Application-aware Interface for SOAP Communication in Web Services

Hao Wang¹,², Yizhu Tong¹,², Hong Liu¹,², Taoying Liu¹
¹(Research Centre for Grid and Service Computing
Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100080, P. R. China)
²(Graduate School of the Chinese Academy of Sciences, Beijing 100039, P. R. China)
{wanghao, tongyizhu, liuhong, liutaoying}@software.ict.ac.cn

Abstract

SOAP protocol has emerged as the Web Service communication standard. Because of the relatively poor performance, many researchers focus on improving the speed of processing SOAP message. In this paper, we propose SPI, which introduces client usage pattern to the low level SOAP process infrastructure, in order to improve the performance of some kind of Web Services applications with specific communication patterns. The pack interface of SPI is an approach to reduce the number of the SOAP messages to improve the latency on client side. This SOAP optimization technique packs concurrent service requests into one SOAP message on client side, and unpacks and executes them on server side. Our implementation and performance study demonstrates the usefulness and effectiveness of the technique.¹

1. Introduction

Web Services include groups of standard protocols and data formats for representing and accessing services in wide area network environment. The Web Services Description Language (WSDL) describes Web Services interface, the XML-based SOAP is the most widely used communication protocol, and some underlying protocols, usually HTTP, define the message format and model in the transport level. Due to the simplicity, extensibility, language and platform independency, and interoperability between clients and servers, the SOAP and other web services protocols have been adopted to implement the basic architecture for Grid Systems.

Some investigations [2] [10] about the SOAP performance point out that the conversion between SOAP message and application data structure is expensive. So, many researchers focus on improving SOAP message processing performance, such as [1] [3] [4] [11]. These methods speed up the SOAP message parsing.

Otherwise, threads of server side process not only the data in transport layer but also those in communication layer and Web Services applications; and likewise on client. One of the most famous grid toolkits conforming to this model is GT4 [9] developed by Globus research group. In this model, protocol processing and application processing are coupled tightly in the same thread. As a result, one SOAP message contains XML data used for one service operation invocation.

From programmers’ perspective, current SOAP protocol processing provides the most fundamental program interface, like the most fundamental IO programming interfaces read and write in OS. And current research focusing on low level performance improvement all ignores the requirements from applications that is of great importance. In fact, current SOAP processing interfaces cannot meet requirements from all kinds of applications. In OS, an application that writes successive small messages using system call write will not get good performance because of the time consumed in frequent context switch. For parallel processing programmers, besides the basic communication primitives provided by OS, they need convenient programming interfaces with high performance like MPI designed specifically for parallel applications. Likewise, SOAP provides the most fundamental communication infrastructure in Web Service programming. But because of the lack of information that reflects the applications usage patterns,

¹ This work is supported in part by National Natural Science Foundation of China (Grant No. 60403023, 60573102, 90412010), 973 Program (Grant No. 2003CB317000, 2005CB321800), and Chinese Academy of Sciences Hundred-Talents Project (Grant No. 20044040)
some Web Service applications cannot achieve high performance with the current SOAP communication interfaces.

We propose SPI, which is the abbreviation for SOAP Passing Interface. Like MPI in parallel processing programming model, SPI aims to introduce some application usage pattern into the low level SOAP message processing. SPI provides interfaces like packing, remote execution and so on. This paper only describes the SPI packing interface, the goal of which is to decrease the transfer number of SOAP messages. We introduce some techniques that make it effective, including:

- staged independent thread pool technique, which is designed to decouple the tight binding between protocol and application, and make one SOAP message invoke multiple service operations.
- assembling technique, which is designed to pack request data or response data from multiple applications into one single SOAP message.
- and dispatching technique, which is designed to dispatch request data in one SOAP message from multiple clients to corresponding services operations, or to dispatch response data in one SOAP message from different services operations to correct client methods.

We evaluate the effectiveness of these techniques with a latency test, which demonstrates that the best performance improvement is up to ten times faster for the scenario of the client issuing successive request data which amount is moderate.

The remainder of this paper is organized as follows. Section 2 describes some background information and related work. Section 3 describes the design and implementation of our approach. Section 4 contains a detailed performance study. We summarize our conclusion and point the future work in Section 5.

2. Background and Related Work

In this section, we describe the commonly used web service architecture on server and introduce other optimization techniques. The goals of this section are twofold. First, we characterize the most popular architecture on server. This provides the necessary groundwork for understanding the intention of those techniques described in Section 3. Second, we compare our techniques with other related approaches, which can share our goal of improving the end-to-end performance.

![Figure 1. Common architecture on server side: Each incoming web service request is dispatched to a thread from the thread pool. This thread will process the HTTP message parsing, SOAP message parsing and service application execution. The similar operations will be done by the same thread, when return the response message.](image)

2.1. Server Architecture

From the perspective of Web Services developers, the architecture of Web Services on server side can be divided into application layer, communication layer, and transport layer. The service implementation belongs to application layer. Grid users can choose any languages to implement their own services, including Java, C, C++, Perl, Python, and C#. SOAP is the most often used protocol in communication layer. Grid users have a wide range of SOAP toolkit alternatives, such as gSOAP[7] [8], bSOAP[3] [4], and Axis[6]. Though SOAP specification permits any transport protocols, such as HTTP and SMTP, HTTP is the most frequently used standard in transport layer.
Figure 1 describes general architecture of SOAP message processing on server side. The thread reads a HTTP message from the socket and parses the message to get the payload that contains SOAP message. Then it parses the SOAP message and gets service request data. Finally, the same thread uses the request data to finish the service execution.

In Figure 1, the thread created in transport layer will complete the functions from the HTTP parsing to service operation execution. HTTP parsing, SOAP parsing and service execution are coupled tightly in the same thread. Such architecture results in the fact that one SOAP message only contains the data of one service operation. So, our design will modify the architecture to satisfy the aim that one SOAP message can invoke multiple services operations.

### 2.2. Other Optimization Techniques

Chiu et al.[2] investigate the limits of SOAP performance for scientific computing and conclude that the phase of serialization and deserialization is the performance bottleneck. In this paper, trie data is used to reduce the times of comparisons for XML tags. This optimization is especially useful in SOAP deserialization. The other optimization technique is message chunking and streaming, which reduces the time to establish the TCP/IP connections.

Devaram et al.[1] describe “parameterized client-side caching” of messages in client files. This optimization allows the client to reuse cached messages in files and only change a few of the parameters for subsequent sends. They report a best speedup of 800% over their own benchmark.

Differential serialization [3] helps bypass the serialization step for the messages that are similar to those previously sent from the same SOAP client or previously responded from a SOAP-based web service.

**Figure 2.** Staged independent thread pool on server side: There are two independent thread pools. One aims to process the data in protocol layer and the other aims to process the data in application layer. After parsing the SOAP message by a thread in the first thread pool, the dispatcher will trigger multiple threads in the second thread pool to execute the multiple service applications simultaneously. When return multiple response data in one SOAP message, the similar operations will be done.

Differential deserialization [4] [11] is the server-side analogue to differential serialization. Both of the approaches take advantage of similarities among messages in an incoming message stream to a web service. In [11], the authors recycle in-memory objects to eliminate the number of new created objects in memory. While in [4], the authors make use of checkpoints and full parser state check pointing to meet the same goal.

The intention of these approaches mentioned above is to improve the performance in processing SOAP message, therefore orthogonal to our motivation which is to reduce the number of SOAP messages.

### 3. Design and Implementation

SPI provides a group of application programming interfaces designed specifically for WS applications, such as packing, remote execution, et al. We only describe the pack interface in this paper.

In this section we will first introduce use case for the SPI pack interface, then propose our new architecture for Web Services on server side, the staged independent thread pool architecture, which is designed to improve the latency when client accesses Web Services and support better server-side concurrency. This technique decouples the binding
between protocol processing and application processing. We also introduce the assembler and dispatcher module both on server and client. An overview of these techniques is illustrated in Figure 2.

3.1. Use case

![Diagram of Travel Agent Use Case](image)

Figure 3. Travel agent use case (from [15])

Figure 3 illustrates the overview of the travel agent use case that is described in [15] by W3C Working Group. The scenarios describe how a user would make a reservation for a vacation package (flight and hotel room) by a travel agent service.

In the scenario, the travel agent service will find a list of flights from each airline service in its catalogue. In Figure 3, there are three airline services. So, the travel agent service should issue airline service requests three times, and the three service requests can be issued simultaneously in logic. If the airline services are in one service container, our approach can pack the service request data in one SOAP message and execute the requests simultaneously on server side. At the same time, there are many other similar situations in this use case, such as travel agent querying the list of rooms from three hotel services. Hence, the use case will benefit from our approach. We will describe the implementation and evaluation of travel agent service with our approach in section 4.3.

3.2. Design Goals

The primary goals of our approach are:

**Improving throughput of client side.** The method introduced in section 2 speeds up the process of SOAP message parsing, which effectively improve the client side latency. Our approach is designed to reduce the number of SOAP messages transferred to services, which can greatly improve the throughput of whole application while at the same time may not increase the latency of every client invocation.

**Supporting server-side concurrency.** Usually, clients can enhance the services concurrency by starting several threads to issue multiple SOAP request messages simultaneously. But this method cannot reduce the number of the SOAP messages. So, on server side, we design the dispatcher that triggers multiple threads to execute the services’ operation according to the service requests in one SOAP message. On client side, the assembler packs the data of multiple service requests into one SOAP message. Our technique requires no change to services’ code.

3.3. Staged Independent Thread Pool

In section 2, we mention that the request message is processed in three layers. HTTP protocol and SOAP protocol are processed in communication layer and transport layer respectively. So, HTTP protocol and SOAP protocol processing can be integrated into one stage, protocol processing stage. The application layer is in charge of application processing stage. Each stage corresponds to different independent thread pool. The reason why we choose thread pool based event driven model [5] is that too many concurrent threads will degrade throughput rapidly due to the frequent switch among threads.

In Figure 2, the protocol processing stage still adopts the methods in common service platform. After parsing the SOAP message, the protocol processing thread goes to sleep. Since the event of protocol processing finishes, one thread will execute the code of the dispatcher and assembler module. If the service request data include concurrent services request, some worker threads from the thread pool of the application processing stage will be assigned to complete the services request. When the event about the completion of services application execution happens, the response data will be disposed in dispatcher and assembler module. Then, the sleeping thread of protocol processing stage will be waked up to complete generating the packet in protocols’ layers. Finally, the HTTP message carrying multiple services response data will be returned to the client side.

In staged independent thread pool architecture, while the protocol processing is done in one thread, the application processing may be involved in multiple threads. The client need not start several threads, which issue multiple services request simultaneously, in order to raise the concurrency. At the same time, if without the technique of staged independent thread pool architecture, the client completes parallel services request only through the modification of the services implementation. This approach will increase the complexity of service development.
3.4. Assembler

Assemblers pack several services request data, or services response data, which are carried by multiple SOAP messages in general model, into one SOAP message.

Assemblers exist both on client and server. On client side, assemblers are responsible for congregating multiple services request data into one SOAP body. On server side, when worker threads complete services execution, the thread that congregates multiple services response data into one SOAP body will be activated.

Figure 4. This message is gSOAP-generated SOAP messages with assembler technique to the WebServiceX.NET that provides many services including weather service. There are two service requests in the SOAP message. The first request gets the weather in Beijing, China and the second gets that in Shanghai, China. The SOAP body contains Parallel_Method element. This element has two child elements that are packed into two service requests respectively.

Figure 4 illustrates the SOAP request message content that is generated by assemblers on client side. Suppose the client wishes to query the weather of Beijing and Shanghai. In the traditional model, the client should issue two service requests in two SOAP messages. In our approach, two service requests are packed into one SOAP message. At the moment, the client should use the library provided by assembler module. In the future, we will try to make the assemblers and dispatchers’ module pack and unpack SOAP message automatically. So, the client who would not like to modify the code will benefit from the same advantage too.

3.5. Dispatcher

Dispatchers dispatch multiple services request data or services response data, which are carried in one SOAP message, to different services operations or to different client methods.

Dispatchers exist both on client and server too. On server side, dispatchers are responsible for extracting multiple services request data and triggering the corresponding threads in the second staged thread pool to execute service operations. On client side, dispatchers extract multiple services response data from one SOAP message and return them to the corresponding client methods.

When the dispatcher receives the SOAP body, as shown in Figure 4, it analyzes the request data, which is parsed by parsers, such as SAX[12] and DOM[13], to decide how many worker threads should be activated from the second staged thread pool. If the services are implemented in Java, the dispatcher passes parameters including class name, target methods, and target method parameters to the worker threads. The operations on client side are similar.

3.6. Implementation

At present, we implemented our techniques in Java both on client and server and in C only on client. We will complete our techniques with C language on server side. Currently, we provide our approach for Axis (Axis 1.3 Final) and gSOAP (2.7.7) respectively. In section 4, the Axis implementation is tested. The gSOAP implementation will be tested in near future.

Due to the handler chains model, which is the Axis’s architecture, we implemented our technique as server handlers. So, services’ code need not be modified. On client side, we provide a library that is responsible for informing the assemblers and dispatchers module to do their work.

4. Performance Study

In this section, we describe the typical scenarios and the performance of our techniques.

The server was run on a dual processor 2.4GHz Pentium 4 Xeon with 1GB DDR Ram running Red Hat 9 Linux whose kernel version is 2.4.20-8mp. We used Apache Tomcat 5.0.28 as HTTP server. The client was run on a processor 1.8GHz Pentium 4 with 512MB DDR Ram running Windows XP with SP2. We will test our technique for Axis. For Java version tests, Sun
JVM 1.4.2_07 was used on both server and client. The JVM was started with "-Xms256m -Xmx1024m” options. The server and client communicated through the Megabit Ethernet link.

4.1. Experiments

This section studies the effect of the latency improvement on client side with our approach. Since we are interested only in the transferring time of the messages and the working time of assembler and dispatcher, we use Echo services, which only return the data whatever they received, to substitute the services of aforementioned use case on server side. We simulate the size of the services request parameters by varying the size of the echo service request data.

The experiments test the latency with three different approaches. The first approach is Serial Service Requests in Multiple SOAP Messages, which is the most popular model the clients use. The second approach is Parallel Service Requests in Multiple SOAP Messages, which represents the case where the clients start multiple threads to access many services simultaneously. The last approach is Parallel Service Requests in One SOAP Message which is implemented using our techniques. In Figure 5, Figure 6 and Figure 7, we use No Optimization, Multiple Threads and Our Approach to represent the three approaches respectively. For the experiments, we consider the following varied factors:

- The number of the request SOAP messages. There can be 1, 2, 4, 8, 16, 32, 64, and 128. In No Optimization approach, we issue M request SOAP messages serially in one thread of the client. In Multiple Threads approach, we issue M request SOAP messages simultaneously in M threads of the client. In Our Approach, we issue one SOAP message, which contains M services request data, in one thread of the client.

- The size of the message. We vary message size by sending a single array containing 10, 1K, and 100K characters, which represent little, moderate, and huge scale of service request data. We use N represent the size of the message.

The x-axis represents the number of service requests on client. The y-axis represents the running time of these service requests.

4.2. Evaluation

In the three figures, when M equals 1 that means only one SOAP message, the time consumption of Our Approach is more than that of No Optimization approach. The overhead is brought in for packing and unpacking multiple requests to and from one SOAP message.

Figure 5 and Figure 6 show that when the scale of service request data is little or moderate, Our Approach can get the least time consumption in the three
approaches. When the number of messages is 128 and the size of each message payload is 10 characters, Our Approach can achieve the performance optimization up to ten times faster than that of the first approach.

Figure 7 shows that Our Approach becomes the most time consuming if the services request data is huge.

The reasons about the variable performance are as follows:

Because our approach reduces the number of the SOAP messages from M pieces to one piece, the number of TCP connection and SOAP Header is reduced from M to one. When the payload in SOAP messages is at most the amount of the overhead that is reduced in our approach, the reduction is significant compared with the original amount of communication. Hence our approach implies a considerable reduction in communication time which is showed in Figure 5 and Figure 6. On the other hand, when the amount of payload in SOAP message is far more than that of the overhead reduced in our approach, the reduction is minor, or even negligible.

In Figure 5 and Figure 6, we can see that transferring M times N request data as a group in one SOAP message is less time consuming than transferring in M SOAP messages under the circumstances of the moderate request data. But, if the request data is large, this feature may not be valid and our approach brings more overhead.

We can draw the conclusion that our approach can improve performance significantly on the condition that there are multiple successive service requests and the payload in each SOAP message is moderate.

Furthermore, considering the implementation of some web service specifications which will add the overhead in SOAP Header, such as WS-security [14], our approach is more attractive in this case.

### 4.3. Optimized Travel Agent Service

This section studies the effect of our approach on throughput of the aforementioned travel agent service in section 3.1. The travel agent service is deployed on the client node, and airline services, hotel services, and credit card service are deployed on three server nodes.

![Sequence diagram for travel agent service](image)

**Figure 8.** Sequence diagram for travel agent service

Figure 8 illustrates the actions of the travel agent service.  
1. It queries a list of flights from each airline services. There are three request SOAP messages, and likewise in step 3.  
2. It requests the user’s chosen airline to reserve the flight. Without loss of generality, assume that the user chooses the most economical airline, and likewise in step 4.  
3. It queries a list of rooms from each hotel services.  
4. It requests the user’s chosen hotel service to reserve the room.  
5. It contacts the credit card service to confirm payment. If successful, the credit card service returns the authorization identifier.  
6. It confirms the flight reservation with the authorization identifier.
It confirms the room reservation with the authorization identifier.

We apply our approach to optimize the implementation of the travel agent service in step 1, packing the three flight request messages into one SOAP message, and likewise in step 3.

We test time consumption with and without optimization respectively. The test in each case is repeated 10 times. To complete the eleven service invocations, the time consumption without optimization is 408 milliseconds, while with optimization it is reduced to 301 milliseconds. Our approach achieves up to 26% performance improvement.

5. Conclusion and Future Work

We propose SOAP Passing Interface and its implementation technique, which helps improve service concurrency and reduce the number of SOAP messages transferred between clients and services. Instead of issuing one service request data in one SOAP message, several service request data are packed into one SOAP message and processed concurrently on server. Experiments have been made to evaluate the performance improvement through our approach. Several conclusions are listed below:

Firstly, our approach implements the application processing and the protocol processing as separate threads rather than as a single thread in tradition, thus providing the agility of usage pattern and the feasibility of improving the application performance.

Secondly, pack interface in SPI improves the performance up to ten times, when the client issues successive service requests and the size of the SOAP message is moderate.

Thirdly, our approach is effective if the size of the successive service request data is no more than 100 K byte.

Currently, users on client side should use the library to get the latency improvement. In the future, we will develop automatic communication techniques in order not to modify the code on client side. In section 4, we mention that if some Web Services specifications add the overhead to SOAP Header, the merit of our approach can be greater. So we will make more experiments on service applications with WS-security specification. Finally, we will implement and evaluate the suite of interfaces in SPI.

References


