On Boosting the Web Service Performances for High Energy Experiments

F.Lelli, G.Maron
INFN Laboratori Nazionali di Legnaro

INTRODUCTION

We present a new XML parser, the Cache Parser, which exploits a cache to reduce the parsing time on the receiver side, by reusing information related to previously parsed documents/messages similar to the one under examination. We will show how our fast parser can improve the global throughput of any application based on Web or Grid Services, or also on JAXP-RPC. Experimental results demonstrate that our algorithm is 25 times faster than the fastest algorithm in the market and, if used in a WS scenario, can dramatically increase the number of requests per second handled by a server (up to 150% of improvement) bringing it close to a system that does not use XML at all.

RELATED WORK

Parsers break documents into pieces such as start tags, end tags, attribute, value pairs, chunks of text content, processing instructions, comments, and so on. These pieces are fed to the application using a well-defined API, implementing a particular parsing model. Four parsing models are commonly in use [1, 2, 3, and 4]:

One-step parsing (DOM): the parser reads the whole XML document, and generates a data structure (a parse tree) describing its entire contents.

Push parsing (SAX): the parser sends notifications to the application about the types of XML document pieces it encounters during the parsing process.

Pull Parsing: the application always asks the parser for the next piece of information appearing in the document associated with a given element. It is as if the application has to "pull" the information out of the parser, and hence the name of the model.

Hybrid Parsing: this approach tries to combine different characteristics of the other parsing models to create efficient parsers for special scenarios.

The DOM parsing is the most general and easy to use technique, but its adoption involves a big use of CPU and memory. Conversely, the SAX and the Pull parsers API allow a better system resource usage but, in case of complex XML structures, the adoption of these programming API can be more difficult than the DOM API.

CACHE PARSER: AN OVERVIEW

As already mentioned in the Introduction, the goal is to cache a set of information, related to XML Document syntactic trees, for fast parsing similar documents received by a given WS through distinct SOAP messages. In particular, we propose the adoption of a checksum to be associated with each XML document. This checksum must be sent by the Sender of each SOAP message, and is used by the Receiver to detect whether a received document is "well formatted", and whether it shares the syntactic tree with an already parsed one. In addition, the Sender includes a set of pointers to quickly retrieve information between XML tags. In other words we have introduced d cooperation between Sender and Receiver. In a WS-based middleware, XML data are encapsulated in XML/SOAP messages, so the Sender can include this checksum and the other information in the header of the message, in order to allow the receiver to quickly distinguish between XML messages with different structures. Since in a WS-invocation two SOAP messages are exchanged (SOAP Request and SOAP Reply) between the service requestor and service provider, this approach could be used on both the client and the server side in order to reduce the de-serialization process time.

An alternative optimization to overcome the de-serialization issues could be to transmit the message contents within encapsulated and compressed SOAP attachments. This solution, which also avoids the verbosity of the XML messages, is possible only if both the sender and receiver have established an agreement on the messages format. The Cache Parser, even if very efficient, does not require such agreement. First, a message prepared for a Cache parser can be de-serialized using whichever other parser, which will simply ignore any extra information contained in the SOAP header. Second, even if no agreement exists between sender and receiver, when all the messages exchanged have similar structures, we pay an extra overhead, typical of standard XML parser, only the first time one of these messages is received.

Like DOM, also our Cache-Parser exploits the syntactic tree of an XML document. However, if another document with the parsing tree has already been encountered, we do not need to build a new syntactic tree from scratch, because we can just navigate an already built one to know where to take the relevant information in the new document. In other words, we just need a visit of a tree structure of the document, instead of building its tree. The work tasks that are saved thanks to this process are: the object instantiation and the document scan process. In particular, since a syntactic tree was already created parsing previous received document, the cached information gives the possibility to skip the parsing of the tree structure of the received XML. In the server-side cache we memorize all the information about this tree, plus a set of pointers that allow us to have a “quick jump” to the information needed, without parsing the tag.

EXPERIMENTAL RESULTS

For better understanding the parsers behavior, and to know “how good” the cache parser is, we tried to estimate the intrinsic limits of the XML un-marshaling process. For computing this lower bound, we suppose that the parser already knows whether a document is well formatted (so it does not need a validation phase) and where the requested information is exactly stored in the document. Under such hypothesis, a parser process is just a string transfer from the XML-document to a memory location. As we will see in the following, this limit is very distant from the actual parsers performance, but using our Cache Parser we can get close to this limit. We performed two different tests to compare different parsing algorithms and the new Cache Parser. First we tested the fastest Java parsers available by parsing for 100,000 times a set of XML documents, and we compared them with our Cache Parser.

We performed these tests in both Unix (Dual Xeon 1.8 GHz, with 2 GB RAM) and Windows Systems (P3 1GHz with 1GB RAM). We noted that the test outcomes are OS independent. Table 1 only reports the values for the Unix system. In particular, the absolute time to parse all the XML documents, and the relative time with respect the Pull Parser time. It is worth considering that the absolute time depends on the hardware equipment of the server, but we found that the relative time is independent of the specific hardware. Our Cache Parser resulted to be 25 times faster than the Pull Parser to complete the tests, we evaluated the Cache Parser in a typical client-server scenario, where clients (PIII 600 MHz, with 256 MB RAM, Ethernet 1 Gbps) send an XML document to a servlet container (that is the base for any WS and Grid Services). The server (Dual Xeon 1.8 GHz, with 2 GB RAM) receives each document, parses it and sends back to the clients a “done” message. The test results are shown in Table 2, from which we can see that the throughput of the server using our Cache Parser is almost doubled with respect to the Pull one. However, we have to point out that the sender of an XML message is required to perform additional operations during the serialization process, like computing the additional tags and preparing the document hash key. In Table 3 we quantify this overhead, by measuring these additional costs. In a Dual Xeon 1.8 GHz, with 2 GB RAM we performed this measure by sending for 100,000 times a set of small XML documents (from 10 to 15 tags), first using the standard WS serialization, and then adding in the SOAP header the additional information that the Cache Parser needs. As we can see, since the sender can prepare the additional information when is building the XML document to send, this , de facto, does not impact on the XML serialization process.

To complete the tests, we evaluated the Cache Parser in a typical client-server scenario, where clients (PIII 600 MHz, with 256 MB RAM, Ethernet 1 Gbps) send an XML document to a servlet container (that is the base for any WS and Grid Services). The server (Dual Xeon 1.8 GHz with 2 GB RAM) receives each document, parses it and sends back to the clients a “done” message.

Parser Name	Parsing Time	Time / Pull Parser Time
DOM2	71658 ms	0,38
SAX	78573 ms	0,346
SAX2	49081 ms	0,555
Pull Parser	27219 ms	1
Cache Parser	1062 ms	25,63
Lower Bound	280 ms	97,211

TABLE 1: Parser Comparison

Parser Name	invocation per sec
SAX2	1500
Parser Upper Bound	3050
Pull Parser	1830
Cache Parser	2820

TABLE 2: Parser Comparison

The test results are shown in Table 2, from which we can see that the throughput of the server using our Cache Parser is almost doubled with respect to the Pull one.

Standard Parser	Cache Parser	Cache Parser / Standard Parser
171082 ms	173193 ms	1,012

TABLE 3: Document Preparation Additional Cost

[1] Yuval Oren. Piccolo xml parser for java. http://piccolo.sourceforge.net/.
[2] C. Fry. JSR 173: Streaming API for XML. Java Community Process Specification Final Release,March 25, 2004.
[3] The Apache Software Foundation. Apache Xerces. http://xml.apache.org.
[4] R. Mordani. JSR 63: Java API for XML processing 1.1. Java Community Process Specification Final Release, September 10, 2002.