A High Performance SOAP Engine for Grid Computing

Ning Wang[1], Michael Welzl[2] and Liang Zhang[1]

[1] Institute of Software, Chinese Academy of Sciences, Beijing, China
wangning@otcaix.iscas.ac.cn
[2] Institute of Computer Science, University of Innsbruck, Austria
michael.welzl@uibk.ac.at

Abstract. Web Service technology still has many defects that make its usage for Grid computing problematic, most notably the low performance of the SOAP engine. In this paper, we develop a novel SOAP engine called SOAPExpress, which adopts two key techniques for improving processing performance: SCTP data transport and dynamic early binding based data mapping. Experimental results show a significant and consistent performance improvement of SOAPExpress over Apache Axis.

Key words: SOAP, SCTP, Web Service

1 Introduction

The rapid development of Web Service technology in recent years has attracted much attention in the Grid computing community. The recently proposed Open Grid Service Architecture (OGSA) represents an evolution towards a Grid system architecture based on Web Service concepts and technologies. The new WS-Resource Framework (WSRF) proposed by Globus, IBM and HP provides a set of core Web Service specifications for OGSA. Taken together, and combined with WS-Notification (WSN), these specifications describe how to implement OGSA capabilities using Web Services.

The low performance of the Web Service engine (SOAP engine) is problematic in the Grid computing context. In this paper, we propose two techniques for improving Web Service processing performance and develop a novel SOAP engine called SOAPExpress. The two techniques are using SCTP as a transport protocol, and dynamic early binding based data mapping. We conduct experiments comparing the new SOAP engine with Apache Axis by using the standard WS Test suite.[1] The experimental results show that, no matter what the Web Service call is, SOAPExpress is always more efficient than Apache Axis. In case of handling an echoList Web Service call, SOAPExpress can achieve a 56% reduction of the processing time.

After a review of related work, we provide an overview of SOAPExpress in section 3, with more details about the underlying key techniques following in section 4. We present a performance evaluation in section 5 and conclude.

[1] http://java.sun.com/performance/reference/whitepapers/WS_Test-1_0.pdf

2 Related Work

There have been several studies [4, 5, 3] on the performance of the SOAP processing. These studies all agree that the XML based SOAP protocol incurs a substantial performance penalty in comparison with binary protocols. Davis conducts an experimental evaluation on the latency of various SOAP implementations, compared with other protocols such as Java RMI and CORBA [4]. A conclusion is drawn that two reasons may cause the inefficiency of SOAP: one is about the multiple system calls to realize one logical message sending, and another is about XML parsing and formatting. A similar conclusion is drawn in [5] by comparing SOAP with CORBA. Chiu et al. point out that the most critical bottleneck in using SOAP for scientific computing is the conversion between floating point numbers and their ASCII representations in [3].

Recently, various mechanisms have been utilized to optimize the deserialization and serialization between XML data and Java data. In [1], rather than reserializing each message from scratch, a serialized XML message copy is cached in the senders stub, which is reused as a template for the next message with the same type. The approach in [8] reuses the matching regions from the previously deserialized application objects, and only performs deserialization for a new region that has not been processed before; however, for large SOAP messages, especially for SOAP messages whose data always changes, the performance improvement of [8] will be decreased. Also Java reflection is adopted by [8] as a means to set and get new values. For large Java objects, especially deeply nested objects, this will negatively affect the performance.

The transport protocol is also a factor that can degrade the SOAP performance. The traditional HTTP and TCP communication protocols exhibit many defects when used for Web Services, including "Head-Of-Line blocking (HOL)" delay (to be explained in section 4.1), three-way handshake, ordered data delivery and half open connections [2]. Some of these problems can be alleviated by using the SCTP protocol [7] instead of TCP. While this benefit was previously shown for similar applications, most notably MPI (see chapter 5 of [6] for a literature overview), to the best of our knowledge, using SCTP within a SOAP engine as presented in this paper is novel.

3 SOAPExpress Overview

As a lightweight Web Service container, SOAPExpress provides an integrated platform for developing, deploying, operating and managing Web Services, and fully reflects the important characteristics of the next generation SOAP engine including QoS and diverse message exchange patterns. SOAPExpress not only supports the core Web Service standards such as SOAP and WSDL, but also inherits the open and flexible design style of Web Service technology because of its architecture: SOAPExpress can easily support different Web Service standards such as WS-Addressing, WS-Security and WS-ReliableMessage. It can also be integrated with the major technology for enterprise applications such as EJB and

JMS to establish a more loosely coupled and flexible computing environment. To enable agile development of Web Services, we also provide development tools as plug-ins for the Eclipse platform.

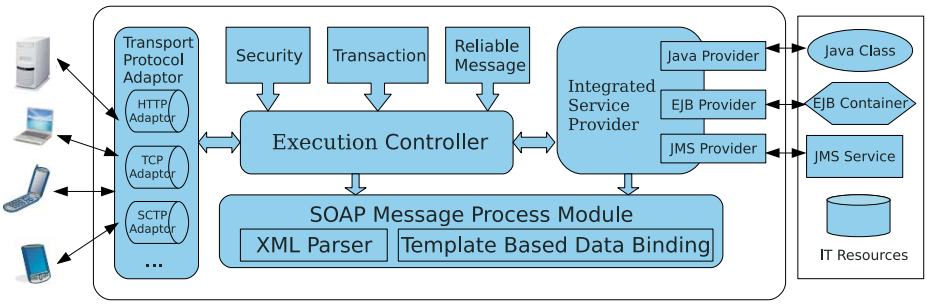

Fig. 1. SOAPExpress architecture

The architecture of SOAPExpress consists of four parts as shown in Fig. 1:

- Transport protocol adaptor: supports client access to the system through a variety of underlying protocols such as HTTP, TCP and SCTP, and offers the system an abstract SOAP message receiving and sending primitive.
- SOAP message processing module: provides the effective and flexible SOAP message processing mechanism and is able to access the data in the SOAP message in three layers, namely byte stream, XML object and Jave object.
- Execution controller: with a dynamic pipeline structure, controls the flow of SOAP message processing such as service identification, message addressing and message exchanging pattern management, and supports various QoS modules such as security and reliable messaging.
- Integrated service provider: provides an integrated framework to support different kinds of information sources such as plain Java objects and EJBs, and wraps them into Web Services in a convenient way.

4 Key techniques

In this section, we will present the design details of the key techniques applied in SOAPExpress to improve its performance.

4.1 SCTP Transport

At the transport layer, we use the SCTP protocol [7] to speed up the execution of Web Service calls. This is done by exploiting two of its features: *out-of-order delivery* and *multi-streaming*. Out-of-order delivery eliminates the HOL delay of TCP: if, for example, packets 1, 2, 3, 4 are sent from A to B, and packet 1 is lost, packets 2, 3 and 4 arrive at the receiver before the retransmitted (and

therefore delayed) packet 1. Then, even if the receiving application could already use the data in packets 2, 3 and 4, it has no means to access it because the TCP semantics (in-order delivery of a consecutive data stream) prevent the protocol from handing over the content of these packets before the arrival of packet 1.

In a Grid, these packets could correspond with function calls, which, depending on the code, might need to be executed in sequence. If they do, the possibility of experiencing HOL blocking delay is inevitable — but if they don't, the out-of-order delivery feature of SCTP can speed up the transfer in the presence of packet loss. Directly using the out-of-order delivery mechanism may not always be useful, as this would require each function call to be at most as large as one packet, thereby significantly limiting the number and types of parameters that could be embedded. We therefore used the multi-streaming feature, which bundles independent data streams together and allows out-of-order delivery only for packets from different streams. In our example, packets 1 and 3 could be associated with stream A, and packets 2 and 4 could be associated with stream B. The data of stream B could then be delivered in sequence before the arrival of packet 1, thereby speeding up the transfer.

For our implementation, we used a Java SCTP library which is based on the Linux kernel space implementation called "LKSCTP".[2] Since our goal was to enable the use of SCTP instead of TCP without requiring the programmer to carry out a major code change, we used Java's inherent support for the *factory pattern* as a simple and efficient way to replace the TCP socket with an SCTP socket in an existing source code. All that is needed to automatically make all socket calls use SCTP instead of TCP is to call to the methods *Socket.setSocketImplFactory* and *ServerSocket.setSocketFactory* for the client and server side, respectively. In order to avoid bothering the programmer with the need to determine which SCTP stream a function call should be associated with, we automatically assign socket calls to streams in a round-robin fashion.

Clearly, it must be up to the programmer to decide whether function calls could be executed in parallel (in which case they would be associated with multiple streams) or not. To this end, we also provide two methods called *StartChunk()* and *EndChunk()*, respectively, which can be used to mark parts of data which must be consecutively delivered. All *write()* calls that are executed between *StartChunk()* and *EndChunk()* will cause data to be sent via the same stream.

4.2 Dynamic Early Binding Based Data Mapping

The purpose of the data mapping is to build a bridge between the platform independent SOAP messages and the platform dependent data such as Java objects. The indispensable elements of the data mapping include XML data definitions in an XML schema, data definitions in a specific platform, and the mapping rule between them. Before discussing our data mapping solution, let us first explain two pairs of concepts.

[2] Java SCTP library by I. Skytte Joergensen: http://i1.dk/JavaSCTP/

– Early binding & late binding: The differences between early binding and late binding focus on when to get the binding information and when to use them, as illustrated in the Fig. 2. Here the binding information refers to mapping information between XML data and Java data. In early binding, all the binding information is retrieved before performing the binding, while in late binding, the binding is performed as soon as enough binding information is available.
– Dynamic binding & static binding: Here, dynamic binding refers to the binding mechanism which can add new XML-Java mapping pairs at run time. In contrast, static binding refers to a mechanism which can only add new mapping pairs at compilation time.

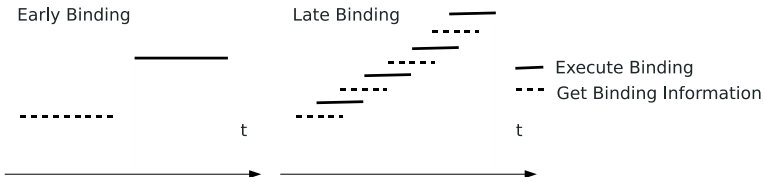

Fig. 2. Early binding & late binding

According to the above explanation, the existing data binding implementations can be classified into two schemes: dynamic late binding and static early binding. Dynamic late binding gets the binding information by Java reflection at run time, and then uses the binding information to carry on data binding between XML data and Java data. Dynamic late binding can dynamically add new XML-Java mapping pairs, and avoid generating assistant codes by using dynamic features of Java; however, this flexibility is achieved by sacrificing efficiency. Representatives of this scheme are Apache Axis and Castor. For example, Castor uses java reflection to instantiate the new class added to the XML-java pairs at the run time, and initialize it using the values in XML through method reflection. Static early binding generates Java template files which record the binding information before running, and then carries on the binding between XML data and Java data at runtime. Static early binding (as, e.g., in XMLBeans) improves the performance by avoiding the frequent use of Java reflection. However, new XML-Java mapping pairs cannot be added at runtime, which reduces the flexibility.

As illustrated in Fig. 3, we use a dynamic early binding scheme. This scheme can establish the mapping rules between the XML schema for some data type and the Java class for the same data type at compilation time. At run time, a Java template is generated based on the XML schema, the Java class and their mapping rules, which we call Data Mapping Template (DMT), by dynamic code generation techniques. The DMT is used to drive the data mapping procedure. Dynamic early binding avoids Java reflection so that the performance can be distinctly improved. Simultaneously, the DMT can be generated and managed at run time, which gives dynamic early binding the same flexibility as dynamic

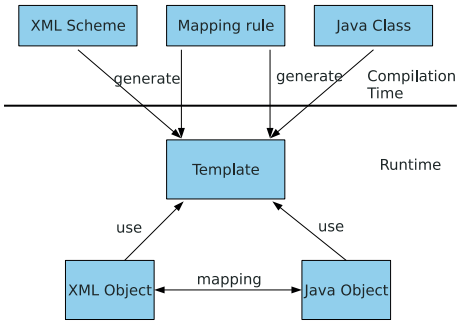

Fig. 3. Dynamic early binding

late binding. Dynamic early binding combines the advantages of static early binding and dynamic late binding.

5 Performance Evaluation

We begin our performance evaluation with a study of the performance improvement from using SCTP with multi-streaming. We executed asynchronous Web Service calls with *JAX-WS 2.0* to carry out a simple scalar product calculation, where sequential execution of the individual calls (multiplications) is not necessary. Three PCs were used: a server and client (identical AMD Athlon 64 X2 Dual-Core 4200 with 2.2 GHz), and a Linux router which interconnected them (a HP Evo W6000 2.4 GHz workstation). At the Linux router, we generated random packet loss with NistNet.[3]

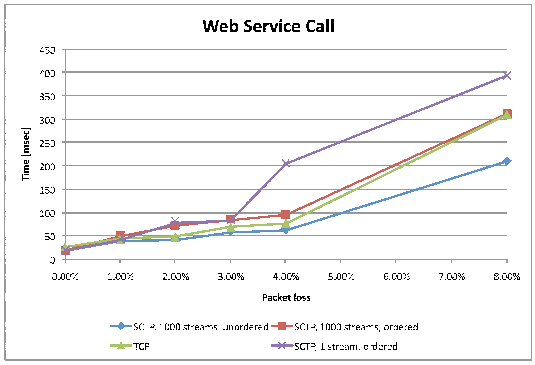

Fig. 4. Transfer time of TCP and SCTP using the Web Service

Figure 4 shows the transfer time of these tests with various packet loss ratios. The results were taken as an average of running 100 tests with the same packet

[3] http://snad.ncsl.nist.gov/nistnet/

loss setting each. As could be expected from tests carried out in a controlled environment, there was no significant divergence between the results of these test runs. For each measurement, we sent 5000 Integer vaues to the Web Service, which sent 2500 results back to the client. Eventually, at the client, the sum was calculated to finally yield the scalar product.

Clearly, if SCTP is used as we intended (with unordered delivery between streams and 1000 streams), it outperforms TCP, in particular when the packet loss ratio gets high. SCTP with one stream and ordered behavior is only included in fig. 4 as a reference value — its performance is not as good as TCP's because the TCP implementation is probably more efficient (TCP has evolved over many years and, other than our library, operates at the kernel level). Multiple streams and ordered transmission of packets between streams would theoretically be pointless; surprisingly, the result for this case is better than with only one stream. We believe that this is a peculiarity of the SCTP library that we used.

We then evaluated the performance of SOAPExpress as a whole, including SCTP and dynamic early binding based data mapping. We chose the WS Test 1.0 suite to test the time spent on each stage in the SOAP message processing. Several kinds of test cases were carried out, each designed to measure the performance of a different type of Web Service calls:

- echoVoid: send/receive a message with empty body.
- echoStruct: send/receive an array of size 20, with each entry being a complex data type composed of an integer, a floating point number and a string.
- echoList: send/receive a linked list of size 20, with each entry being the same complex data type defined in echoStruct.

The experimental settings were: CPU: Pentium-4 2.40 GHz; Memory: 1 GB; OS: Ubuntu Linux 8.04; JVM: Sun JRE 6; Web Container: Apache Tomcat 6.0. The Web Service client performed each Web Service call 10,000 times, and the work load was 5 calls per second.

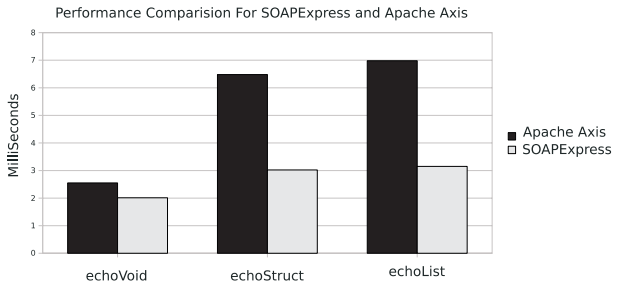

Fig. 5. Performance comparison among different types of Web Service calls

The benchmark we have chosen is Apache Axis 1.2. Fig. 4 shows the experimental results. For echoStruct and echoList, the XML payload was about 4KB. The measure started from receiving a request and ended with returning the response. We observed that for echoVoid, the processing time is very close between

the two SOAP engines, since echoVoid has no business logic and just returns the SOAP message with an empty body. For echoStruct, the processing time of SOAPExpress is about 46% of Apache Axis, and for echoList, the proportion reduces to about 44%. This is a very sound overall performance improvement of SOAPExpress over Apache Axis.

6 Conclusion

In this paper, we presented the SOAPExpress engine for Grid computing. It uses two key techniques for reducing the Web Service processing time: the SCTP transport protocol and dynamic early binding based data mapping. Experiments were conducted to compare its performance with Apache Axis by using the standard WS Test suite. Our experimental results have shown that, no matter what the Web Service call is, SOAPExpress is more efficient than Apache Axis.

7 Acknowledgments

We thank Christoph Sereinig for his contributions. This work is partially supported by the FP6-IST-045256 project EC-GIN (`http://www.ec-gin.eu`).

References

1. Nayef Abu-Ghazaleh, Michael J. Lewis, and Madhusudhan Govindaraju. Differential serialization for optimized SOAP performance. In *HPDC '04: Proceedings of the 13th IEEE International Symposium on High Performance Distributed Computing*, pages 55–64, Washington, DC, USA, 2004. IEEE Computer Society.
2. R. W. Bickhart. Transparent TCP-to-SCTP translation shim layer. In *Proceedings of the European BSD Conference*, 2007.
3. Kenneth Chiu, Madhusudhan Govindaraju, and Randall Bramley. Investigating the limits of SOAP performance for scientific computing. In *HPDC '02: Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing*, page 246, Washington, DC, USA, 2002. IEEE Computer Society.
4. Dan Davis and Manish P. Parashar. Latency performance of SOAP implementations. In *CCGRID '02: Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, page 407, Washington, DC, USA, 2002.
5. Robert Elfwing, Ulf Paulsson, and Lars Lundberg. Performance of SOAP in Web Service environment compared to CORBA. In *APSEC '02: Proceedings of the Ninth Asia-Pacific Software Engineering Conference*, page 84, Washington, DC, USA, 2002. IEEE Computer Society.
6. Christoph Sereinig. Speeding up Java web applications and Web Service calls with SCTP. Master's thesis, University of Innsbruck, April 2008. http://www.ec-gin.eu.
7. R. Stewart. Stream Control Transmission Protocol. RFC 4960, September 2007.
8. Toyotaro Suzumura, Toshiro Takase, and Michiaki Tatsubori. Optimizing web services performance by differential deserialization. In *ICWS '05: Proceedings of the IEEE International Conference on Web Services*, pages 185–192, Washington, DC, USA, 2005. IEEE Computer Society.