Building High-speed Roads: Improving Performance of SOAP Processing for Cloud Services

Huang Liu, Xudong Liu, Jianxin Li, Yongwang Zhao, Zhuqing Li
Department of Computer Science
Beihang University
Beijing, China
{liuhuang, liuxd, lijx, zhaoyw, lizq}@act.buaa.edu.cn

***Abstract*—With the advent of cloud computing, more services should be delivered from a cloud or datacenter to clients via Internet. Web Services has many mature specifications and has become a popular mode to support cloud application. However, the performance has been a key character to guarantee the quality of remote service in a cloud. In particular, the Simple Object Access Protocol (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. Experiments have shown that the XML parsing, formatting and type mapping are the primary performance bottlenecks of SOAP processing. In this paper, we proposed a new high-performance SOAP processing architecture named HiSOAP which is based on several techniques such as on-demand building, delayed-databinding and adaptable services invoking mechanism. HiSOAP has the benefits of reducing XML object model, saving memory and eliminating Java reflection. We have implemented the SOAP engine HiSOAPprototype, and the experimental results show that the delayed-databinding and eliminated-reflection techniques improve the performance of SOAP processing dramatically, and it out-performances Axis2, especially in the complex compound data types of Web Service.**

Keywords—Web Services; SOAP; high-performance; optimization

I. INTRODUCTION

Currently, many international companies have launched their own cloud computing products, such as Microsoft Windows Azure, IBM Blue Cloud, Amazon Elastic Compute Cloud, etc. With the advent of cloud computing, more services should be delivered from a cloud or datacenter to clients via Internet, and this injects new vitality into Web Services which has many mature specifications and has become a popular mode to support cloud application. At the same time, the cloud computing puts forward higher requirements of Web Services, including less response time, low memory usage, higher data transmission rate, etc. However, the performance has been a key factor to guarantee the quality of remote service in a cloud. In particular, the SOAP protocol has emerged as a mature Web Services communication standard, providing extensibility, language and platform independence, and interoperability. However, SOAP inherits not only the advantages of XML, but its relatively poor performance. This makes SOAP a poor choice for many high-performance Web Services. We know that communication protocol is the key of distributed computing model. Therefore, studying and improving the performance of SOAP processing is still very important to cloud computing.

F. Bustamante pointed out that the performance of distributed systems is strongly determined by their `wire format', or how they represent data for transmission in a heterogeneous environment [1]. The performance of the XML-based Web Services is much poorer than that of the other binary-based distributed computing models. Comparing with these models (such as CORBA, Java RMI, etc) [2][3], the results show that there are two primary factors that affect the performance of Web Services: (1) XML parsing and formatting, (2) Network communication of Web Services.

In this paper, we focus on the performance improvement of SOAP processing which includes XML parsing, formatting, type mapping (marshalling and unmarshalling), etc. We designed an experiment to analyze the time cost of main stages of SOAP processing in the problems analysis section. It is concluded that the XML parsing, formatting and type mapping are the primary performance bottlenecks of SOAP processing.

How to improve the performance of SOAP processing? One approach is to use or implement a specific high-performance XML parser for SOAP message. Because the SOAP message is special XML Infoset, so the specific SOAP message parser is much faster than a general XML parser. The other approach is to accelerate the type mapping which is used to convert between XML data types and Java data types. Generally, we first convert the XML bytes stream to XML object model, and then convert the XML object model to Java data types (primitive data types or JavaBean). If we directly convert the XML bytes stream to Java data types, the conversion will avoid creating XML object model and traversing XML tree. On the Java platform, the type mapping of compound data type relates to Java reflection. Because the compound data type is represented by JavaBean, and the SOAP engine doesn’t know the details of JavaBean in compiling time, so it must use the Java reflection to set or get the value of JavaBean.

To address the above issues, we proposed a new high-performance SOAP processing architecture named HiSOAP, and implemented the SOAP engine HiSOAPprototype with this architecture. The major contributions are as follows:

- We have analyzed the main stages of SOAP processing and concluded the performance bottlenecks of SOAP processing by experiment.

- A new specific SOAP parser delayed-parser: This parser is based on on-demand building technique which makes the users to control the parsing process, and it is transparent to the users.
- Delayed-databinding: This is a new specific databinding mechanism for SOAP message. This technique can reduce XML object model creation, reduce memory overhead, and avoid traversing XML tree. Moreover, it is compatible with the extended protocol processing.
- Eliminated-reflection: Using the customized service invoker, the adaptable service invoking mechanism can eliminate Java reflection and accelerate type mapping.
- A high-performance SOAP engine HiSOAPprototype: Experimental results show that our SOAP engine that adopts these performance optimization techniques out-performances Axis2, especially in the complex compound data types of Web Service.

The remainder of this paper is organized as follows. In Section Ⅱ, we review related approaches to improve the performance of Web Services. In Section Ⅲ, we state the problems of current SOAP processing architecture, and present our analysis on the performance costs of SOAP processing, and design a high-performance architecture named HiSOAP. We then discuss our implementation of the high-performance architecture in Section Ⅳ, which is followed by the results of the experiments we conducted to compare the performance of different HiSOAPprototypes with that of Axis2. Finally, we present our conclusions and future work in Section Ⅵ.

II. Related Work

A variety of approaches have been proposed to optimize the performance of Web Services. These studies focus on the performance analysis, data compression and transmission, message caching, efficient XML parsing and type mapping, etc.

The research in [4] pointed out that the most significant bottleneck is ASCII/double conversion in science computing environment; Kohlhoff thinks that the performance of Web Services can be improved by optimizing the SOAP encoding and decoding in business field [5]; researches in [7-9] shows that the XML-based SOAP protocol increases the amount of data transmission and reduces the efficiency of transmission. Using data compression algorithm, it can reduce the data size and improve the transmission efficiency. However, it is a trade-off between the performance overhead of data compression and the efficiency of transmission improved by it. All these approaches are only applied to different specific environments.

By caching request message at client [10], it can reduce the duplication of serialization operations and improve the performance of Web Services, and the other approach is caching response message at server [11]. Researches in [12-14] propose that each response message of Web Service has similarity. So it improves the performance of serialization by differential serialization which essentially is a caching mechanism. Experimental results show that the caching mechanism can improve the performance of Web Services, but it only applies to the scene when the Web Service does not change frequently.

XML parsing and type mapping are the key factors which affect the performance of Web Services [2-5] [19]. Researchers (Ng, etc) verified this conclusion by testing commercial SOAP engine. The type mapping is the performance bottleneck of Web Services [15], and the author proposes a dynamic template method to eliminate Java reflection to accelerate it. Another performance improvement is to design efficient XML parser [16-18].

Axis2 is based on Axiom (Axis Object Model) and early-databinding (JAXB, XMLBeans, JiBX, etc.), but its performance is not very good. Because the Axiom uses StAX(Streaming API for XML) as its on-demand building approach, and the performance of StAX is much poorer than that of XPP3(XML Pull Parser 3rd). Moreover, the StAX parser is referenced by all elements, and this design increases the memory overhead and reduces the performance.

III. Problems Analysis

A. Problem Statement

In commercial SOAP engine, as shown in Fig.1, a typically SOAP processing architecture includes four modules: Message Transmission, Message Processing, Extended Protocol Processing and Services Invoking.

The Message Transmission module is used to send and receive SOAP message. The transmission protocol can be HTTP/HTTPS, STMP, POP3, etc; the responsibility of Message Processing module is to parse, dispatch and format SOAP message; the Extended Protocol Processing module processes the extended WS-* protocol, including resource/notification, security, transaction, etc; the Services Invoking module includes message validation, unmarshalling and marshalling, etc. The function of unmarshalling is to convert the XML object model to Java data types. But the function of marshalling is contrary to unmarshalling. The Services Invoking is responsible for validating the request message, unmarshalling the parameters object from the request message, invoking the Web Service and marshalling the return value to response message. The ServiceImpl is a service package which contains deployment file, WSDL (Web Service Description Language) file and the implementation classes of Web Service, etc. When the SOAP engine starts up, the service package will be deployed according to the deployment file and then invoked by the service invoker.

In the general SOAP processing architecture, the XML parsing and marshaling will create an amount of XML object model. When we convert the XML bytes stream into XML object model, even using the most streamlined XML object model, the data size will be increased by 4 times. This architecture not only takes up a large amount of memory but also reduces the performance of SOAP processing. Currently, to avoid creating XML object model, the general optimization mechanism for the SOAP processing architecture is early-databinding, as shown in Fig.2. It directly unmarshals the parameters object from XML bytes stream and marshals the return value to XML bytes stream. On the Java platform, there are several early-databinding tools, such as JAXB, JiBX, XMLBeans, etc.

When the engine receives the request SOAP message, it does not parse the message as XML object model any longer, but it directly uses unmarshalling databinding to unmarshal the parameters object from XML bytes stream. When the engine constructs the response SOAP message, it does not create the XML object model of response message any longer. On the contrary, it directly uses marshalling databinding to marshal the return value to output stream. The databinding can avoid creating XML object model, save memory, and improve the performance of SOAP processing. Since the early-databinding does not create XML object model, so the Extended Protocol Processing module has no data to process. Therefore, this SOAP processing architecture based on early-databinding is only suitable for the common Web Service which does not contain extended protocol.

B. *Performance Analysis*

In order to optimize the SOAP processing, we designed an experiment to analyze the time cost of main stages of it and find out the primary performance bottlenecks of it.

Test Services: We design and implement four common services of Axis2, as shown in TABLE 1. In order to minimize the impact of service performing, they are implemented very simple. They are used to find out which stage is the primary performance bottleneck of SOAP processing for different data types of Web Service.

Test Environment: Hardware: 1000 Mbps Ethernet card, 4GB memory, Intel(R) Core(TM) 2 quad CPU Q9550 @2.83GHZ. Software: Windows server 2003, jdk 1.5.0, Axis2 1.5.1, eclipse 35.

Test Method: Firstly, we construct the request SOAP message of four services. Secondly, use the function of Axis2 to simulate the main stages of SOAP processing, including parsing request SOAP message, unmarshalling parameters object, invoking Web Service, marshalling return value and formatting response SOAP message. Lastly, compute the time cost of main stages.

Test Results: As TABLE.2 shown, in the *EchoAdd* and *EchoArray*, the XML parsing and formatting are the primary time cost of SOAP processing. The message size and complexity of *EchoArray* and *EchoBean* are similar, but the time cost of each stage is very different. The gaps of these two services are produced by Java reflection which is one of the primary costs of type mapping. In the *EchoBean* and *EchoBeanList*, the type mapping is the primary time cost. Additionally, the compound data type is more complex, the time cost of type mapping occupies greater proportion. As the message size increase, the time cost of XML parsing and formatting increases very slowly, but the time cost of type mapping increases very quickly. So, for the primitive data types or small-message, the XML parsing and formatting are the primary performance bottlenecks of SOAP processing; for the compound data types or large-message, the type mapping is the primary performance bottleneck.

TABLE 1. TEST SERVICES

Web Service	Message Size(Byte)	Parameters & Return Type	JavaBean (Number)
EchoAdd	239	int	0
EchoArray	530	int[]	0
EchoBean	535	JavaBean	1
EchoBeanList	1595	Complex JavaBean	12

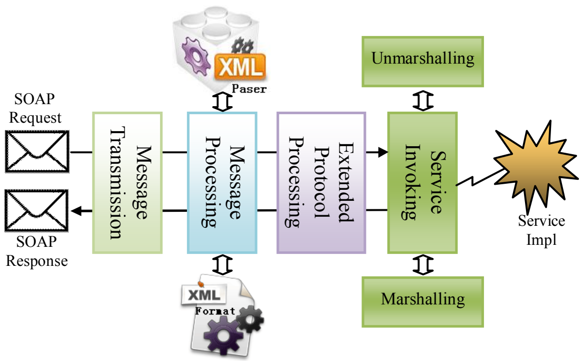

Figure.1 General SOAP processing architecture

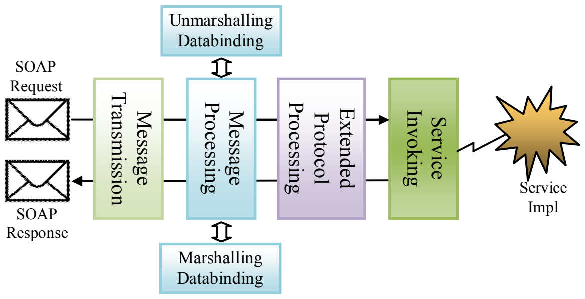

Figure.2 Based on early-databinding SOAP processing architecture

TABLE 2. TEST RESULTS

Web Service	Parsing	Unmarshalling	Marshalling	Formatting
EchoAdd	8ms	0.2 ms	0.4 ms	3.2 ms
	67.7%	1.6%	3.3%	27.1%
EchoArray	10.2 ms	0.8 ms	5.9 ms	3.5 ms
	50%	3.9%	28.9%	17.1%
EchoBean	9.6 ms	1.1 ms	17.2 ms	4.5 ms
	29.6%	3.3%	53%	13.8%
EchoBeanList	13.7 ms	5.5 ms	142.6 ms	9.4 ms
	8%	3.2%	83.2%	5.4%

C. *Time Cost*

The previous section has introduced and analyzed the SOAP processing. In this section we shall consider various steps involved in processing a SOAP message, associate cost functions to these steps, and illustrate the impact of possible optimizations based on the changes to the cost functions. These can be broken down into the following steps.

1) Parse the SOAP message as XML object model. The t_{parse} represents the XML parsing time; the $t_{p\text{-}create}$ represents the entire XML object model creating time; the $t_{p\text{-}create\text{-}h}$ represents the time of creating entire Header object model; the $t_{p\text{-}create\text{-}b}$ represents the time of creating entire Body object model. $t_{p\text{-}create} = t_{p\text{-}create\text{-}h} + t_{p\text{-}create\text{-}b}$.
2) Process the extended protocol (decryption, transaction, etc) or do nothing. Use the $t_{in\text{-}ext}$ to represent the processing time.
3) Unmarshal the object model to Java data types. The $t_{u\text{-}tra}$ represents the XML tree traversing time; the $t_{set\text{-}ref}$ represents the Java reflection time; the t_{xml2j} represents the type conversion time.
4) Marshal the Java data types to XML object model. The $t_{get\text{-}ref}$ represents the Java reflection time; the t_{j2xml} represents the type conversion time; the $t_{m\text{-}create}$ represents the XML object model creating time; the $t_{m\text{-}create\text{-}h}$ represents the time of creating entire Header

object model; the $t_{m\text{-}create\text{-}body}$ represents the time of creating entire Body object model. $t_{m\text{-}create} = t_{m\text{-}create\text{-}h} + t_{m\text{-}create\text{-}b}$.

5) Process the extended protocol (encryption, transaction, etc) or do nothing. Use the $t_{out\text{-}ext}$ to represent the processing time.
6) Format the XML object model to XML bytes stream. The $t_{f\text{-}tra}$ represents the XML tree traversing time; the t_{output} represents the XML bytes conversion time.

We ignore the time cost of identification of service invoker, the dispatch of the call to service invoker, the message validation and service performing. So, the total time of SOAP processing represented by t_{total} is following.

$$t_{total} = (t_{parse} + t_{p\text{-}create}) + t_{in\text{-}ext} + (t_{u\text{-}tra} + t_{set\text{-}ref} + t_{xml2j}) + (t_{m\text{-}create} + t_{get\text{-}ref} + t_{j2xml}) + t_{out\text{-}ext} + (t_{f\text{-}tra} + t_{output})$$

In the general SOAP processing architecture, Web Service is invoked by the fixed service invoker. Using this mechanism, the unmarshalling of parameters and the marshalling of return value are both implemented through Java reflection. As we know Java reflection is one of performance bottlenecks of type mapping, especially in the deep nested complex compound data types. For the primitive data types, the $t_{set\text{-}reflection}$ and $t_{get\text{-}reflection}$ are both essentially zero, but for the compound data types, $t_{set\text{-}reflection} + t_{get\text{-}reflection}$ will be the primary cost of type mapping. So, eliminating Java reflection will improve the performance of SOAP processing. We presume that the Java reflection has been eliminated in the following analysis.

There are three cases in SOAP processing. In the first case, there is no extended protocol need to be processed (common Web Service). If we use early-databinding, we can directly parse XML bytes stream to the Java data types and marshal the Java data types to XML bytes stream. This approach can avoid creating and traversing XML tree. In the second case, some extended protocols needs to be processed in the Extended Protocol Processing stage, but only the Header will be processed. The early-databinding is not suitable for this case, because the entire SOAP message will be parsed as XML object model. In the third case, when the engine processes the extended protocols, the Body will be processed. The entire SOAP message will be parsed as XML object model too. The time cost of each case is shown as follows.

The first case time cost is

$$t_{total} = t_{parse} + t_{xml2j} + t_{j2xml} + t_{output}$$

The second and third case time cost is

$$t_{total} = (t_{parse} + t_{p\text{-}create}) + t_{in\text{-}ext} + (t_{u\text{-}tra} + t_{xml2j}) + (t_{m\text{-}create} + t_{j2xml}) + t_{out\text{-}ext} + (t_{f\text{-}tra} + t_{output})$$

In fact, we can still use the databinding in the second case, because it is not necessary for Body to be parsed as XML object model. We can parse the Header as XML object model for Extended Protocol Processing module and use the remaining bytes stream (Body) for databinding. This databinding approach is named as delayed-databinding that can be used to optimize the first and second case. In the first and third case, the performance of delayed-databinding and early-databinding is the same. But in the second case, using delayed-databinding, the total time of SOAP processing will be

$$t_{total} = (t_{parse} + t_{p\text{-}create\text{-}h}) + t_{in\text{-}ext} + t_{xml2j} + (t_{m\text{-}create\text{-}h} + t_{j2xml}) + t_{out\text{-}ext} + t_{output}$$

Because of the $t_{p\text{-}create\text{-}h} < t_{p\text{-}create}$, $t_{m\text{-}create\text{-}h} < t_{m\text{-}create}$, $t_{output} < (t_{f\text{-}tra} + t_{output})$, $t_{u\text{-}tra} = t_{f\text{-}tra} = 0$, so the performance of delayed-databinding is better than that of early-databinding. Moreover, the delayed-databinding is compatible with the extended protocol processing.

From the time cost analysis, we know that eliminating Java reflection and using delayed-databinding will improve the performance of SOAP processing. If we want to implement the function of delayed-databinding, the XML or SOAP parser must support XML pull parsing technique that we can decide which part of SOAP message is parsed as XML object model and which part is used to databinding. This parsing technique is named as on-demand building which is basis of delayed-databinding. Therefore, if we want to implement all the above optimization mechanisms, we need a new SOAP processing architecture.

D. Architecture Analysis

The general SOAP processing architecture is very simple and flexible, but causes to copy large amounts of data, not only takes up a lot of memory, but also reduces performance. Currently, the early-databinding is only suitable for common Web Service and not compatible with the extended protocol processing. This databinding mechanism only optimizes the first case not the others. Thus, we propose a new high-performance SOAP processing architecture named HiSOAP which is based on several techniques such as on-demand building, delayed-databinding and adaptable services invoking mechanism, as shown in Fig.3.

In the Message Processing stage, the XML delayed parser just only parses the Envelope element, and the others (Header, Body, etc) still preserve in the buffer as bytes stream. In the Extended Protocol Processing stage, it parses the Header element if presents. If the extended protocol indicates that the Body needs to be processed, then the Body will be parsed. In the Services Invoking stage, if the immediate children of Body have not been parsed, the service invoker can directly get the remaining bytes stream of request message from the parser, and then the unmarshalling databinding unmarshals the bytes stream to parameters object, otherwise, unmarshals the XML object model to parameters object.

After the Web Service has been invoked, the engine does not immediately marshal the return value to response message. In the Extended Protocol Processing stage, if it needs to process the Body, the marshalling databinding marshals the return value to XML object model. In the Message Processing stage, if the return value has been marshaled, then the marshalling databinding formats the XML object model to bytes stream, otherwise, formats the return value to bytes stream directly.

To eliminate Java reflection, the SOAP engine opens the interface of service invoker and the function of unmarshalling and marshalling databinding to Web Service. So the service implementation can construct a customized service invoker itself. When invoking Web Service, this mechanism can avoid using Java reflection.

Base on the above analysis, we conclude that the HiSOAP can process the databinding and extended protocol processing concurrently, and eliminate Java reflection, and improve the performance of SOAP processing.

IV. A HIGH-PERFORMANCE SOAP ENGINE

According to the previous section analysis, we design and implement a SOAP engine named HiSOAPprototype that adopts the proposed high-performance architecture HiSOAP. Except the SOAP processing, the engine contains service deployer, service register, WSDL parser, etc. The service deployer is responsible for analyzing deployment file, loading service invoker and class, registering the service to service register, etc. The WSDL parser parses WSDL file as service model. In this section, we just give the implementation of On-demand building, delayed-databinding and adaptable services invoking mechanism.

A. On-demand Building

The core idea of on-demand building is to parse XML based on the user's request. The core technique of on-demand building is the pull parsing technique (such as StAX or XPP3). Because the advantage of pull parsing is that the user can control the parsing process. Moreover, the performance of XPP3 is better than that of StAX, so the XPP3 is widely used in SOAP parser. Generally, the XML parser parses the entire XML file and creates XML object model, but using the delayed-parser, the XML object model controls the processing of XML parsing and drives the parser to create itself.

The on-demand building is the basis of delayed-databinding which needs to parse the Header as object model and preserves the Body as bytes stream. Therefore, based on the characteristics of SOAP message format and the order of SOAP processing, we divide the SOAP elements into delayed-element and non-delayed-element. The Envelope, Header, Header Block, Body and Body Block are delayed-element. The others are non-delayed-element. In order to indicate whether the element has been parsed completely, there is a finish symbol in the delayed-element. The immediate children of each delayed-element are not parsed automatically until the user requests, but the no-delayed-element will be.

As shown in Fig.4, if an element wants to have the delayed function, it has to preserve a reference of the pull parser. When the user traverses the XML tree, the current element checks whether the request element has been parsed. If not, the current element calls the parser to parse and create the request element as XML object model. All these operations are transparent to the user. Another important function is that each delayed-element can directly get the instance of pull parser for the delayed-databinding.

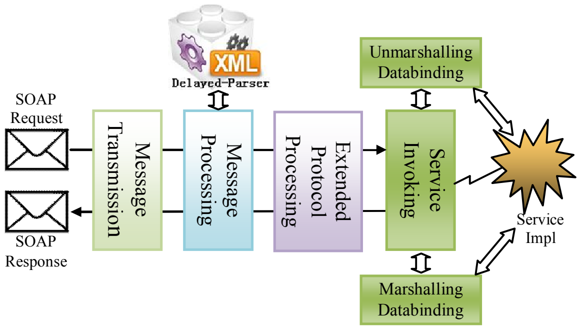

Figure.3 High-performance SOAP processing architecture HiSOAP

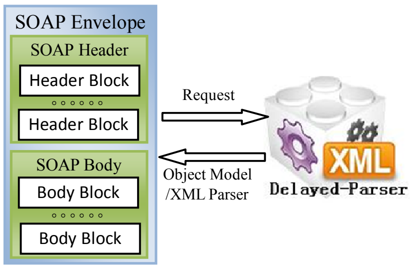

Figure.4 On-demand building

B. Delayed-databinding

There are the two differences between the delayed-databinding and early-databinding. The first difference is the binding time. The delayed-databinding which is based on on-demand building is performed at the Services Invoking stage. But the early-databinding which is based on any parsing style is performed at the Message Processing stage. The other difference is that the delayed-databinding is specially designed for SOAP message binding, and only uses the Body element to bind, and creates XML object model for other elements. But the early-databinding uses the entire SOAP message to bind.

As shown in Fig.5, the unmarshalling databinding gets the parameters type and XPP3 parser or XML object model from service invoker, and then uses the TypeMapping to convert XML data types to Java data types. At last, it returns the parameters object. On the contrary, the marshalling databinding gets the return value type and object from the service invoker, then uses the TypeMapping to convert return value object to output stream or XML object model.

The TypeMapping is the core of databinding, and its responsibility is to convert between the primitive data types of Java and XML Schema. It contains all primitive data types mapping, such as integer, string, long, etc. The compound data type is composed by some primitive and compound data types. Essentially, the compound data type mapping is a group of primitive data types mapping.

Firstly, the unmarshalling databinding analyzes the state of XML parser. If the parser has parsed completely, the unmarshalling databinding uses the XML object model to bind, otherwise uses the parser to bind. Secondly, it analyzes each parameter type. If the parameter type is primitive data type, then the unmarshalling databinding uses the TypeMapping to convert it directly, otherwise, uses Java reflection to analyze the JavaBean recursively. Lastly, the unmarshalling databinding returns the parameters object to the service invoker.

On the other hand, the marshalling databinding first analyzes the user's request. If the user needs XML object model, then the marshalling databinding marshals the return value object as XML object model, otherwise marshals it as output stream. The analysis of marshalling databinding is contrary to unmarshalling databinding.

C. *Adaptable Services Invoking Mechanism*

To eliminate Java reflection, we propose an adaptable services invoking mechanism, as shown in Fig.6. The developer needs to provide a customized service invoker which has implemented the service invoking interface of SOAP engine, and specifies the class name of it in the service deployment file. When the Web Service has been deployed, the service invoker specified will be loaded and registered to the engine. When the engine receives a request, it will check the services register to find the service invoker, and use this service invoker to invoke the Web Service. If we move the function of unmarshalling and marshalling from the SOAP engine to the Web Service itself, we can eliminate Java reflection. Because the customized service invoker knows the details of JavaBean, so it knows how to set or get the value of them directly. Thereby, this mechanism can eliminate Java reflection and improve the performance of SOAP processing. The more JavaBeans in Web Service, the more dramatically the performance improves.

V. EXPERIMENT

A. *Experiments Setup*

According to the proposed high-performance SOAP processing architecture, we develop and implement the SOAP engine HiSOAPprototype that adopts this architecture independently. We use HTTP as the underlying protocol for transporting SOAP XML payloads, although 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.

Test Services: We implement four common services of HiSOAPprototype. They are the same design as previous section, as shown in TABLE 1.

Test Case:

a) To validate the delayed-databinding and eliminated-reflection that can improve the performance of SOAP processing, we implement the HiSOAPprototype with different techniques, and then compare the performance of different HiSOAPprototypes.

b) To validate our high-performance SOAP processing architecture is better than the others; we compare the performance of HiSOAPprototype with that of Axis2, using their best performance. The Axis2 uses JAXB, and the HiSOAPprototype uses delayed-databinding and eliminated-reflection.

Test Environment: Client and Server hardware: 1000Mbps Ethernet card, 4GB memory, Intel(R) Core(TM) 2 quad CPU Q9550 @2.83GHZ. Server software: Windows server 2003, jdk 1.5.0, Axis2 1.5.1 and HiSOAP4.1.2. Client software: Windows 7, soapUI 3.6.1.

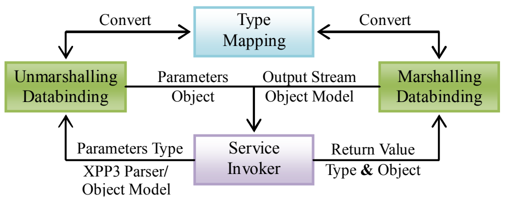
Convert
Type Mapping
Convert
Unmarshalling Databinding
Parameters Object
Output Stream Object Model
Marshalling Databinding
Parameters Type
XPP3 Parser/ Object Model
Service Invoker
Return Value Type & Object

Figure.5 Delayed-databinding

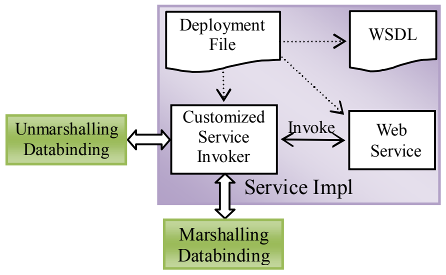

Figure.6 Adaptable services invoking mechanism

Test Method: Firstly, in order to minimize the impact of network latency and network communication, we connect the client and server in an intranet, and configure the same HTTP connection parameters of HiSOAP and Axis2. Secondly, deploy these services and start the HiSOAPprototype or Axis2. Lastly, use soapUI to send and receive the SOAP message consciously in a minute, and compute the average time cost of each service performed.

B. *Experiments Results*

As shown in Fig.7, in the *EchoAdd*, the performance gaps of four kinds of HiSOAPprototypes are very small. Because the size of request and response SOAP message of *EchoAdd* are very simple; for the primitive data type of Web Service, there is no Java reflection in type mapping, and the XML parsing is the primary cost of SOAP processing; only the delayed-databinding improves the performance but limited. The message size of *EchoArray* is a bit larger than that of *EchoAdd*, but the result is same as *EchoAdd*. The reasons are same as previous.

In the *EchoBean*, the performance gaps of different HiSOAPprototypes are very large, especially in the *EchoBeanList*. For the compound data types of Web Service, the type mapping is the primary time cost of SOAP processing, and the Java reflection is one of primary cost of type mapping. So, the delayed-databinding and eliminated-reflection techniques can both improve the performance of SOAP processing dramatically. Especially in the *EchoBeanList*, there are 12 JavaBeans in it, so, eliminating Java reflection is a very important performance improvement.

The eliminated-reflection technique can optimize the compound data types of Web Service, particularly when there are many JavaBeans in the Web Service. However, the delayed-databinding technique can optimize any data types of Web Service, and its performance improvement is only related to message size, especially the large message. If we adopt the delayed-databinding and eliminated-reflection techniques in the SOAP processing, the message size will be the only primary factor of SOAP processing time, not the complexity of data types.

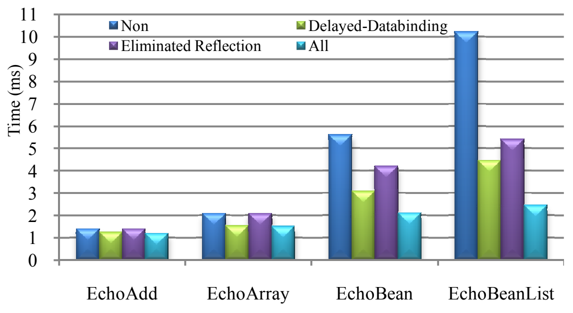

Figure.7 The performance of different HiSOAPprototypes

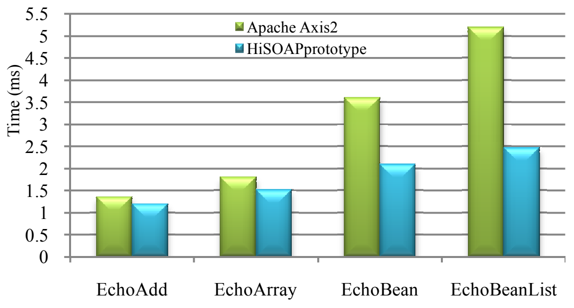

Figure.8 The performance of HiSOAPprototype VS. Axis2

As shown in Fig.8, in the *EchoAdd* and *EchoArray*, the performance gaps of HiSOAPprototype and Axis2 are very small. Because the message of both services is very simple, and there is no Java reflection in type mapping. So, the performance improvement is very limited. But the performance of delayed-databinding is better than that of early-databinding, and the XPP3 is faster than StAX. So, the performance of HiSOAPprototype is a bit better than that of Axis2. In the *EchoBean*, the performance of Axis2 decreases more quickly than that of HiSOAPprototype, especially in the *EchoBeanList*. In a word, the performance of HiSOAPprototype is better than that of Axis2.

VI. CONCLUSIONS AND FUTURE WORK

In this paper, we proved that the XML parsing, formatting and type mapping are the primary performance bottlenecks of SOAP processing by experiment. Then we proposed a new high-performance SOAP processing architecture named HiSOAP which not only improves the performance but also processes the databinding and extended protocol processing concurrently. Experimental results show that the delayed-databinding and eliminated-reflection techniques can improve the performance of SOAP processing dramatically. The HiSOAPprototype engine that adopts these techniques out-performances Axis2, especially in the complex compound data types of Web Service.

Because of the time and energy constraints, the HiSOAPprototype is still not perfect, such as the delayed-databinding does not support message validation and type reference. Moreover, the development of customized service invoker is relatively complex, and it is a hard job for service developer. So, in the next step, we plan to improve and enhance the function of delayed-databinding, and develop a service development tool that can auto-generate code for customized service invoker. Additionally, we will import the message caching mechanism to HiSOAPprototype too.

REFERENCES

[1] F. Bustamante, G. Eisenhauer, K. Schwan, P. Widener. Efficient Wire Formats for High Performance Computing. In Proceedings of the ACM/IEEE SC 2000 Conference. Dallas, USA, 2000, 39.

[2] D. Davis, M. Parashar. Latency Performance of SOAP Implementations. IEEE Cluster Computing and the Grid 2002.

[3] R. Elfwing, U. Paulsson, and L. Lundberg. Performance of SOAP in Web Service Environment Compared to CORBA. Proceedings of the Ninth Asia-Pacific Software Engineering Conference 2002 IEEE.

[4] K. Chui, M. Govindaraju, R. Bramly. Investigating the Limits of SOAP Performance for Scientific Computing. IEEE High Performance Distributed Computing 2002.

[5] C. Kohlhoff, R. Steele. Evaluating SOAP for High Performance Business Applications: Real-Time Trading Systems. In: Proceedings of the WWW2003, Budapest, Hungary, 2003.

[6] A. Ng, C. Shiping, P. Greenfield. An Evaluation of Contemporary Commercial SOAP Implementations. In: Proceedings of the 5th Australasian Workshop on Software and System Architecture, Adelaide, Australia, 2004.

[7] N. Mike. Improve XML Web Services' Performance by Compressing SOAP. http://drdobbs.com/windows/219401262, 2003.

[8] G. Heqong, L. Zhengxi. Improve The Transmission Performance of Web Service With Compression Technology. Computer Applications and Software 2006.

[9] Eriemm M. The Effects of XML Compression on SOAP Performance. World Wide Web, 2007, 10(3): 279-307.

[10] 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.

[11] A. Daniel, S. David, D. Kiran, R. P. Venkatesh. A High-performance Caching SOAP Implementation. IEEE Parallel Processing 2004.

[12] N. Abu-Ghazaleh, M. J. Lewis, G. Madhusudhan. Differential Serialization for Optimized SOAP Performance. In: Proceedings of the 13th IEEE International Symposium on High Performance. Distributed Computing (HPDC-13), Honolulu, Hawaii, 2004.

[13] N. Abu-Ghazaleh, M. J. Lewis, G. Madhusudhan. Performance of Dynamically Resizing Message Fields for Differential Serialization of SOAP Messages. In proceedings of International Symposium on Web Services and Applications, pp: 783-789, June 2004.

[14] T. Suzumura, T. Takase, M. Tatsubori. Optimizing Web Services Performance by Differential Deserialization. In: Proceedings of the IEEE/ACM International Conference on Web Services, orlando, USA, 2005,185

[15] L. Hua, W. Jun, N. Chunlei. High Performance SOAP Processing Based on Dynamic Template-Driven Mechanism. Chinese Journal of Computers 2006.

[16] L. Lei, N. Chunlei, C. Ningjiang, W. Jun. High Performance Web Services Based on Servie-Specific SOAP Processor. IEEE International Conference on Web Service 2006.

[17] Z. Wei, E. V. Robert. A Table-Driven Streaming XML Parsing Methodology for High-Performance Web Services. IEEE International Conference on Web Services (ICWS'06), 2006 IEEE.

[18] Z. Wei, E. V. Robert. An Adaptive XML Parser for Developing High-Performance Web Services. Fourth IEEE International Conference on eScience, 2008 IEEE.

[19] G. M. Tere, B. T. Jadhav. How to Improve XML Web Services Performance? International Conference and Workshop on Emerging Trends in Technology – TCET, Mumbai, India. 2010.

[20] Axis2. http://axis.apache.org/axis2/Java/core/