Improving the performance of XML based technologies by caching and reusing information

Francesco Lelli [1,2], Gaetano Maron [2], Salvatore Orlando [1]

[1]*Dipartimento di Informatica, Università Ca' Foscari di Venezia*
[2]*Istituto Nazionale di Fisica Nucleare, Laboratori Nazionali di Legnaro*

francesco.lelli@lnl.infn.it gaetano.maron@lnl.infn.it orlando@dsi.unive.it

Abstract

The growing synergy between Web Services and Grid-based technologies is enabling profound, dynamic interactions between applications dispersed in geographic, institutional, and conceptual space. Such deep interoperability requires the simplicity, robustness, and extensibility for which XML has been conceived, making it a natural lingua franca for the network. Along with these advantages, there is a degree of inefficiency that may limit the applicability of XML.
In this paper, we investigate the limitations of XML for high-performance and high-interactive distributed computing. Our experimental results clearly show that focusing on parsers, that are routinely used for desterilize XML messages exchanged in these system, we can improve the performance of a generic the end to end web services based solution. Then we present a new parser, the Cache Parser, which uses a cache to reduce the parsing time by reusing information related to previously parsed documents/messages similar to the one under examination. Finally, we will show how our fast parser can improve the global throughput of any application based on Web or Grid Services, or also JAXP-RPC.

1. Introduction

XML [1] is a mark-up language used for describing structured data. An XML document consists of elements and their attributes, where each element has a name and is characterized by start and end tags. Element's content is included between the tags, and may consist of other elements, data or may be empty. Each element may have attributes that consist of pairs (name=value). XML enables users to introduce elements and attributes, their names and their relations in the document, by specifying a particular XML syntax (DTD/Xschema). The purpose of this syntax is to define the legal building blocks, the structure and the list of legal elements of an XML document.

An XML-based set of technologies are those at the basis of Web Services (WSs) [2], [3],[4],[5], by which existing legacy systems can be wrapped as WSs, and made available for integration with other systems. Applications exposed as Web services are accessible by other applications running on different hardware platforms and written in different programming languages. Using this approach, the complexity of these systems can be encapsulated behind XML/SOAP protocols.
A common trade-off in computing is between the need of universality and performance, and this is particularly true when WSs must be exploited to design a system in which both high performance and QoS requirements are mandatory. A limit case, in witch fulfilling both such requirements is really necessary, is scientific computing, which demands the full range of capabilities that industrial computing does: reliable transfer in distributed heterogeneous environments, parallel programs often exchanging data, self-contained modules sending events to steer other modules, and complex run-time systems designed for heterogeneous environments with dynamically varying loads, multiple communication protocols, and different Quality of Service (QoS) requirements. Unfortunately, the qualities of SOAP that make it universally usable tend to lower the communication performance. In particular, the features that make XML communication inefficient regard the primarily ASCII format of XML, and the verbosity of XML, due to the need of expressing tags and attributes besides the true information content.
As we will see in session 2, in a WS environment a lot of runtime activity is however spent in parsing XML documents: every client or server process needs to exploit an XML parser to send/receive messages. So speeding up the parsing algorithm should have a big impact on the total communication time, by largely reducing overheads. In particular, we are interested in reducing the overheads on the receiver side, where the task of a parser is to desterilize the message by checking whether it conforms to the DTD/Xschema syntax, and extracting data from the textual XML representation.

We noted that in high performance computing systems based on WSs, like contemporary Grids currently programmed through Grid Services,(i.e. a technology build on top of WSs), each subsystem routinely exchanges information by using very similar XML-formatted messages. The exchanged XML information often has the same "structure", i.e not only the same DTD/Xschema syntax, but also the same particular syntactic tree. "Standard" parsers do not use this information to improve unmarshaling algorithm performance, so we develop a cache-based system that takes advantage of this behavior. When the system parses a new XML document, it first tries if his structure matches an already know structure. This is quickly carried out by testing a document checksum. In case of a cache hit, the document will be parsed with a fast algorithm that exploits the stored knowledge on the document syntactic tree. Otherwise, the document will be analyzed by using a standard parser, and a new cache entry will be created to store the syntactic tree of the new document.

Our test-case and our motivating application is the Instrument Element (IE) Grid Component, which consists of a coherent collection of services, which allow us to remotely configure, partition and control a physical instrument, and permit this instrument to be better integrated into a computational Grid. The IE has been successfully exploited to design several pilot applications of our GridCC project [6], [7] and its implementation is currently based on WS technologies. Inside this component a set of WS interface called VIGS (Virtual Instrument Grid Service) allow users to access a real instrument, thus also plugging the specific instrument into the more traditional Data and Computational Grids. An example of use of this IE is the Compact Muon Solenoid (CMS) experiment [8], where the IE is the master controller of the Data Acquisition (DAQ) system. The problem to solve is when the experiment is taking data, since it demands high network traffic. In this case a multitude of services interoperate with each other in a large LAN composed of about 6000 machines, by using XML/SOAP lingua franca for exchanging information. As show in Figure 1, the instrument Manager (i.e., an IE component) organizes the elements of a DAQ in sets, checks their status, controls the "quality" of theirs computing behaviors, and so on.

The VIGS, i.e. the user interface of our IE, has a static structure. In addition the XML documents/messages that are exchanged between the IE's processes and the specific instruments are usually characterized by a "persistent" structure. Note that in our WS implementation such messages are XML-formatted ones, which are inserted in SOAP envelopes and then passed via HTTP to a receiver that parses it to extract valuable data.

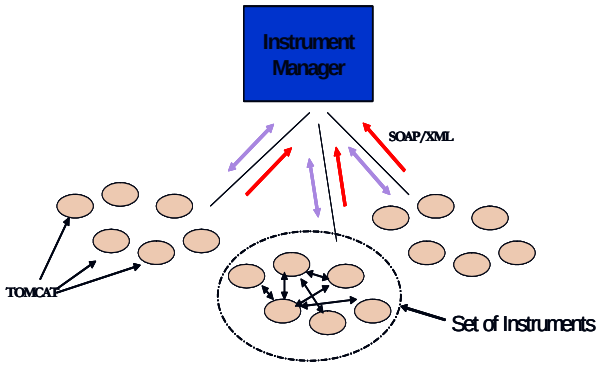


Figure 1 the Instrument Element Use Case

Since in our use-case the XML documents are short and transmitted over a fast network, the idea of reducing the transmission delay by compressing the XML messages does not seem as good as developing a more efficient parsing algorithm based on XML documents persistence, thus reducing the overheads on the receiver.

Even if all the remarks about the persistence of the structures of the message are motivated by our specific Grid use-case, where a multitude of senders have to send multiple times messages characterized by the same structure to a small set of receivers, a similar persistence in XML messages exchanged can also be observed in several other distributed applications based on Web/Grid Service technologies.

The rest of the paper is organized as follows: Section 2 will present an overview of the considered architecture for building our benchmarks and the obtained result. In Section 3 we will see our new parsing algorithm in detail, by focussing our attention on the issues and adopted solutions. In Section 4 we will present the performed test, the obtained experimental results. Finally, in Section 5, we will show our conclusions.

2. Understanding the XML limit

This paragraph describes a high level and general architecture, used to build a generic modern web-based application like our IE. This general architecture reviewed is applicable across technologies [10], [11] so that we use it to understand the limits that an XML solution can introduce in term of interactivity and handled requests per second. A modern web-based enterprise application has 3 layers (see figure 2):

- A client layer, which is responsible for interacting with the user, e.g., byWeb Page rendering;
- A middle tier which includes:
  1. A Presentation Layer which interprets user inputs (e.g., her/his submitted

HTML forms), and generates the outputs to be presented to her/him (e.g., a WebPage, including their dynamic content).

2. A Business Logic Layer which enforces validations, and handles the interaction with the data layer.

- A data layer, which stores and manage data, and offers the handling interface to the upper layers.

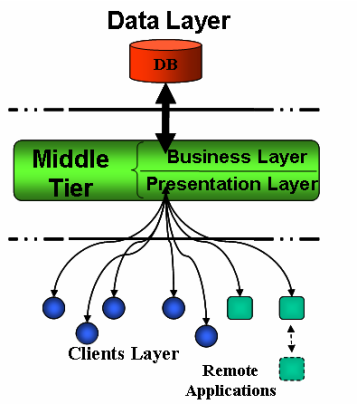


Figure 2 3-tier Architecture

This structure allows changes in legacy host access and development of new business logic to be kept separate from the user interface, dramatically reducing the cost of maintenance. Three-tier architectures also enable large-scale deployments, in which hundreds or thousands of end users are enabled to use applications that access business information.

Our motivating application, the Instrument Element, follow this abstract architecture: it is just a 3-tier application, with a strong separation between the Business Layer and the Presentation layer, that use a very simple data layer.

Talking about business to consumer applications, the client layer of a web application is implemented as a web browser running on the user's client machine. Its job in a web-based application is to display data and let the user enter/update data.

In a business to business scenario the client layer can be a generic application, compliant to the web-service standard. The presentation sub-layer generates (or displays) WebPages, or produces (or interprets) XML-based SOAP messages in a Web Service scenario. If necessary, it may include dynamic content in them. The dynamic content can originate from a database, and it is typically retrieved by the Business logic that:

- performs all required calculations and validations;
- manages workflow (including keeping track of session data);
- manages all the needed data access.

For smaller web applications, it may be unnecessarily complex to have two separate sub-layers in the middle tier. In addition the sub-layer communications typically do not use XML.

From a temporal point of view (showed in figure 3) a client (Web Services, web browser, java, c++, etc) performs a request to the business logic that dynamically retrieves the requested information. During the elaboration phase, the server can either perform one or more queries to a persistent storage, or interact with other sub-business unit. Once the information is retrieved, the server formats the retrieved information in a way that the client is able to understand, and sends it back to the client.

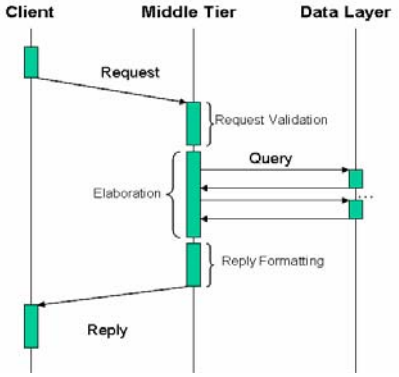


Figure 3 3-tier Sequence diagram

In a Web Service scenario, the clients need to exploit a further SOAP protocol layer over HTTP communications, while in human client scenario they the communication can be made just via HTTP.

The purpose of the next session is understood the real costs of the validation and reply formatting phases, by also varying the used technology. We setup a set of tests, using a synthetic application that follows the mentioned 3 layer architecture.

Others benchmark was already published on SOAP performance, but our approach differ from the ones discussed in [12] and [13]. What we would like to evaluate is the WS behaviour in a real scenario, understanding WS limits and giving the reader a clear idea of when the WS approach fails. In particular we will focus our attention on WSs exchanging short messages on a LAN network, which is the typical setting of our motivating application.

2.1. 3-tier Test bed:

In order to reproduce the architecture presented at the beginning of this paragraph, in the same local LAN we set up two different server machines, the former hosting

an Oracle DB Server as a storage system, and the latter hosting a Tomcat Application Server as a Middle Tier. Then we run a set of clients on other different machines in the same cluster.

The hardware used in this test is:

- **Database Server:** Dual Xeon 1.8 GHz, with 2 GB RAM, Ethernet 1 Gbps, OS Red Hat Linux Advanced Server 2.1.
- **Application Server**: Dual Xeon1.8 GHz, with 1.5 GB RAM, Ethernet 1 Gbps, OS Red Hat Linux Advanced Server 2.1.
- **Clients**: 25 Pentium III 600 MHz, with 256 MB RAM, Ethernet 1 Gbps, OS Linux Red Hat 9.0.

As Client/Server communications we used:

- Simple HTTP
- SOAP/XML over HTTP

Within the same Tomcat server, we also set up an Axis engine, in order to evaluate the performance of a WS communication (i.e., SOAP/XML+HTTP with automatic generated stubs) between client and server.

The DB table structure and the complexity of the queries are really simple (e.g., 1 table, with 5 attribute, as DB structure).

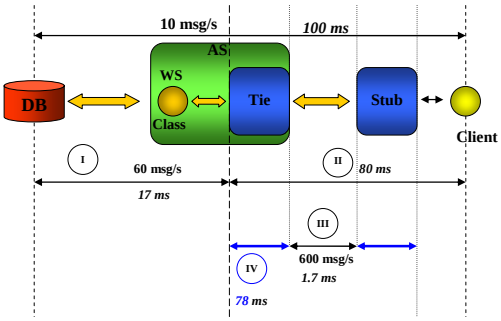
Our benchmark is thus the following: the clients (java applications) perform a request to a Tomcat Servlet or to Web Service. The business logic layer, in order to present the result to the client, performs a query using a pre-generated JDBC connection, to the data base server. Then the same layer formats the result and sends it to the client.

In this scenario we evaluate both the response time of a single client connection, and the global application server throughput in term of satisfied requests per second. The results are shown in next paragraph.

2.3 Service Request Time

We first measured the service request time, just using one client. Therefore we can assume that both the application server and the data base server are not busy, since they have to serve one request at a time.

Figure 4 shows the average times when using either servlet or Web Service technology.



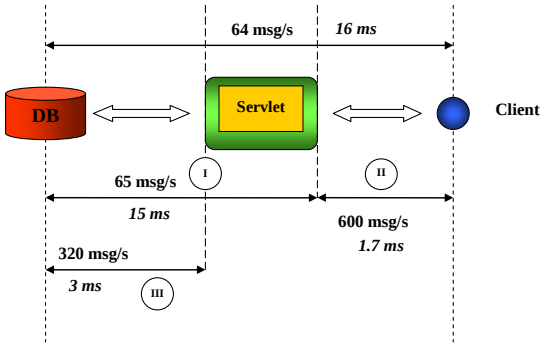


Figure 4: Service request time for servlet and WS

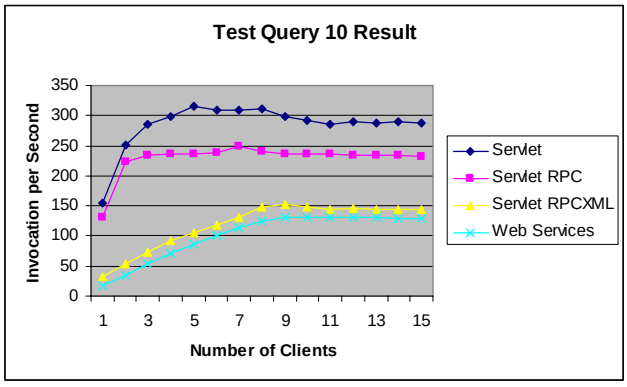
As we can see the total delay time of the WS (100 ms) is greater than the servlet one (16 ms). We can also note that the most of the request time is spent during the SOAP/XML serialization / de-serialization of the objects exchanged between the client and the server.

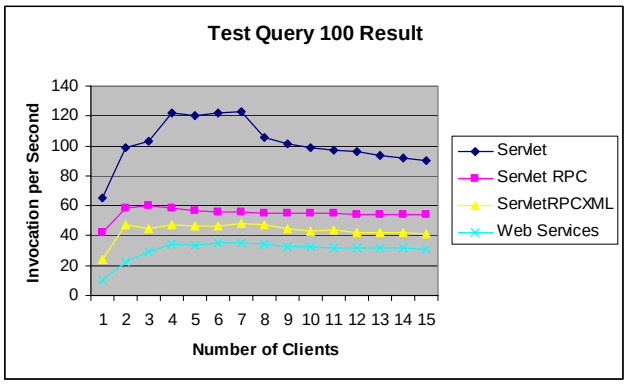
We have to note that the marshalling and un-marshalling process only depends on the exchanged data, and not on the complexity of the elaboration phase. This means that the WS cost, in terms of service delay, is important and non negligible for real time or interactive applications, where the final user/program typically needs a reply in a short amount of time. For non interactive service, this delay is acceptable.

2.4. Handled Request per Second

According with the scenario described in session 2 we also performed a test aimed to understand the number of requests handled per second in two different scenarios. In the first one, a servlet or a Web Service, once invoked, performs a query on the storage system and sends back the result. In the second scenario, we remove the Data Layer in order to exactly measure the service invocation time.

The plots in Figure 5 are related to the first performed measure.





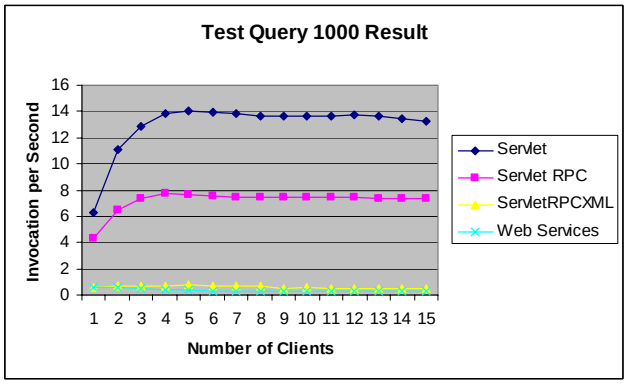
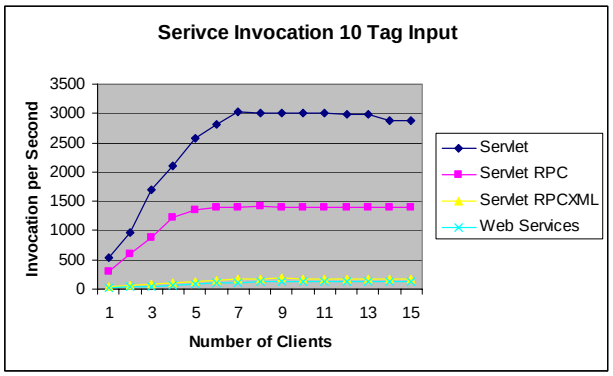


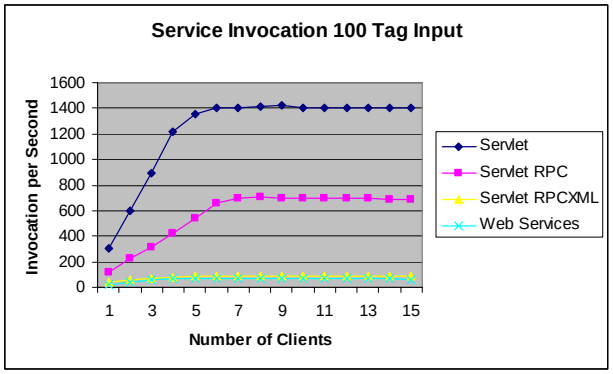
Figure 5: Handled request per second using a storage tier

The "ServletRPCXML" curve refers to a servlet that does exactly the same web service actions: in particular, it sends back to the client the result only at the end of the preparation of the result layout. The "ServletRPC" refers to a servlet that do exactly the same job of the previous one but the exchanged information are serialized in simple HTTP, while the "Servlet" curve differs from the first because, instead of buffering the result and sending it at the end of the preparation phase, it immediately starts sending it in a streaming way using an HTTP custom serialization

Finally, we can note that the performance of WS is similar to "ServletRPCXML". This means that its large performance decrease is due to both, buffering of the result and XML serialization, since clients and server have additional computational and synchronization overhead. As we can see, WS infrastructure introduces a remarkable performance overhead, and can not allow, due to the adopted RPC semantics, simple, but effective, code optimizations, like the one previous mentioned, to be exploited.

The plots in Figure 6 are related to the invocation of an empty service that immediately returns the results to the client.





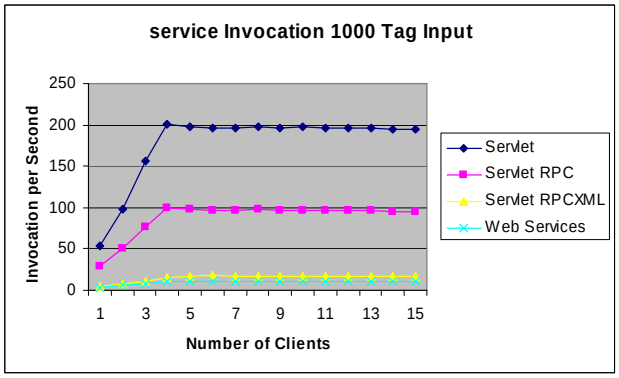


Figure 6: Handled request per second

As we can see, once again, the cost of WS in terms of number of handled requests per second is not negligible, and limits scalability since the number of invocations per second does not increase when we raise the number of clients. Finally, as one can expect, when messages are short and less complex (in terms of XML tags) the throughput is better than when messages are large and more complex.

As point in the introduction of this paper, we want increase the WS interactivity in a scenario where the exchanged messages between components are short so we decide to focus our attention on parsing algorithms in order to reduce the un-marshalling computational time.

Next paragraph present our solution that strongly reduce the overhead due to the XML use.

3. Cache Parser: an overview

In this section we present a high level view of our parsing algorithm, while in the next sections we will discuss in more detail each step of the algorithm.
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 contained in SOAP messages. In particular, we use a checksum, which is exchanged between the Sender and the Receiver of a given SOAP message. This checksum 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 also a set of pointer to quickly retrieve information between XML tags. In other words we introduce cooperation between Sender and Receiver. Since in a WS-based middleware, XML data are encapsulated in XML/SOAP messages, the Sender can include this checksum and the other information in the header of the message, in order to allow the receiver to fast distinguish between XML messages with different structures.
Like DOM [14], also our Cache-Parser exploits the syntactic tree of an XML document. However, in case another document with the parsing tree has been already 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 a tree construction. Unlike Deltarser [9], in the server-side cache we memorize all 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.
In paragraphs we will see in detail the 3 main part of the algorithm:

- The hashing of a received Document;
- The algorithm core that uses the cached information;
- The cache insertion/replacement strategy.

3.1. Hashing a Document

This subroutine in the receiver side must accomplish the following two tasks:

- To recognize if the new document syntactic tree is already known;
- To find the right entry in the cache data structure.

Achieving the second task is simple, if we can "well recognize" a document tree structure. All what we need is just an associative memory, where to store information using as a key a checksum computed over the syntactic tree of documents.
On the other hand, to accomplish the first task, the Sender of an XML document has to create the associated checksum: it is a hash key obtained from a syntactic tree representation of the XML document. This representation simply is a parenthesized string, summarizing the nesting of XML elements/attributes and the associated tags. Note that, since we are interested in characterizing the specific syntactic tree of a document, the representation used to compute the checksum does not include the information contained between tags. The Sender stores the computed checksum value in the SOAP Header, and sends it to the receiver together with the XML document.

3.2. Cache Algorithm

We show how our caching technique can be exploited using the same API of a Pull parser, even if we can it in any possible parser scenarios. The Pull parsers are the fastest ones developed till now. Using such parsers, the user asks for the information referring to a given tag, and provides a handler to elaborate the information between the tags.
The main difference between our cache-based algorithm and a standard Pull parser is that the Receiver can take advantage of the XML-Document tree structure knowledge, by quickly retrieving the information that must be passed to the specific handler.
If we have a cache hit we can assume that:

- NewXMLdocument shares a previous analyzed document syntactic tree;
- There exists a cache entry that stores "all the interesting information" about the syntactic tree of the NewXMLdocument;

First, the handler (user API) notifies that it wants to know the information associated to a particular tag (i.e., <login> see Figure 7). The Cache Parser algorithm "core" asks the cache to know where the information is memorized in the XML document. The cache returns the positions where the needed information can be found in the XML document. Finally we can give such information to the handler by using just a string copy.
In conclusion, by using this algorithm, we achieve a "well and fast" information retrieval from the XML document, if its syntactic tree is already known. Figure 7 gives an idea of what we need to memorize inside a cache entry.
It is worth noting that typical XML documents are more complex than the one we took as an example. In particular:

- Information contained between Tags normally does not have the same length.
- Tags have also attributes, and the lengths of their values may not be the same.
- Documents that share the same syntactic tree can have different tag start position.
- An XML document may use namespaces [27] to avoid collisions among tag names or attribute names, so an XML parser must handle namespaces for documents that use them.

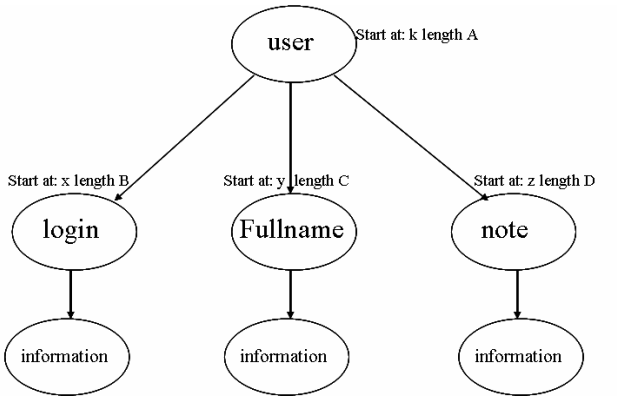


Figure 7: XML parsed document with pointer to the nodes

Therefore, in order to quick retrieve relevant information from a document, in the general case a Receiver needs to know the tag value start and end position. Our idea is that the Sender adds this information in a tagged element that we call "Map". We have to point out that this new element can be part of the SOAP Header, in order to not modify the original document.
Adding the "Map", we can know when the information associated with a Tag starts and finishes, just interpreting this attribute and without parsing the entire Tag information.
Note that the Map information refers to the associated (tree structured) Tags of the document. So it must be parsed to extract the right start/end positions of a given XML element.
In common use, typical data structure appearing in SOAP messages exchanged between Sender and Receiver are vectors. In out test case we noted situations where the numbers of vector elements are different, even if the structure of the message is the same. This poses serious problems to our cache parser, since the syntactic trees of two XML documents appear to be different when each vector element is stored as a distinct XML element. Our cache parse treats the two documents as completely different, thus storing their parsing information in different cache entries.. In order to reduce the memory usage and increase the cache hit rate without slowing down the algorithm, we can take into account how many sequential repetitions of the same Tag are present in the received XML-document. With this additional information, that it is stored in the SOAP header, two XML documents that only differ for a vector size, can be memorized in only one cache entry, just changing the way in which the document checksum is computed.

3.4. Cache Insertion/Replacement Strategy

Thinking of a cache insertion and replacement strategy we have to note that:

- Add a new cache entry cost time and this could not be a good investment if the document is not frequently exchanged between sender and receiver.
- Each cache entry memory as not a fix memory size and the exchanged XML-Document could be big with a consequential rapidly increases of the cache size.

We choose to adopt a Least Recently Used (LRU) cache replacement policy. In our use case the number of different XML messages exchanged is small so a little quantity of memory suffices to store all the information associated with all the different messages received

4. Parser Comparison

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 [15], [16], [17], [18]:

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. Notifications are typically implemented as event callbacks in the application code.
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.
Cache Parsing: we decide to adopt the pull-parsing approach, though in principle our technique can be used in conjunction with any other parser. As previously discussed, we maintain information about already parsed syntactic tree XML document to fast parse new document sharing the same XML tree.

4.1 Lower Bound of a Parser Algorithm

For better understanding the parsers behaviors, and to know "how good" the cache parser is, we try to estimate the intrinsic limit of an XML un-marshaling process elaboration phase. For computing this upper bound, we suppose that the parser already knows if a document is well formatted (so it does not need a validation phase) and that we can exactly know where the requested information is 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 actually parsers performance, but using our Cache Parser we can bring it close to this limit.

4.2 Parser benchmark

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 and Windows Systems, with different hardware configurations. We noted that the test outcomes are OS independent. Below we report only the values for a Unix System.
The absolute time obviously depends on the hardware equipment of the server, but, as shown in Column 3 of the Table 1, the relative time with respect to the Pull Parser time is an independent value.

Table 1: Parser Comparison

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

To complete the tests, we evaluated the Cache Parser in a typical client-server scenario, where clients send an XML document to a servlet container (that is the base for any WS and Grid Services).The server receives the document, parses it and sends back to the clients a "done" message.
The test results are show in the Table 2.
As we can see, if we use the Cache Parser, we can really improve the receiver part of a generic sender/receiver system. We have to point out that the Cache Parser requests more operations on the sender side, like the additional tags and the hash key preparation. In the Table 3 we quantify this overhead, by measuring this additional cost on the sender side. We perform this measure sending 100,000 times a set of small XML documents (from 10 to 15 tags) first using the standard WS serialization and second 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 "traditional" document, this do not impact on the information XML serialization

Table 2: Parser Comparison

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

Table 3: Document Preparation Additional Cost

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

5. Conclusion

The Web Services fully solve the "global enterprise integration" problem, but the proposed solution seems to exhibit a poor performance, and we believe that this could pose serious limitations on their actual applicability, as the number of commercial users will increase.
As shown in paragraph 2 we also believe that one of the limitations in using a full WS approach for implementing complex and highly interactive systems, comes from the large de-marshalling costs incurred on the receiver sides. To solve this problem, we have designed a new parser: the Cache Parser. It is able to "well and quickly" retrieve information from XML documents, using previous knowledge about the document syntactic tree. In particular, our Cache Parser, which is used to de-marshal XML messages on the receiver side:

- Uses a checksum to detect if a new document is "well formatted", and to know if it shares the syntactic tree with an already parsed one.
- Takes advantage of this XML-Document syntactic tree stored in a cache.
- It is based on a strict collaboration between sender and receiver.
- Consistently reduces the receiver parse time, without increasing the sender document creation time.

This algorithm is 25 time faster than a pull parser and if used in a WS scenario it can allow a 1.54 performance improvement factor in term of request handled per second. Finally it can be applied in any scenario where the client and the server decide to cooperate.

Acknowledgement

The GridCC project is supported under EU FP6 contract 511382.

6. References

[1] XML Specification: http://www.w3.org/TR/REC-xml/
[2] UDDI Specification. Version 2.0, 3.0. http://www.uddi.org/specification.html. 2005.
[3] Simple Object Access Protocol (SOAP) 1.1/1.2 W3C http://www.w3.org/TR/SOAP/. 2005
[4] W3C. Web Services Description language (WSDL) 2.0. Note. http://www.w3.org/TR/wsdl20-primer. 2005.
[5] Ian J. Taylor, from P2P to Web Services and Grids, Peers in a Client/Server World, Springer, October 2004
[6] GridCC Home Page: www.gridcc.org
[7] Gaetano Maron, Angelos Lenis, Sakis Moralis, Mary Grammatikou, Theodoros Karounos, Symeon Papavassiliou, Vasilis Maglaris, Paris Sphicas, Symeon Papavassiliou, Tiziana Ferrari, Constantinos A. Kotsokalis, Andrew Stephen McGough, Tatiana Kalganova, Peter Hobson, Roberto Pugliese, Francesco Lelli, David Colling, The GridCC Architecture(GridCC Architecture design www.gridcc.org) May 2005. (Authors are in no particular order).
[8] Compact Muon Solenoid project home page: http://cmsinfo.cern.ch/
[9] Toshiro Takase Hisashi MIYASHITA Toyotaro Suzumura Michiaki Tatsubori An Adaptive, Fast, and Safe XML Parser Based on Byte Sequences Memorization The 14th International World Wide Web Conference, Japan 2005
[10] Daniel Menasce, Daniel A. Menasce, Virgilio A. F. Almeida Capacity Planning for Web Performance: Metrics, Models, and Methods (Paperback) Prentice Hall PTR; Bk&CD Rom edition (May, 1998)
[11] John Blommers, Hewlett-Packard Professional Books Architecting Enterprise Solutions with UNIX Networking (Paperback) Prentice Hall PTR; 1st edition (October 15, 1998)
[12] Michael R. Head, Madhusudhan Govindaraju, Aleksander Slominski, Pu Liu, Nayef Abu-Ghazaleh, Robert van Engelen, Kenneth Chiu, Michael J. Lewis, "A Benchmark Suite for SOAP-based Communication in Grid Web Services", International Conference for High Performance Computing, Networking, and Storage, Seattle WA, November 2005.
[13] Satoshi Shirasuna, Hidemoto Nakada, Satoshi Shirasuna, Satoshi Sekiguchi, 11th IEEE International Symposium on High Performance Distributed Computing 2002 Evaluating Web Services Based Implementations of GridRPC.
[14] A. L. Hors, P. L. Hegaret, L. Wood, G. Nicol, J. Robie, and M. Champion. Document Object Model (DOM) Level 2 Core Specification Version 1.0, W3C Recommendation, November 2000. http://www.w3.org/TR/2000/REC-DOMLevel-2-Core-20001113.
[15] Yuval Oren. Piccolo xml parser for java. http://piccolo.sourceforge.net/.
[16] C. Fry. JSR 173: Streaming API for XML. Java Community Process Specification Final Release,March 25, 2004.
[17] The Apache Software Foundation. Apache Xerces. http://xml.apache.org.
[18] R. Mordani. JSR 63: Java API for XML processing 1.1. Java Community Process Specification Final Release, September 10, 2002.
[19]M. C. Chan and T. Y. C. Woo. Cache-based compaction: A new technique for optimizing web transfer. In Proc. INFOCOM'99, pages 117–125. IEEE, 1999.
[20]K. Chiu and W. Lu. A compiler-based approach to schema-specific XML parsing. In Proc. Workshop on High Performance XML Processing, May 18, New-York, NY, USA, 2004. Online publication.
[21]D. Davis and M. Parasha. Latency performance of soap implementations. In Proc. CCGrid'02, Workshop on Global and Peer-to-Peer on Large Scale Distributed Systems, Berlin, Germany, May 22-24, 2002, pages 407–412. IEEE Computer, 2002.
[22]M. Yuhara, B. N. Bershad, C. Maeda, and J. E. B. Moss. Effcient packet demultiplexing for multiple endpoints and large messages. In Proc. USENIX Winter 1994 Technical Conference, pages 153–165.USENIX Association, 1994.
[23]K. Chiu, M. Govindaraju, and R. Bramley. Investigating the limits of SOAP performance for scientific computing. In Proc. HPDC-11, pages 246–254. IEEE Computer, 2002.
[24]A. L. Hors, P. L. Hegaret, L. Wood, G. Nicol, J. Robie, and M. Champion. Document Object Model (DOM) Level 2 Core Specification Version 1.0, W3C Recommendation, November 2000. http://www.w3.org/TR/2000/REC-DOMLevel-2-Core-20001113.
[25]Investigating the limits of SOAP performance for scientific computing. In Proc. HPDC-11, pages 246–254. IEEE Computer, 2002.
[26]S. Shirasuna, H. Nakada, S. Matsuoka, and S. Sekiguchi. Evaluating Web Services based implementations of GridRPC. In Proc. HPDC-11, pages 237–245. IEEE Computer, 2002.
[26]A. Slominski, M. Govindaraju, D. Gannon, and R. Bramley. Design of an XML based Interoperable RMI System: SoapRMI C++/Java 1.1. In Proc. of The 2001 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001), June 2001.
[27] T. Bray, D. Hollander, and A. Layman. Namespaces in XML. W3C Recommendation, January 14, 1999, http://www.w3.org/TR/1999/REC-xml-names-19990114. Kaufmann, August 2002.