOAP RPC client sends the same request over and over again, basically requesting the time of the day. For each such request, the time spent in XML encoding is critical to the performance of the client because of the meager amount of processing involved at the server. This scenario is very common in Web services. Saving the SOAP payload that the client generates the first time it makes a call to the Web service and re-using it for future calls, can optimize the performance of the client to a great extent. This saves the time spent in encoding and serialization every time a call is made to the Web service. This solution of caching the SOAP requests at the client-side is discussed in detail in the next section. A variant of this technique, "*Parameterized Caching*", which has better performance, is also explained.

3.2.3 Total Caching

Consider the Time-of-Day Web service that we talked about in the last subsection. As said before, we chose the RPC-style model. One rationale behind this choice is one of the SOAP's design goals, which is to encapsulate and exchange RPC calls. A typical SOAP RPC client that uses this Web service will make a method call to the Web service with no input parameters supplied. The snippet of code required to invoke such a Web service is shown in Fig. 3.

```
Line 1:   Call call = new Call();

Line 2:   call.setTargetObjectURI("urn:TimeService");
Line 3:   call.setMethodName("getTime");
Line 4:   call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);

Line 5:   Response resp = call.invoke(url, "" );
```

Figure 3. SOAP RPC client code invoking urn:TimeService Web service

Fig. 3 shows the SOAP RPC elements that are required to make a Web service invocation. Line 1 shows the creation of an instance of org.apache.soap.rpc.Call, a Java class that encapsulates a SOAP RPC method call. This method call requires some attributes to be set, before invoking the Web service. Line 2 shows how a Target Object URI is set for the call object. Target Object URI is, essentially, the resource address of the service that we want to use. It is used by the SOAP engine to locate the targeted Web service among all other services that are deployed on it. The name of the method that is

supposed to be called is set in line 3. The setting up the parameters requires line 4, where in, we set the encoding style to be used to serialize the parameters. In this case, the encoding style is given by Constants.NS_URI_SOAP_ENC, which is the standard SOAP encoding namespace. For our simple client, which doesn't provide any input parameters for the service, this line of code is useless. Finally, the SOAP request is sent to the Web service by calling invoke() of the Call object at line 5. The invoke() returns the Response object, from which we extract the actual return value of the Web service. Fig. 4 shows the SOAP payload that the client generates when making call to the Web service.

```
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>
```

Figure 4. SOAP payload generated by a simple SOAP RPC client.

This message is very large in size when compared to a similar request of a Java RMI client. Every time this client wants to invoke the Web service, it has to build this payload, which involves encoding XML. We have already estimated the time spent in

serialization by such a client. There is a better way to handle such multiple requests from the client. This is where the notion of caching SOAP payload comes up.

Every client application has a finite set of different requests that are sent to the server over time. As discussed before, we can cache the requests and use them later. Consider an example of a stock quote requesting application. This application is a little different from the Time-of-Day service, in that, it makes similar requests rather than identical requests to the server. This is because the client sends an input parameter when invoking the Web service. This generates a SOAP payload that is a little different from the payload generated for a different input parameter. The technique implemented in this report is to cache 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 as this causes heavy latency because building of a SOAP payload takes a significant amount of processing time. Fig. 5 shows a SOAP client-server architecture in which, the client implements such a caching mechanism.

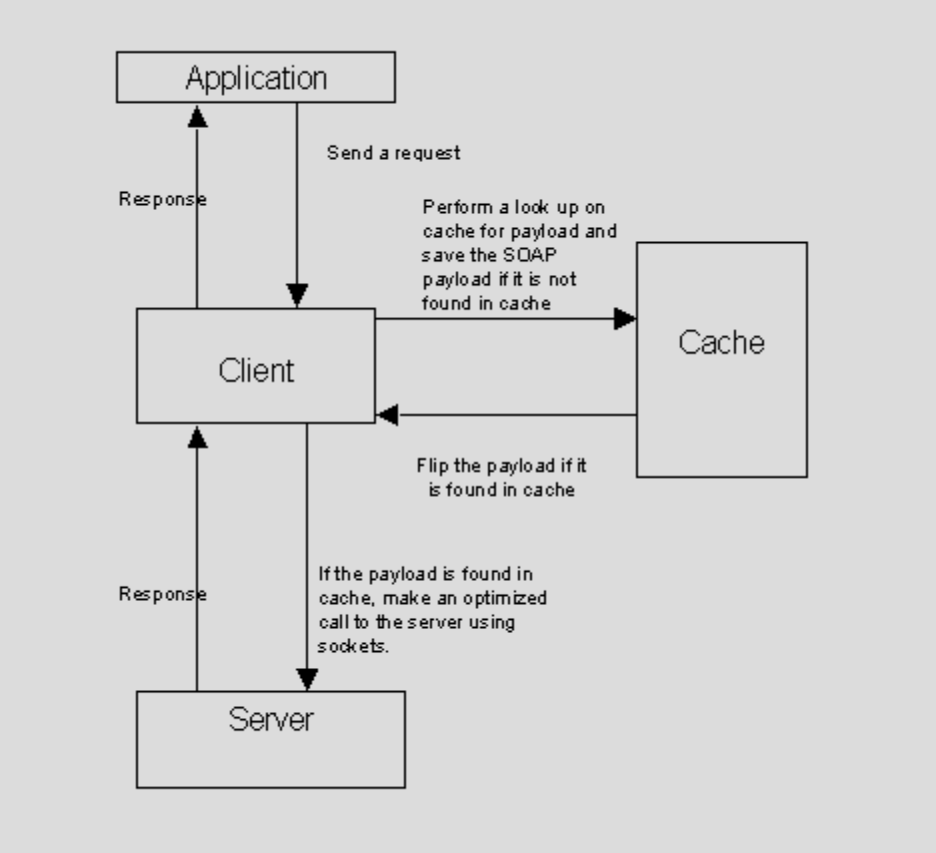

Figure 5. SOAP RPC client–server using client side caching.

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.

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 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 previously done by creating an instance of org.apache.soap.rpc.Response.

3.2.4 Parameterized Caching

The total caching technique implemented on an SOAP RPC client increased its performance to a great extent. The client works much faster when it wants to repeatedly invoke the Web service by using the cache. The only additional computation done when this strategy is used is the File I/O. The size of the cache increases with the number of different payloads the client generates for each type of request it sends to the Web service. The performance increase obtained with the total caching outweighs the latency due to XML encoding when the size of the cache is small. But if the number of different types of requests is more, the size of the cache gets larger, ultimately increasing the computation time for File I/O. This will have an overall impact on the performance.

Let us considered a scenario in which the client invokes a Web service repeatedly with requests that differ only by the values of a few XML tags in the SOAP payload generated. For instance, consider a Web service that provides flight information. The SOAP RPC client requests flight information between two cities by providing the city names as parameters to the server. Fig. 6 shows the code required to invoke a Web

```
Line 1:   Call call = new Call();
Line 2:   call.setTargetObjectURI("urn:FlightInfoService");
Line 3:   call.setMethodName("getFlightInfo");
Line 4:   call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);

Line 5:   Vector params = new Vector();
Line 6:   params.addElement(new Parameter("From", String.class, from, null));
Line 7:   params.addElement(new Parameter("To", String.class, to, null));
Line 8:   call.setParams(params);
Line 9:   Response resp = call.invoke(url, "" );
```

Figure 6. SOAP RPC client code invoking urn:FlightInfoService Web service

service by sending parameters to it. To send a request to the server, the SOAP RPC client creates an instance of org.apache.soap.rpc.Call in Line 1. After specifying the name of the service and the method being invoked in line 2 and line 3 respectively, we set the parameters, which in this case are the names of the two cities, using the setParam() method of the Call object in line 8. The Web service is invoked in line 9. Fig. 7 shows the SOAP payload that the client generates. This payload differs from the payload we have seen in Fig. 4. This payload has got <From> and <To> as additional tags to carry the supplied parameters. These additional tags store the source and destination cities in their value attribute.

```
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 7. SOAP payload generated by a SOAP RPC client.

Looking at such a Web service and 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. The solution of using *total caching* technique does improve the efficiency of the client. But the increase in the size of the cache hinders the performance due the time spent in File I/O computation. From above observation, we discover that there is better ways to handle such requests rather than caching the total payload even when just a few tags differ between different requests. This is where the idea of partial caching or parameterized caching 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. *Parameterized 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 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. The client identifies the elements for which the values are to be replaced, by performing a string search on the payload for *parameter-name*. The *parameter-name* for a parameter is set by the client when it creates org.apache.soap.rpc.Parameter object for that input parameter. In Fig. 6, for example, line 6 indicates the *parameter-name* as "From" for that input parameter. So, the client has to search for the <From element in the payload to replace the value with new parameter. This limits the size of the cache, as requests differing in the values of few tags are not cached. This decreases the time spent in file I/O. The client then establishes a socket connection, like in *Total caching* strategy, at the port where the Web service is deployed on that host.

3.3 Server-Side Optimization

After having optimized the client using different techniques, the performance of the SOAP RPC client improved manifold times. When looking from the point of view of the a server, it still has to open the SOAP request envelope and build a SOAP response after making a method call to the Web service that the user implements. Just like on the client-side, this process takes a lot of computation time as it involves XML encoding. The aim

of this report is an overall optimization of the SOAP protocol. This section first explains the architecture of the SOAP engine and the way caching mechanism is added to it.

3.3.1 Architecture of Apache SOAP Engine

The work done of the client-side shows that caching proves itself as a good solution for the latency associated with a SOAP call. To employ similar techniques on the server side, we should first understand its architecture. In our implementations, we used the Java implementation of Apache SOAP 2.3. Fig. 8 shows the architecture of the

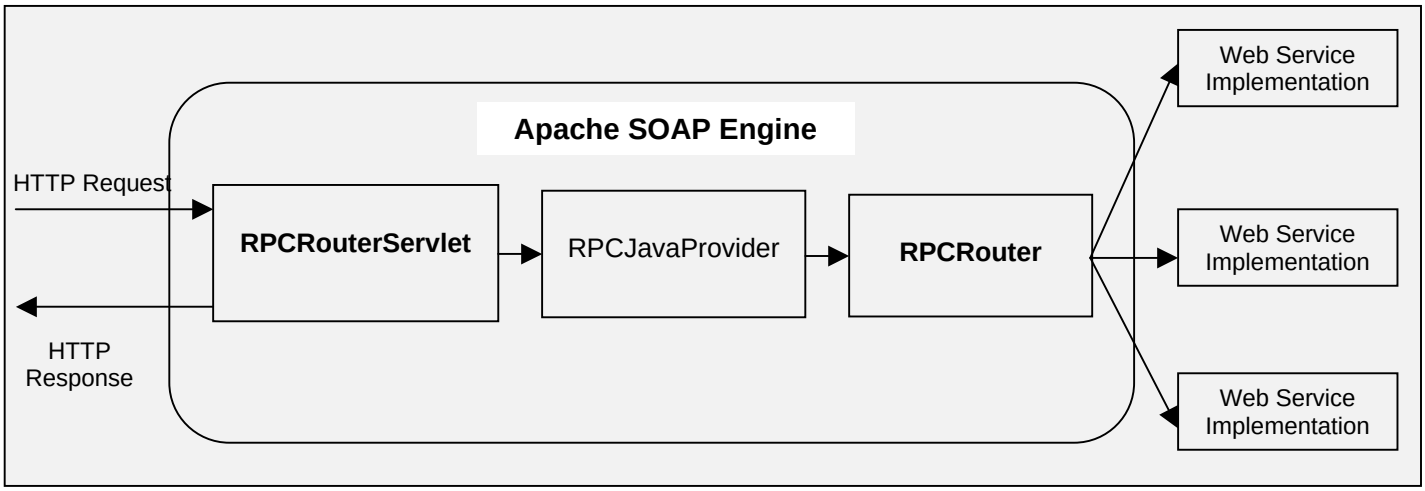

Figure 8. Architecture of Apache SOAP Engine

Apache SOAP engine (or processor). Apache SOAP engine basically consists of three components namely RPCRouterServlet, a Provider and RPCRouter. Each of these components has a unique functionality in the process of servicing a SOAP request. The functionality of each of the components is explained below.

RPCRouterServlet: RPCRouterServlet component extends the functionality of HttpServlet. This component receives the HTTP request from the client and returns a HTTP response. It is responsible for extracting the SOAP envelope from the HttpServletRequest. From the envelope, it extracts the Call object with the help of RPCRouter component. It is also responsible for determining the appropriate "Provider" to process the request.

RPCJavaProvider: RPCJavaProvider is the default Provider for Java implementations. The "Provider" is one of the most important components of the SOAP engine and is responsible for locating the Web service implementation. It makes use of RPCRouter component to make the actual method call to the Web service implementation. The value returned by RPCRouter is used to build the SOAP envelope, which is later sent to the client. The Provider also determines the actual Web service to which the request is targeted.

RPCRouter: The RPCRouter extracts the information required to make the call from the envelope. It also determines whether the call being made is valid or not. The main functionality of this component, however, is to actually perform the method-call to the Web service implementation. It makes the call to the target method by extracting the method name and input parameters from the envelope. The return value of the method of Web service is supplied to the RPCJavaProvider to build the response.

The SOAP engine abstracts the invocation of a Web service from the point of view of the client. As mentioned before, the RPCJavaProvider is the component, which is responsible for building the SOAP envelope. This indicates that the provider performs the serialization before the response is wired to the client. Considering a simple Web service implementation with a method *doNothing()*, the maximum time it takes once the request reaches the SOAP engine, is the time spent by RPCJavaProvider which does the XML encoding. So, for a Web service that provides weather information of a day, the return value will be the same, at least for that particular day. Such implementations perform serialization of the response every time the Web service receives a request. This can be avoided by using the caching strategy on the server-side. The placement of the cache in the SOAP engine is critical to the extensibility and generality of the implementation.

3.3.2 SOAP Engine with Caching

After careful understanding of the architecture of the Apache SOAP engine and considering several different design issues, it was decided to add the caching logic to the Provider component of the Apache SOAP engine. Fig. 9 shows the modified Apache SOAP framework, which has now got the caching mechanism added to the Provider component. The RPCJavaProvider as said before, is responsible for building the SOAP envelope with the return value it receives from the RPCRouter component. RPCRouter is the component, which makes the actual method call onto the Web service, by supplying the input parameters.

RPCJavaProvider is modified in such a way that, before it is required to invoke the RPCRouter component for making the method-call to the Web service, it first unwinds the SOAP envelope and extracts the input parameters supplied (if any). Using these parameters as search keys, RPCJavaProvider performs a look-up on the Cache to see if there is a "Response" that is already cached corresponding to the parameters list. If there is one, it flips the "Response" object from the Cache, breaks the call there, bypassing RPCRouter component and sends the "Response" back to the client on HTTP via RPCRouterServlet. If the look-up ends up not finding a Response, then the call proceeds in the usual way, through the RPCRouter component.

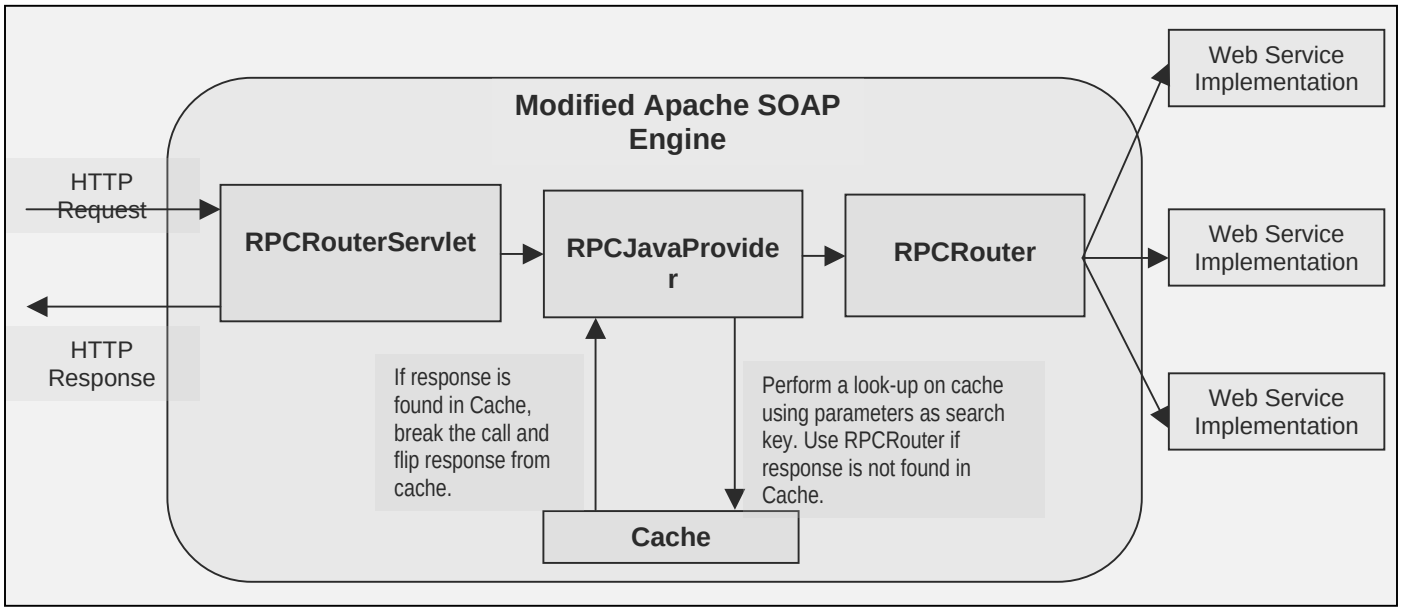

Figure 9. Architecture of modified Apache SOAP Engine with caching mechanism.

The "Response" from the RPCRouter is cached before it is sent back to the client, for future use.

The Cache component placed in the SOAP engine uses HashTables for high-speed access of items and look-ups there by avoiding the file I/O computation that was done on the client-side. The overall performance of the SOAP framework increases, for repeated requests, as the serialization costs encountered by the RPCJavaProvider are avoided by using the cache. Also, by bypassing the RPCRouter component, we can save the time spent in making the method call to the Web service.

3.3.3 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. A better caching mechanism is suggested as countermeasure.

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 look-up. 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 *parameterized 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 *parameterized 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.

The usage of cache on the server-side is restricted by the functionality of the Web service deployed. For example, consider a Web service providing weather information of a requested city. Using a cache to store the Responses for different requests works fine as long as the information in the cache is not stale. Several characteristics like staleness, time-to-live have to be considered and features such as flushing the cache after sometime should be enabled for correct operation of this technique.

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 (or engine) 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 re-cached.

3.4 Technologies Used

This section lists the technologies used for this project and lists some reasons explaining why such a choice was made.

Java™ 2 SDK, Standard Edition

The Java 2 SDK is the obvious choice for a development environment for building platform-independent client-server models using the SOAP protocol. Java™ version 1.4.2 was used for all the implementations.

SOAP Engine

The SOAP engine generates and parses requests and responses. The Java™ implementation of Apache SOAP version 2.3 was used both on the server and the client sides. This choice was obvious after comparing efficiencies of other implementation of the SOAP protocol. Also, being Java based, platform-independence is easily met for all the modifications made to the SOAP engine to add the caching logic.

XML parser

Apache SOAP 2.3 requires a namespace aware XML parser to parse XML messages. Apache Xerces 1.4.4 is a JAXP compatible namespace aware XML parser that was chosen.

Servlet Engine

Apache Tomcat 5.0 is used as a servlet engine, which is responsible for creating and starting Java Virtual Machine. It provides a container for holding and starting the SOAP engine. It also comes with a built-in Web server to forward all the HTTP traffic to the SOAP engine.

Java™ 2 SDK, Enterprise Edition

All the implementations on the server-side need enterprise version of the Java™ SDK. J2EE version 1.3.1 is used for this project.

JavaMail 1.3.1

The JavaMail package is for the Simple Mail Transfer Protocol (SMTP) support included in Apache SOAP. Apache SOAP uses it in the messaging component of its implementation.

JavaBeans Activation Framework

The JavaBeans Activation Framework or simply JAF is used by Apache SOAP to invoke software extensions that can handle non-textual data like jpegs or bitmaps.

Profiler

The profiler used to profile the SOAP RPC client is Hpjmeter version 1.5.3.

4 Evaluation

This section first lists a series of experiments that were run to compare SOAP with Java RMI, a binary protocol. These tests enabled us to compare the performance of SOAP with Java RMI and, focus on the stages of client-side processing of SOAP that slow-down its execution speed. 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 total 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 *parameterized caching* mechanism at the client side.

To optimize the SOAP server, the SOAP processor was modified to enable parameterized caching that is similar to the one implemented on 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 1.4.2 to test these applications on an Apache Tomcat 5.0 web server. Xerces was used as the XML parser for Apache SOAP 2.3. These applications were tested on SunOS 5.9 running on a 750

MHz, 2GB RAM Sun Blade 1000 system. Fig. 10 gives the performance comparison of SOAP, with total client-side caching, Java RMI and traditional SOAP. The performance of Java RMI is far better than that of SOAP, and this is evident from the Fig. 10. 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

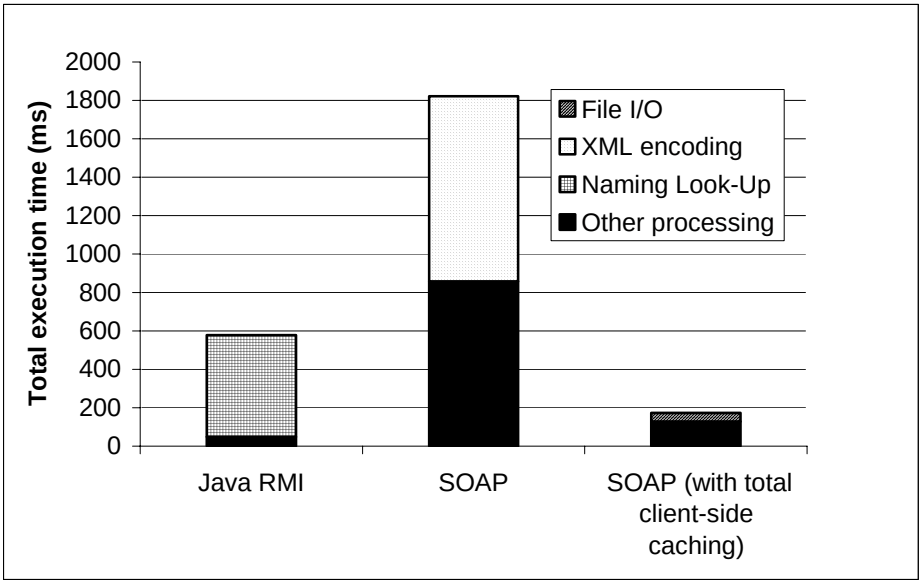

Figure 10. Comparison of SOAP (with total client-side caching) with Java RMI and the traditional SOAP

client, which now has a caching mechanism. Its performance is compared with both the traditional SOAP and Java RMI in Fig. 10.

As the caching mechanism is implemented using files, there is an additional computation involving file I/O, which replaces 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.

The total client-side caching pushes the performance of this client, making it works 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 *parameterized 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. 11. The graph shows that the performance of SOAP degrades as the size of the request increases. We see that SOAP with *parameterized caching* is more efficient than SOAP with total 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 *parameterized*

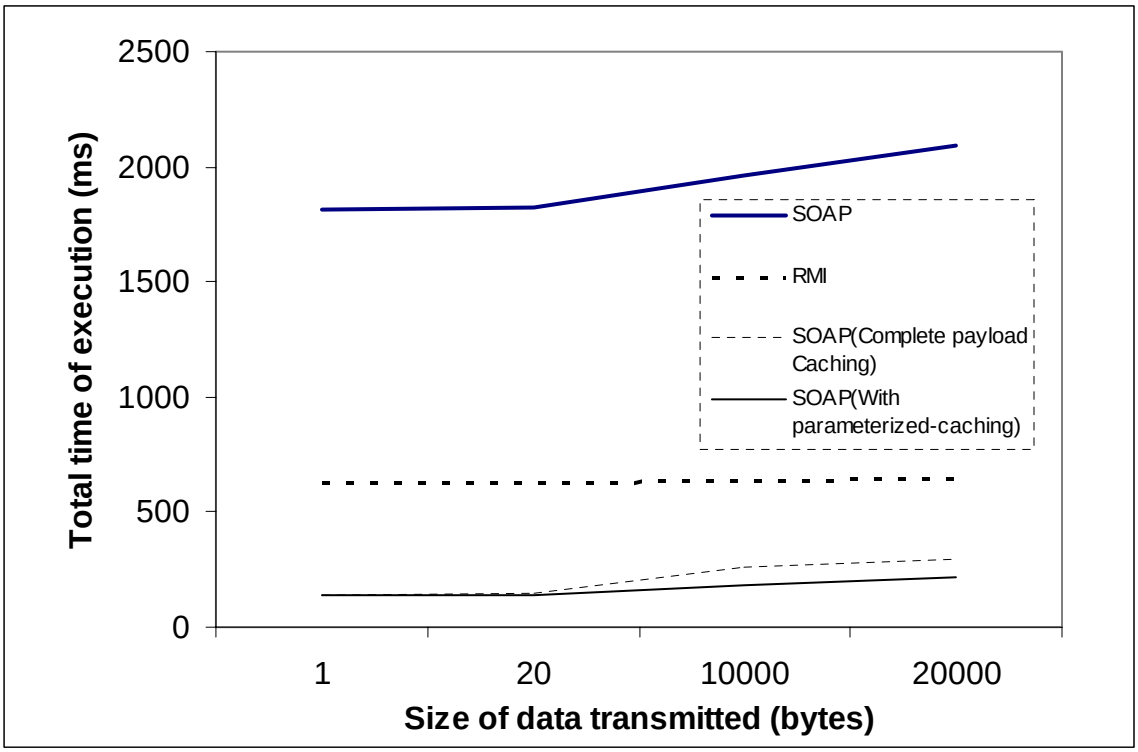

Figure 11. Effect of size of data on performance

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, independent of the parameters supplied to the server. This limits the time spent for file I/O computation.

Similar testing was done on the server side after adding caching logic to the SOAP engine. Testing was done with the sample Web services deployed on a Tomcat 5.0 Web container running on a 2.66MHz, 2GB RAM Linux box. The client applications invoking the above Web services were run on a 933MHz, 526MB RAM, Pentium III machine with Linux platform on it.

The server with an optimized SOAP engine performed more efficiently than the ordinary Apache server. The overhead due to caching was negligible compared to the

overhead of caching implemented on the client-side. This is due to the fact that, server-side caching was performed using HashTables rather than files. To evaluate the performance of optimized Apache SOAP, numerous test cases were coded, which test the impact of increasing sizes of request and response on client and server respectively.

First, the effect of request size on performance of the application was studied by coding several test cases, ranging from a client sending 0 bytes to 15KB of data to the Web service. The Web service does not return any huge response to the client. The output of this testing is shown in fig 12.

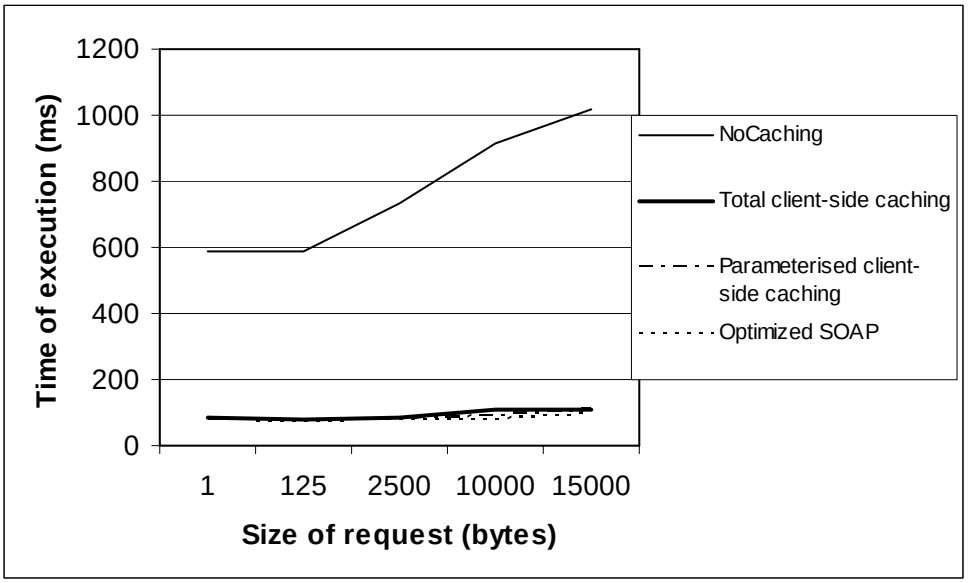

Figure 12. Effect of request size on performance

Fig. 12 shows that the performance of the optimized SOAP, which is Apache SOAP 2.3 with parameterized caching on both client and server sides, is much better than the performance of "Total client-side caching" and "Parameterized client-side caching". This is the result of saving the time spent in serialization at both ends of the client-server

model. Also, the increasing request size did not hinder the performance of parameterized client-side caching because of the small cache size.

5 Conclusion and Future Work

This report addresses the efficiency of the Simple Object Access Protocol and suggests techniques to improve its performance by using parameterized caching. This report also demonstrates this idea and evaluates various effects on this technique by experimental testing of numerous scenarios. From the results achieved, we can say that, caching of SOAP requests and responses to avoid serialization might be a good idea to improve the latency of application using SOAP. However, the techniques suggested do not provide a cure-all solution to every Web service running. Every Web service will have its own quality of service parameters that need to by taken care off while employing such optimization strategies. Nevertheless, these experiments imply the performance boost that is possible using these strategies. However, several important issues still remain open for further research. A caching mechanism, which is more optimal, can further aid in achieving better efficiency. The indexing of cache contents is one other area that needs refinement. Apache comes up with AXIS, which is the next version of SOAP implementation from Apache. AXIS provides a feature called as pluggable providers for customized provider functionality that the legacy SOAP doesn't provide. Future work includes usage of such pluggable providers to support caching. These advancements in SOAP can make it a good choice for not just high performance Web services but also for supercomputing.

6 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

[11] World Wide Web Consortium, http://www.w3.org/

[12] Common Object Request Broker Architecture, http://www.corba.org/