

High Performance Web Services with pull parsing and code generation

*Srinath Perera, Dimuthu Leelarathne
{hemapani,muthulee}@apache.org*

Abstract

This paper explains architecture for developing High Performance Web Service middle-ware. The architecture is built on top of two techniques “use of generated code that is optimized to process a request for given Web Service” and “Pull based reading and Push based writing of XML”.

At the deployment time of the Web Service most of the information about the Web Services is known from the WSDL file and that information can be use to generate a “code that is optimized for processing the request for the given Web Service”.

Further more even though the Push API is the most natural way to write the XML out the pull is the most natural way to read the XML. This fact can be used to write a web Service middle-ware with Pull to read the message in and Push to write the message out and that would yield a natural and efficient architecture for Web Services. This paper discusses such architecture and a sample implementation of that architecture AxisMora.

Introduction

As Web Services technology emerged, the First researchers of Web Services wanted to show that the concepts of Web Services are real and it can be used for at least some of the Use Cases. Similarly the Second generation of the Web Service middle-ware thrives to show that Web Services can be used to do the most of the Use Cases. But now the expectation from the third generation of the Web Services middle-ware is that they would works for all Use Cases including the cases that require high performance.

The “Performance” has been constantly identified as the price tag attached to Web Services. Therefore improving performance of Web Services has become a vital requirement. This paper purposes architecture for high performance Web Services. The proposed architecture is based on two initiatives.

- 1)Using XML Pull API to parse Web Service requests.
- 2)Employing optimized, specialized code to process requests of a given Web Service.

The paper starts by providing background information about the Web Services and XML parsing. Then the paper explains the purposed architecture followed by the problems encountered and the solution anticipated.

XML Pull API is the most natural way to read XML while XML Push API is the most natural way to write XML. This fact can be used to write a Web Service middle-ware with Pull to read the messages in and Push to write the messages. Hence it can be expected to yield a natural and efficient architecture for Web Services.

It is possible to generate optimized, specialized code for processing requests for a given Web Service using the information that is known at the time of its development.

The next section would introduce the background technologies and it will be followed by the proposed architecture explaining the problems and the solutions. Then the proposed architecture will be evaluated by giving statistics obtained using the implemented prototype, namely “AxisMora”.

Background

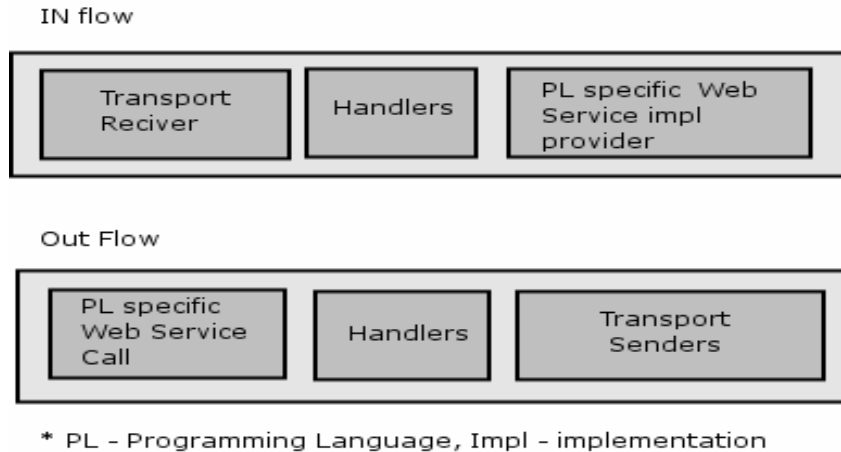
Web Services is a distributed service technology that uses XML messages send through a web protocol. The distinct feature of Web Services is ability to interoperate with different programming languages and platforms. The most widely used sub set of Web Services are Web Services based on SOAP[1] and WSDL [2] where SOAP defines the message format and the WSDL provides a way to describe the Web Services. Existence of the Web Services that uses messaging format other than SOAP and Web Service description other than WSDL is theoretically possible yet the SOAP and the WSDL are the world standards.

SOAP Message consists of SOAP Envelope. Envelope has optional SOAP Headers and mandatory SOAP body. SOAP body has the information about the current Service invocation while SOAP Headers have information about add on services like security and transaction.

The WSDL is a XML based, programming language neutral description of the Web Service. The usage of Web Services framework is to start with a WSDL, and convert it to programming language specific representations.

The conversion of the WSDL to the programming language specific artifacts can be automated and tools are available to generate the programming language specific implementation for the Web Service defined in the WSDL. At the runtime the conversation of the SOAP messages to the programming language representations is done by the Web Service middle-ware for each programming language. This process is done at both the client and the Server and the developer at the both side interact with the programming languages they are familiar with while the real Web Service Messaging is taken care by the middle-ware. Such middle-ware is called a Web Service implementation or a SOAP engine. Few examples of such SOAP engines are Axis Java/C++, .NET tools, WASP, GSOAP, and Glue.

Typical SOAP engine consists of two pipelines input and output which can be used at the both the Server and the client side.

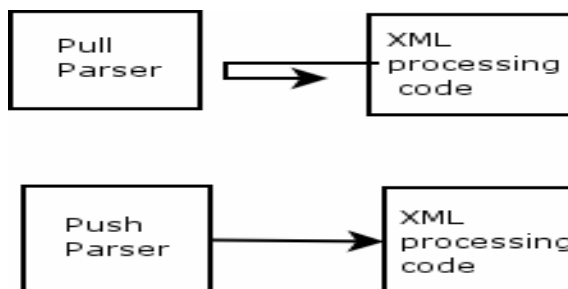


In the inflow there is a Transport Receiver accepts the incoming SOAP message and parses the SOAP envelope. Then the SOAP message is run through set of components which are the extensions mechanism for the web Services and these components will be called handlers - the usual java name. The Handlers are set of callbacks by mean of which the extensibility is provided to the SOAP engine. Usually the handlers process the SOAP Headers which provides add-on services such as security. After the Handlers, the Programming language specific implementation of the Web Service is invoked where the SOAP Message Body is converted in to the programming language specific representations and the Web Service implementation is invoked.

Out flow send the SOAP Message out and it contains SOAP representation of programming language parameters. This can be a Java/C++ object, a String, a DOM tree. Then the SOAP message is run through set of handlers. Once the Handlers are invoked the transport sender at the end of the out flow will convert the parameters from their programming language representations to SOAP message (XML) and send it to the other end through the transport protocol e.g. HTTP.

XML Parsing

XML parsing have basically two approaches - Push and Pull.



In the Push API the user write a “XML processing code” and register the code in to the parser. Then the push parser will walk through the XML documents and push the event in to the “XML processing code”. Once the push parsing is started the parser cannot be stopped until the complete XML file is processed as the Push parser is the one who has control of the execution.

When using the Pull API the user employ the parser to pull out the event from it. There is an imaginary cursor inside the Pull parser and the cursor move forward along the XML document upon users command. In the Pull Model the “*XML processing code*” holds the control of the execution and the user parses at his pace and as required. Furthermore the pull parsers are forgets what they read once move forward. Pull parsers can start parsing the SOAP message once the first character arrived in the stream and this is a huge advantage in network environments.

It is important to note that the Pull and Push are two complementing technologies that are used for the XML parsing. Both has interfaces are complementing each other. In both cases “*XML events*” goes through that interface of the parser where as difference lies in the fact that who holds the control, in the Push “*XML events*” are injected to the user while in the Pull “*XML events*” are requested by the User.

Related Works

Several approaches for High performance Architecture [6], [9] has been purposed over the time and they fell basically in to the two categories, alternation to the transport and the alternation to the architecture. Category one is based on the efficient transports and the efficient representations of data. Use of binary representation of the XML infoset [10], use of data compressions is one of the prominent solutions among them. Second Category is to use correct XML parsing models and fast and memory efficient SOAP processing model.

These two categories are independent in the sense that one can create a SOAP engine using architecture from one category and transport from other category. This paper provides an architectural solution for high performance Web Services.

Apache Axis takes first step towards high performance architecture with implementations based on SAX about three years ago. It is against Apache Axis Architecture the purposed Architecture is compared in the paper. But in the sense of the third generation web service engines Apache Axis is no longer fast.

Implementations like GSOAP go on to the length of including the XML parsing in to the architecture and extensive use of code generation [7], [8]. Reference [8] brings up the code generation which is one of the strong points of the purposed architecture.

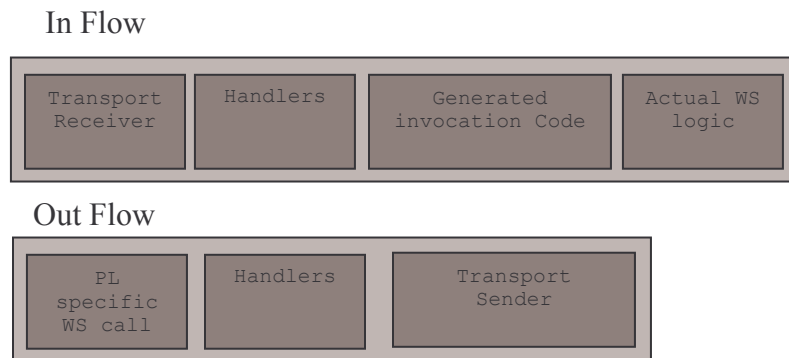
XSOAP written by Indiana University’s Extreme Laboratory uses XPP3 pull parser but does not use code generation. They compared the performance of SOAP with JavaRMI, and also analyzed the performance limitations. They found that the encoding and decoding time for SOAP was greater than for protocols such as Java’s object serialization. Also Microsoft SOAP Toolkit uses a XML Pull parser but the implementation is proprietary and finer details are not available.

Architecture

The architecture proposed basically build on top of two concepts First using XML Pull Model to read the SOAP message [2] and using XML push model to write the SOAP Message[1] . At the input path of the SOAP message it is most natural to pull the available content from the stream in need rather that wait till all the input available and

push them. It is interesting to note that as the SOAP Message is coming over the network the Push parser pull the bytes from the input stream while reading from the input stream converting the information that available in pull form to push form and with this approach that step is handed over to the SOAP engine. As explained above the with the pull model SOAP engine can start parsing once the first few characters of the input stream is available which would be huge advantage if the SOAP message is in range of Mega Bytes.

Pull model fits well with the SOAP processing as the at the ninety present of the time Handlers do not touch the SOAP Body so it make sense to SOAP Engine to parse the message only till the SOAP body, run Handlers and parse the rest of the Message at the invocation of the Service implementation logic, the programming language specific implementation. Then the code that do the conversion of XML to the programming language can directly read the events from the stream and create the programming language representation with out any from of storing in the intermediate states.



When the SOAP Message is received the SOAP engine will parse the SOAP Envelope and the SOAP Header. The SOAP Headers are converted in to the DOM elements and stored so that the Handlers can access them easily. If the Handlers do not ask for the SOAP Body the body will not be touched and when the execution comes to the Provider the *“programming language representation of the Web Service parameters”* is built directly out of pull events. One can have the same effect using push by putting the logic of invoking the handlers inside the XML processing code that is to be registered with the Push parser. But in that case the push parser is driving the code and effectively the pull like effect is generated with programming tricks while compromising the simplicity of the architecture.

At the deployment time the WSDL is available and WSDL contains the most of the information about the Web Service. For an example the Deployer can find from the WSDL the service interface, Kind of parameters to be expect, order of the Parameter etc. So it is make sense to process the SOAP Message [1] using the generated code that is specialized for parsing the XML message for a given Web Service. This generated code knows about the Web Service and the format of the XML in the SOAP message and can be viewed as a SOAP engine that is optimized for each Web Service.

Using this method the expensive dynamic inspections in the programming language representations (e.g. Java reflection) can be removed. The biggest advantage is

at constructing the programming language representation from the XML message. This construction is about lot of String processing and String comparisons and when the generated code parse the XML the code know what elements comes next and structure and the code can work with fewer string comparisons.

With the generated code there are lots of optimizations available for an example all the string comparisons can be replaced by the hash code comparisons where most of the hash codes are generated deployment time. Another example would be to write out SOAP message in the binary form to avoid the overhead of converting the String to binary. Plus the code generation can be done easily as in any case there is code generation already exists in the current implementations and the new code can be written minimal about of effort.

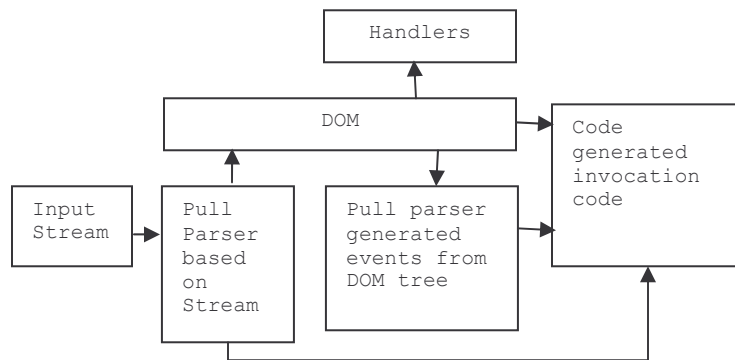
SOAP engine will code generate two parts, the Web Service invocation code and the code that construct programming language representation from the XML. From the two it is the construction of the programming language representation reward most and there is an added advantage of possibility of plugging in the date binding tool generated code when the programming language is java. The generated code will be register in to engine and will be invoke by the engine when the call to the corresponding service is received. This code is specially tailored for the given web service and will provide the best performance.

Problems and solutions

The use of pull parsing and the code generation presents following issues to be addressed.

First is How should the the SOAP Header processing is to be done? We can not assume that the SOAP Headers are in order nor that one handlers does not access the more than one SOAP Header. Further more in order to do the “Must Understand” attributes checks [1] the SOAP engine needs the all SOAP Headers. As the result the SOAP engine does not afford to forget the SOAP Headers while parsing and the SOAP Headers must be stored in some form. The DOM structure is used in the implementation that is the most common way the Handlers expect the SOAP Headers. Even though this add a overhead but as the Headers are comparatively small the solution is acceptable.

Second is what happens when the Handlers access the SOAP Body? Usually in the SOAP processing model Handlers are suppose to process the SOAP Headers, but it does not stop the Handlers from accessing the SOAP Body and there is inevitable example of Web Service Security that the SOAP Body access is required. But once the SOAP Handlers process the SOAPBody the body is forgotten as the Body is not remembered like the SOAP Headers. The problem is first addressed by declaring it is Handlers responsibility to set back the Body if they have accessed the body. But the architecture is improved to loosen the burden on the Web Service developer. Same problem occurs at the other response flow or the other flow. Here the Body is in the form of the java object we need to convert this object in to the XML as late as possible and give Handlers ability to change the body if needed. We address those both problems using the following approach.



At the request If the Handlers ask for the SOAP Body the SOAP Body is parsed in to the DOM element remember it and give it to the Handler.
 Handler may read it or change it and set it back in the form of the DOM element.
 When the engine precede it check the form of the request and if it is a DOM element it will wrap the DOM element with a code that traversal the DOM element and generate the pull events. This way the engine will not notice a difference.

At the response flow the result is wrapped by an object that knows how to convert the result to XML form.

This wrapper able to build the DOM tree from the result or write the result to a stream.

If the SOAP Handlers ask for the Body the result is converted to the DOM tree and given to the Handlers, the Handler may change it and set it back or read it.

At the end of execution DOM tree is written to the stream as XML.

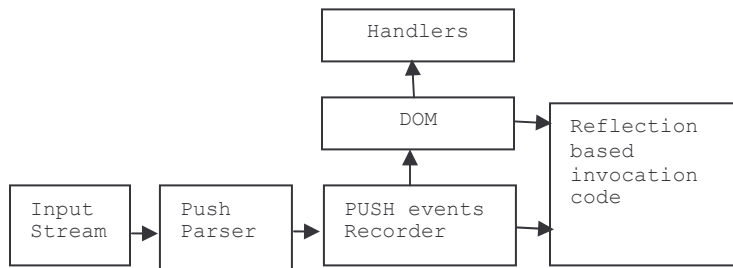
If the body is not accessed the result is written straight to the stream.

Third case is provide support for the “*href*”s or in other words ID’s inside the SOAP message. Any XML element can be have a ID and the other XML element may refer to that elements and as a result the SOAP message become a Graph where the sequential processing is of the SOAP message is impossible. This can be addressed by the code that process the pull events by remembering the Objects with ID’s but on the other hand using “*href*”s optimized few special cases and it is not clear that the effort that put in to address “*href*s” has reasonable payback as a result the architecture does not support “*href*”s.

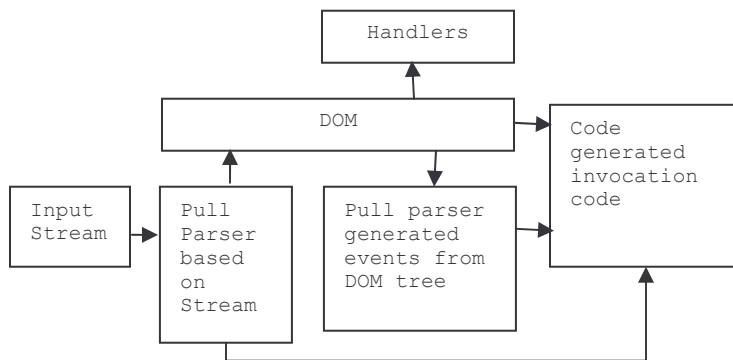
Fourth case is that with this approach the code generation is a essential and there is a compiler dependency on the SOAP engine if we try to do the HOT deployment. On the other hand but it is a limitation of the architecture and if absolutely necessary this can be addressed by using tools like Cglib[7] which generate the byte code directly rather than the java source.

Implementation, Axis Mora

Axis Mora is an implementation of the above architecture on top of Apache axis [8]. Axis Mora uses the Apache Axis design patterns, the basic architecture and the code when it is possible. The codes that are reused are deployment and serialization. Apache At the incoming message Axis original architecture can be shown by the following.



In apache Axis the input message is parsed with the SAX [9] parser and the Sax event are recorded. Then the SAX events are processed by the reflection based invocation code. The down side of the approach is all the time the SAX events that corresponds to the message must be kept in the memory.



With Axis Mora if the Handlers do not access the body invocation code can access the pull events from the stream with out a intermediate representation. If the Handlers access the body the Pull parser events converted to the DOM, then process by handlers and then converted to pull parser back so pull events based invocation code can handle them.

At the out flow both Apache Axis and Axis Mora uses the same architecture where as the push event generator shown in the Diagram is reflection based in apache Axis and it is code generated inside Axis Mora.

Apart form the message processing improvements at the most of the places Both the Apache Axis and the AxisMora use same concepts and the same code. But Axismora only demonstrate the applicability of the architecture and it does not have the other features like Attachments.

Performance, bench marks

AxisMora is compared for the performance with the Apache Axis using a Web Service that sends the array of Structs as the arguments. The Size of the SOAP Message can be changed by varying the size of the array. Test is done on 800MHz, 128 MB, Pentium three XP machine and in a 1.6GHz, 512MB, Pentium IV machines.

Both the Client and Server is run on the same machine to reduce the effect of the overhead added by the transport delay. SOAP requests were sent to both Axis1.1 and AxisMora and the Size of the Message was increased as 4,10,100,500,1000 and readings were taken. Time is measured via System.getTimeInMillis() method and message is repeated 30 – 100 times and the average is taken. There are about 10 requests of each message are sent before real benchmarking done to count for the initial overhead of the Web Service loading. The code that we used can be found on line [8] and following is a summery of the performance observed.

Petnium III, 800MHz 128MB XP			
	axis		axismora
4		244	34
10		584	79
100		6292	707
500MOB			3485
1000MOB			7492

Pentium iv, 1.6Gz, 512MB			
	axis		axismora
4		82	16
10		232	31
100		2779	275
500MOB			1349
1000MOB			3165

MOB indicates that the mem ory goes out of bound. The Graph shows the time taken by a one request and the data shows that as the size and complexity of the XML documents increase the performance gets better and better. The reason is fast deserialization at wrappers. As data get more complex and large the advantage of deserialization of AxisMora grows, giving better performance gain. Even though to establish the relationship between the performance gain would called for more systematic and complete performance analysis the above figure shows formidable evidence that the AxisMora is faster.

Conclusion

The explained Architecture try to analyze the Web Service invocation Scenario and try to optimized the best way to implements it. There are places that is identified as a possible improvements and First is use the information available at the deployment time to generate a optimized code for a given Web Service and second is to use natural XML parsing models pull for reading the information in and the push for writing the information out. The implementation, the analysis of the problems encounter and the possible answers and the performance analysis show that the architecture is viable.

References

- [1] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte, and Dave Winer, "Simple Object Access Protocol (SOAP) 1.1," World Wide Web Consortium (W3C), May 2000, <http://www.w3.org/TR/2000/NOTE-SOAP-20000508>, visited Jan. 2005.
- [2] Erik Christensen, Francisco Curbera, Greg Meredith, and Sanjiva Weerawarana, "Web Services Description Language (WSDL) 1.1," World Wide Web Consortium (W3C), Mar. 2001, <http://www.w3.org/TR/2001/NOTE-wsdl-20010315>, visited Jan. 2005.
- [3] JSR 101, JAX-RPC, Java API for XML based Remote Procedure Calls
- [4] Java Community Process Program, JAXP, "Java API for XML Processing"
- [5] Java Community Process Program, JSR-000173, "Streaming API for XML"
- [6] Kenneth Chiu, "Web Services Performance: A Survey of Issues and Solutions"
- [7] Robert A. van Engelen, "Constructing Finite State Automata for High Performance Web Services"
- [8] Robert van Engelen, "Code Generation Techniques for Developing LightWeight, XML Web Services for Embedded Devices"
- [9] Dan Davis y and Manish Parashar, "Latency Performance of SOAP Implementations"
- [10] Fast Web Services, Sun article