SOAP Processing Performance and Enhancement

Joe M. Tekli, *Member*, *IEEE*, Ernesto Damiani, *Senior Member*, *IEEE*, Richard Chbeir, *Member*, *IEEE*, and Gabriele Gianini

Abstract—The web services (WS) technology provides a comprehensive solution for representing, discovering, and invoking services in a wide variety of environments, including Service Oriented Architectures (SOA) and grid computing systems. At the core of WS technology lie a number of XML-based standards, such as the Simple Object Access Protocol (SOAP), that have successfully ensured WS extensibility, transparency, and interoperability. Nonetheless, there is an increasing demand to enhance WS performance, which is severely impaired by XML's verbosity. SOAP communications produce considerable network traffic, making them unfit for distributed, loosely coupled, and heterogeneous computing environments such as the open Internet. Also, they introduce higher latency and processing delays than other technologies, like Java RMI and CORBA. WS research has recently focused on SOAP performance enhancement. Many approaches build on the observation that SOAP message exchange usually involves highly similar messages (those created by the same implementation usually have the same structure, and those sent from a server to multiple clients tend to show similarities in structure and content). Similarity evaluation and differential encoding have thus emerged as SOAP performance enhancement techniques. The main idea is to identify the common parts of SOAP messages, to be processed only once, avoiding a large amount of overhead. Other approaches investigate nontraditional processor architectures, including micro- and macrolevel parallel processing solutions, so as to further increase the processing rates of SOAP/XML software toolkits. This survey paper provides a concise, yet comprehensive review of the research efforts aimed at SOAP performance enhancement. A unified view of the problem is provided, covering almost every phase of SOAP processing, ranging over message parsing, serialization, deserialization, compression, multicasting, security evaluation, and data/instruction-level processing.

Index Terms—Web-based Services, XML/XSL/RDF, performance measures, performance evaluation, security, integrity, and protection.

1 Introduction

Over the past decade, web services (WS) have transformed the web from a publishing medium used to simply disseminate information, into a ubiquitous infrastructure that supports transaction processing [48]. The WS technology differs from traditional software integration frameworks such as CORBA [54], DCOM [35], and Java RMI [66], in that WS utilize well established and open web protocols and formats, chiefly HTTP and XML [7], allowing smooth interoperability among heterogeneous systems. Nonetheless, the very feature that makes WS universally usable, namely the adoption of the ubiquitous XML standard [7], makes it difficult to reach the performance levels required by large-scale processes and applications [12]. In this paper, we survey a number of issues related to WS performance, particularly in the context of WS communications, discussing the main performance bottlenecks and possible improvements.

- *J.M. Tekli is with the Department of Computer Science and Statistics (ICMC), University of Sao Paulo, Av. Trabalhador Saocarlense, 400, 13566-590 São Carlos, SP, Brazil. E-mail: joe.tekli@icmc.usp.br.*
- *E. Damiani and G. Gianini are with the Department of Information and Technology, University of Milan, via Bramante 65, 26013 Crema, Italy. E-mail: {ernesto.damiani, gabriele.gianini}@unimi.it.*
- *R. Chbeir is with the LE2I Laboratory, UMR-CNRS, University of Bourgogne, 9 Alain Savary, 21078 Dijon, France. E-mail: richard.chbeir@u-bourgogne.fr.*

Manuscript received 1 July 2010; revised 29 Nov. 2010; accepted 28 Jan. 2011; published online 17 Feb. 2011.
For information on obtaining reprints of this article, please send e-mail to: tsc@computer.org, and reference IEEECS Log Number TSC-2010-07-0092.
Digital Object Identifier no. 10.1109/TSC.2011.11.

An individual web service generally comes down to a self-contained, modular application that can be described, published and invoked over the Internet, and executed on the remote system where it is hosted [61]. WS mainly rely on two standard XML schemata:

- Web Service Description Language (WSDL) [10] which supports the machine-readable description of a web service's interface. It allows the definition of XML grammar structures for describing WS as collections of communication endpoints capable of exchanging messages.
- Simple Object Access Protocol (SOAP) [82] is the protocol specification for message exchange among WS. It is based on the XML data model, and usually relies on existing application layer protocols (e.g., HTTP, FTP, SMTP, etc.) for message negotiation and transmission.

While these basic building blocks of WS technology are now firmly in place, performance issues have prevented using WS to implement large-scale distributed processes over large corporate networks or on the global Net. A major performance bottleneck resides in SOAP message processing [68]. The reason for SOAP performance criticality is twofold:

- On one hand, SOAP communication produces considerable network traffic, and causes higher latency than competing technologies, like Java RMI and CORBA [38]. This is a central problem especially within wireless communication networks with their

relatively low bandwidth and high latency [59], as well as the rising number of mobile computing devices (e.g., PDAs and mobile phones) increasing service demand, and consequently network bandwidth consumption [48].

- On the other hand, and perhaps more importantly, the generation and parsing of SOAP messages, and their conversion to-and-from in-memory application data can be computationally very expensive [1], [4]. In this paper, we adopt the following terminology: the process of translating a memory object according to a serialization format into an XML object is called *serialization*. The process of converting an XML structure into a memory object will be called *deserialization*. For complex XML structures, both these processes are computationally expensive. In fact, the translation between in-memory numeric data of type double and the ASCII-based XML representation format has been shown to consume over 90 percent of the end-to-end SOAP message processing time [12], which proves critical for various kinds of WS applications, ranging over business transactions (e.g., online booking and stock quote services), and scientific data processing (e.g., grid computing).

Several techniques have been proposed to improve SOAP processing performance. Many of them exploit the well-known concepts of similarity and differential encoding to 1) reduce processing time, in message parsing [45], [70], [71], serialization [4], [21], and deserialization [1], [68], as well as to 2) reduce network traffic via SOAP message compression [81] and multicasting [6], [58], [59]. Similarity-based SOAP performance enhancement is based on the straightforward observation that SOAP message exchanges usually involve highly similar messages. Messages created by the same implementation usually have the same structure, and those sent from a server to multiple clients tend to show similarities in structure and content (e.g., stock quote services [59] involving a large number of similar transactions requesting the latest stock data, as well as online booking and meteorological broadcast services [6]).

Thus, various efforts have been undertaken to process SOAP messages taking into account their similarities. The main idea is to identify the common parts of SOAP messages, to be processed once, regardless of the number of messages. Processing is only repeated for those parts which are different, avoiding a large amount of unnecessary overhead.

Another source of overhead is checking SOAP messages against security policies. Recently, several research efforts have focused on the impact of WS-Security policy evaluation on SOAP messages. WS-Security policies [19] specify authorizations, signature, and encryption schemes on SOAP elements and contents, and may introduce substantial processing overhead without (or despite) ad hoc performance enhancement [6], [14], [71]. Indeed, evaluating WS-Security policies can introduce an overhead much larger than standard WS invocation processing (6.9 times in average, according to [37]). A major portion of this overhead is related to the requirement of providing message level security (as opposed to channel-level security such as with TLS [79]) and to the XML encoding of message content.

Other performance bottlenecks arise from the limited amount of parallelism available on a conventional processor. Efficient parsing of SOAP and XML streams, as well as processing variable length encoded character streams would require hardware support for longer processing pipelines than standard CPUs can support. Handling XML streams entirely in software (for instance, by mapping processing pipeline stages to software threads) prevents the execution speed to be improved beyond the best processing rate of tens of clock cycles per character, and that best case performance can result in rates on the order of hundreds of clock cycles per character for many practical XML applications [78]. As a result, recent studies have addressed these performance bottlenecks by investigating nontraditional processors, namely parallel processing architectures and "XML machines," e.g., [8], [23], [30].

The goal of this survey paper is to provide a unified view of the problem, connecting the different aspects and techniques related to SOAP processing performance enhancement, including similarity-based and differential encoding techniques, WS-Security policy evaluation, and XML parallel processing architectures. The remainder of the paper is organized as follows: Section 2 presents a glimpse on SOAP processing, introduces its performance metrics, and discusses its main bottlenecks. In Section 3, we categorize, discuss, and compare some of the most prominent methods to SOAP performance enhancement. Section 4 discusses some ongoing challenges. Section 5 concludes the paper.

2 WS and SOAP Processing Performance

Experience with Service Oriented Architectures (SOA) has shown that WS performance is a crucial success factor for large-scale business processes [48]. It becomes even more crucial when services are made available on the open web, where 1) user requests to a certain service provider/company tend to increase with the amount of information and services the company makes available online [49], and 2) the fidelization of service consumers is on average lower than on a SOA infrastructure. If service latency becomes too high, clients may become frustrated and simply switch to another site or service offering the same functionality. Hence, WS performance problems can bring all kinds of undesired consequences, including financial and sales losses, decreased productivity and a bad reputation for a company [48]. Moreover, as the web evolves, mobile computing devices (e.g., PDAs and mobile phones) add another challenge to web services performance: wireless communication networks with their relatively low bandwidth and high latency [59]. Finally, current web systems and services are usually characterized by integration with databases, scheduling and tracking systems (e.g., Google Maps), requiring altogether high performance levels [27].

In the following, we first briefly present the key metrics which characterize WS performance levels. We subsequently discuss the various aspects of SOAP processing, and the corresponding performance bottlenecks.

2.1 Evaluation Metrics

Service-oriented infrastructures share some properties with component-based [26], [60] and web-based [47] applications, hence to some extent is it possible to apply existing resource metrics from the component-based software

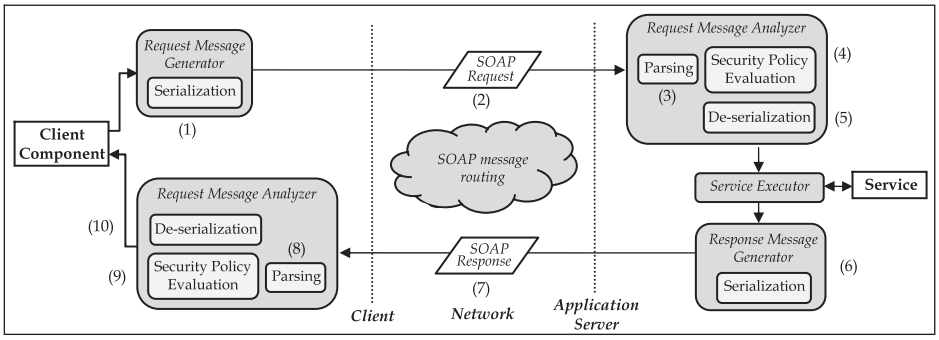


Fig. 1. Outline of a classic SOAP remote service call processing chain.

engineering and web applications domains in the context of SOA [60]. Namely, it is possible to classify performance metrics in three main categories: delay, bandwidth and usage, with *response time*, *throughput* and *network traffic* [48], [59] as the most relevant metrics normally used to assess the performance of WS for each category, respectively. Summary values of those metrics are normally obtained by aggregation in time and/or aggregation in space, or concatenation in space. A taxonomy of the relevant metrics can be found in [72] and references therein.

2.1.1 Response Time

Response time (also called *latency* or *end-to-end response time*) is the time perceived by a client to obtain a reply for a request for a web service. It includes the network time (latency and transmission delays on the communication link), as well as the processing delays at the server end point (service execution) and at intermediary nodes (switching time introduced by hubs, routers, and modems) [48]. The process with the longest processing delay in the processing chain is usually the key determinant of response time, and is identified as *bottleneck* (or time-sink). Response time is measured in time units.

2.1.2 Throughput

While response time is a performance metric typically of interest to end-users, throughput, which is defined as the number of requests executed per unit of time (e.g., I/O operations per second), is of more interest to administrators. It is usually evaluated on the server side [48]. There are many possible throughput metrics depending on the definition of unit of work. It is common to distinguish between point-to-point (or link) throughput (to quantify transport performance), node throughput (to quantify processing performance), and overall throughput in the system (also known as (a.k.a.) consistent throughput in the system) [60]. The overall system throughput is bound by the local throughput (link throughput and nodal throughput) of the least performing components in the transport and processing chain. Its basic unit of measure is *byte/sec*, however, for web service providers, it can be measured in *req/sec*—requests per seconds, *HTTPops/sec*—HTTP operations per seconds for web servers, or *tps*—transactions per seconds [71].

2.1.3 Network Traffic

The total network traffic for a communication scheme or session (e.g., *conversation*, i.e., a SOAP message exchange among two service end points) consists of the total size of all session-related messages sent over the network for the duration of the communication [59]. In other words, it encompasses the total number of bytes (corresponding to all messages exchanged during the communication session being evaluated) that are transmitted over the network [81]. Other related performance metrics exist, including: average utilization of a node, incoming/outgoing message rates, incoming/outgoing traffic for a node or the overall message rate in the system, which can also be measured in bytes or number of messages.

Over the past few years, several works have studied web service performance, e.g., [3], [22], [45], [68].[1] Most of them focus on SOAP processing and message exchange as the major players affecting web service performance levels. In the remainder of this section, we present a glimpse on SOAP processing, so as to pinpoint SOAP performance bottlenecks.

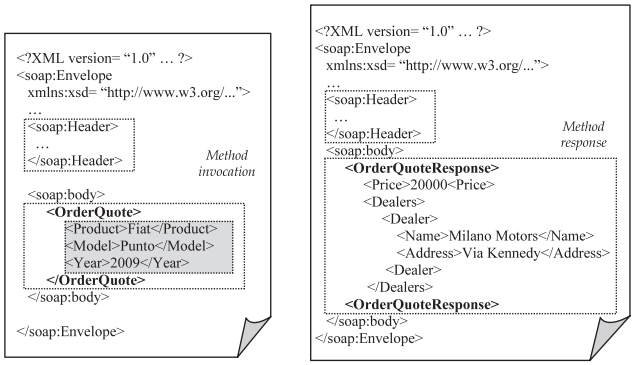
2.2 A Glimpse on SOAP Processing

SOAP [29] was specifically conceived as a messaging protocol to support interdependent interactions between otherwise independent entities, namely WS [12]. It is based on XML [4] and can support a variety of message exchange patterns, including request-response, one way messages, remote procedure calls, and peer-to-peer interactions [28].

Fig. 1 depicts a simplified activity diagram describing a typical SOAP remote service call processing scenario. Given two end-point services, usually identified as *client* and *application server*, an outgoing client SOAP message consists of a method invocation, a.k.a. client SOAP request, underlining a client call for method destined to the application server. An outgoing server SOAP message consists of a method response, a.k.a. server SOAP response, carrying the result of the action performed at the application server, following the corresponding method invocation. SOAP request and response messages are usually similar in structure. They both follow the same schema defined in the WSDL interface definitions of the services involved in the communication process. In general, a SOAP request/response message consists of a root node entitled *Envelope*, encompassing too elements: *Header* and *Body*. Consider for instance the sample SOAP messages in Fig. 2.

- *Envelope* provides the serialization context and namespace information for elements and parameters utilized in the message.

1. Most references in this paper address, in one way or another, web service performance. We only give a few here for clearness of presentation.



a. SOAP request message. b. SOAP response message.

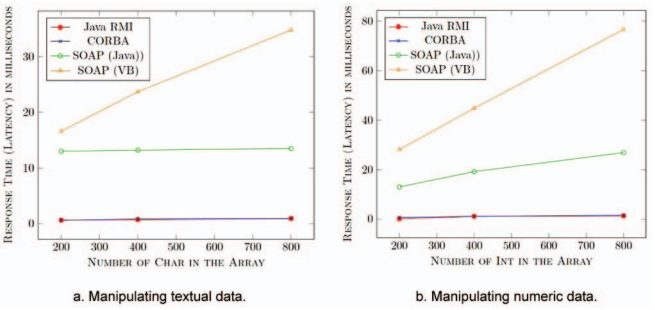
Fig. 2. Sample SOAP request and response messages.

- *Header* contains auxiliary information which is not related to the method invocation (or response) itself, such as transaction management and client/server information (e.g., client/server addresses, URL of final message destination).
- *Body* contains the actual data carried in the SOAP message. It usually starts with a subelement entitled with the method (or method response) name. The latter would encompass a child node for every parameter required to perform the local invocation.

As shown in Fig. 1, a common SOAP message exchange scenario consists of the following steps: First, a SOAP request message is created at the client side. Message creation requires serialization which consists in converting between in-memory application data representations and XML-based messages (Step 1). The request message is sent to the server application, usually via classic IP unicast routing (Step 2). At the server side, the message is first parsed, i.e., processed for lexical analysis (identifying characters and extracting tokens such as tags and contents) and validation (verifying the message's structural integrity w.r.t. the corresponding WSDL definition) (Step 3). The application server consequently evaluates its security policy rules on the received message, so as to identify and process those parts of the message which were assigned security constraints (authorization rules, signature verification, etc.) (Step 4), followed by message deserialization (converting between XML and the in-memory data representation) in order to be processed via the service executer (Step 5). As for the SOAP response message, the same procedure is undertaken, but this time in the inverse direction. The response message is created, i.e., serialized (Step 6), sent back to the client service via unicast routing (Step 7), parsed (Step 8), evaluated w.r.t. the client security policy rules (Step 9), and deserialized so as to transfer the processed data to the client service component (Step 10).

2.3 SOAP Performance Bottlenecks

SOAP's XML-based nature, which makes the SOAP protocol universally usable, tends unfortunately to work against achieving high performance [12]. The impact of XML message encoding on overall SOAP performance is omnipresent in almost every step of SOAP processing, underlining: 1) high response time and low throughput in SOAP



a. Manipulating textual data. b. Manipulating numeric data.

Fig. 3. Comparing SOAP response time, with CORBA [54] and Java RMI [66].

serialization [2], [4], parsing [45], [70], [71], security evaluation [6], [14], and deserialization [1], [68], mainly due to XML processing and the conversion between in-memory data and the ASCII-based XML format, as well as 2) high network traffic and bandwidth consumption during message transmission and routing [58], [59], [81], due to XML's verbosity and redundant textual characteristics.

To give an idea of the problem size at hand, we discuss the results of three studies, [17], [37], [81], evaluating the performance levels of SOAP in comparison with existing integration technologies, namely CORBA [54] and Java RMI [66]. Fig. 3 depicts the response time for a SOAP service call processing, i.e., the time required to generate and send a service request message and to receive its corresponding service response message, using two SOAP implementations (Java-based, Microsoft VB 6.0 toolkits) [17], in comparison with similar procedures to remote method invocations using CORBA [54] and Java RMI [66]. Timing results in both Figs. 3a and 3b show that SOAP performs very poorly in comparison with competing technologies. The time performance gap increases significantly when exchanging numeric data (e.g., integer arrays in [17]), which is due to the expensive process of converting in-memory numeric data to-and-from ASCII-based XML [12]. Fig. 4 depicts network traffic created by SOAP (two Java-based and Microsoft .Net-based toolkits were considered) [81], CORBA [54] and Java RMI [66], when varying the number of method invitations between two client and application server end points. Results show that SOAP produces significantly more network traffic than existing technologies. It requires almost three times more bandwidth than Java-RMI and CORBA, the latter using dedicated binary encodings for message exchange, in comparison with SOAP's XML-based textual format [81].

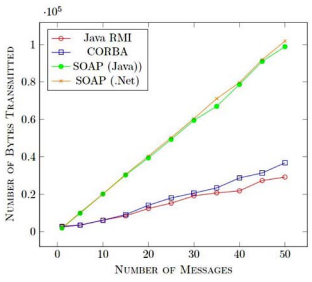


Fig. 4. Comparing SOAP service call network traffic, with CORBA [54] and Java RMI [66].

The need of encrypting and signing SOAP messages, which is of paramount importance especially when accessing services available on the open Net, has introduced additional delays. The WS-Security standard [19] is now widely used to express (in XML) the service providers' policies regarding what parts of the SOAP XML tree need to be encrypted and signed. In a recent study [37], the authors evaluate the additional overhead introduced by WS-Security policy evaluation w.r.t. standard processing of SOAP invocations. Their results show that WS-Security increases SOAP response time by a factor of 3 on average, while SOAP messages when using WS-Security are 6.9 times larger than unsecured SOAP messages (affecting network traffic accordingly).

In addition to evaluating the performance bottlenecks of SOAP itself, related works in [8], [39], [78] (among others) have addressed the shortcomings of conventional hardware computing architectures in handling XML-based data for large scale data sets and WS computing environments. They highlight the limited amount of parallelism in XML processing: both at the data level [8], [78] (i.e., in processing multiple pieces of data with one instruction), and at the instruction level [39], [78] (i.e., executing multiple instructions concurrently, a.k.a. multiprocessing). This family of hardware-based studies usually underlines the limitations of conventional processors in providing an efficient enough solution to evaluate multiple conditions of various types in parallel, which is central in XML string and character processing (e.g., verifying character integrity, whether an end tag matches a previously processed start tag, whether an attribute name is unique for a given element, and so on).

Some works [12], [28] address transport protocol bindings, namely the shortcomings of HTTP [24] as the application layer protocol used with SOAP for message negotiation and transmission. The authors in [12], [28] conclude that HTTP (specifically the earlier HTTP 1.0 version) negatively affects SOAP processing, and that it induces higher SOAP response time due to connection and message transmission overheads.

All relevant aspects of SOAP processing, the impact of the XML-based parallelism on SOAP performance, as well as the various solutions to SOAP performance enhancement to-date, are detailed in the following sections.

3 Improving SOAP Processing Performance

As mentioned previously, SOAP processing performance enhancement has been widely researched [6], [45], [58], [59], [70], [71]. Many approaches build on the simple observation that SOAP message exchange usually involves a number of highly similar messages. Invocations sent from the same client often reflect similar information needs, and thus similar SOAP message requests [21]. Likewise, messages sent from the same server to a single and/or multiple clients usually share strong similarities. Typical examples are various [6] such as stock quote services [59] (involving a large number of transactions requesting the latest stock data, hence similar stock quote request and response messages are processed), as well as online booking systems, and meteorological broadcast services [6], etc.

Several proposals addressing SOAP performance enhancement exploit, in one way or another, the similarity between SOAP messages, in order to gain in performance, e.g., reducing execution time, increasing throughput, and saving on network traffic. The main idea is to identify the common parts of SOAP messages, to be processed once, regardless of the number of messages.

We classify these solutions based on the performance metrics they target, and on the specific SOAP processing operations they address.

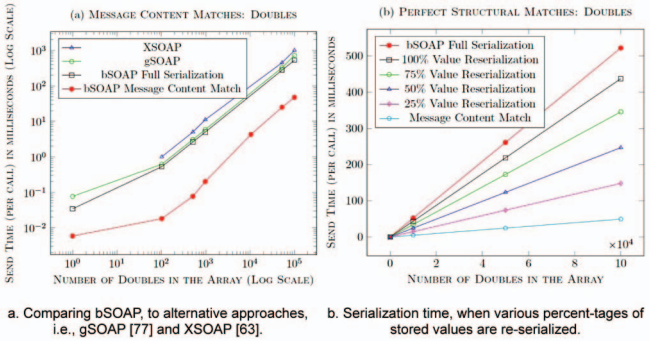
3.1 Methods for Improving Service Execution Time

Improving service execution time (i.e., attaining lower response time and higher throughput), has been investigated in various aspects of SOAP processing, addressing serialization, parsing, and deserialization operations.

3.1.1 SOAP Serialization

As mentioned previously, the serialization of SOAP messages consists in converting in-memory data types into XML. In this context, the main bottleneck consists in transforming in-memory data of numeric types into the ASCII-based XML representation format [12]. Consequently, the authors in [4], building upon the findings in [12], introduce a method for differential SOAP serialization, called bSOAP. The main idea consists in storing the SOAP messages in a dedicated buffer, to be used as templates for future outcalls, instead of discarding them after they have been sent over the wire. The message is normally serialized and saved during the first invocation of the SOAP call. Subsequent calls which share identical or similar message structures, as the message in the buffer, would avoid a significant amount of processing by only serializing the changes to the previously sent message. The authors address the problem of change tracking between in-memory data, and their serialized representations. Dedicated indexed tables, i.e., Data Update Tracking (DUTs), are associated with each serialized message, keeping track of the in-memory location of each field in the original structure to be serialized, and its position in the serialized message. A *dirty bit* is associated with each field, to keep track of those fields whose values have changed since the last send, in order to check which parts of the last message could be reused. Experimental results in [4] confirm the approach's better time performance, in comparison with regular serialization, and show that serialization time is linearly dependent on the percentage of in-memory values that must be reserialized (reflected by the number of dirty bits that are changed). When the whole message has to be serialized, bSOAP's serialization time is almost equivalent to that of existing SOAP toolkits, e.g., gSOAP [77] and XSOAP [63] (cf. Fig. 5a). Nonetheless, when the exact message is to be sent again (i.e., when none of the dirty bits are changed), time performance gain is maximal (almost 1,000 percent, cf. Fig. 5b).

In subsequent studies [2], [3], the authors address bSOAP's buffer management, mainly padding, which consists in stuffing the serialized message with white spaces to reduce the cost of message expansion when the latter is to be updated. Padding is useful when the new serialized form of some value does not fit in the current



a. Comparing bSOAP, to alternative approaches, i.e., gSOAP [77] and XSOAP [63].

b. Serialization time, when various percent-tages of stored values are re-serialized.

Fig. 5. Time performance of bSOAP differential serialization (reported from [4]).

space allocation (e.g., the value of an integer variable $i = 3$ which holds a single character space, is to be updated to $i = 1{,}003$ in the new serialized message, which requires four character spaces). Hence, padding allows on-the-fly message expansion, DUT table entries being updated accordingly.

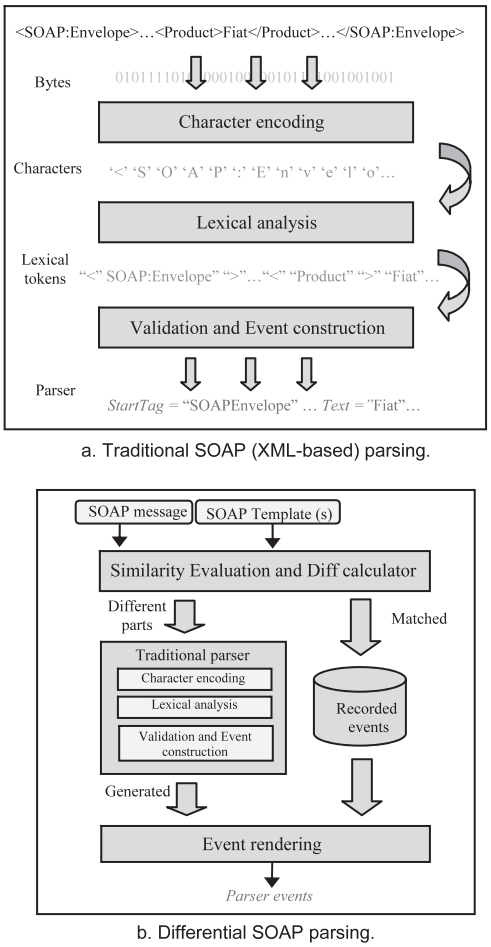
Various other SOAP buffer optimization techniques have been proposed [2], [3], [12], [77], namely chunking (dividing the SOAP message into chunks stored in different memory locations, to be processed separately) and streaming (pipelined-send, each message chunk being sent as soon as it is serialized, thus allowing an overlap of computation and communication). However, even after these optimizations, the conversion from in-memory data to the ASCII representation (over 90 percent of the end-to-end time) remains the most critical bottleneck [12], which emphasizes the relevance of differential serialization [4].

An approach comparable to differential serialization [4] is introduced in [21]. It addresses client-side SOAP message caching and allows entire request messages to be cached and sent as is. It also allows partial caching by reusing cached messages with identical structures, updating element values for subsequent sends. Similarly to [4], it relies on dedicated indexed structures in detecting correspondences between cached and outgoing messages. Nonetheless, the approach in [21] does not address partial structural matches (i.e., caching messages with partially different structures) as in [4], but only caches messages with identical structures. In addition, the authors in [21] do not discuss how to handle mismatched data sizes that require message resizing and expansion.

3.1.2 SOAP Parsing

As mentioned previously, SOAP parsing consists in analyzing the contents of the incoming SOAP message, to be consequently transformed into their in-memory application format via the deserialization component. In general, SOAP parsing consists in analyzing the characters in the SOAP message, extracting tokens such as tags and text, and then extracting and validating the underlying XML structure (cf. Fig. 6a). These tasks can be achieved using functions of existing XML parsers such as DOM [84] and SAX [47].

In this context, a few studies have proposed using special-purpose parsers, considering the particularities of XML and SOAP messages in order to amend performance.



a. Traditional SOAP (XML-based) parsing.

b. Differential SOAP parsing.

Fig. 6. SOAP parsing.

One of the earlier XML-based approaches promotes partial parsing [53], by 1) extracting the XML document structure (node references and hierarchical relations) in a preprocessing phase, and then 2) parsing only those parts of the document required by the application program, by looking up the document structure. The authors in [53] show that performance improves only when document (application) coverage is less than 80 percent, and that it otherwise declines due to preprocessing overhead. In [11], [74], the authors investigate the optimization of SOAP lexical analysis, using schema (WSDL) information, to more efficiently identify lexical tokens (e.g., tag names, attributes, etc.). Yet, such methods only target lexical analysis, disregarding byte-level character encoding and validation optimizations [69]. On the other hand, XSOAP [63] targets validation optimization and attempts to improve SOAP message validation performance by only executing the validation process on those elements specific to SOAP, namely *Envelope*, *Header*, and *Body*. Remaining parts, which usually consist of classic XML tagging, are disregarded in order to gain in parsing time. However, when the corresponding service requires complete message validation, the invalidated SOAP message parts have to be processed via a dedicated validation function to be added by the programmer in the service program [70], thus minimizing performance enhancement. A recent work [87] introduces a Table Driven XML (TDX) parser, that combines

the lexical analysis and validation of SOAP XML messages in a single pass. The idea is to prerecord the states of an XML parser produced from the corresponding (Schema) WSDL service description, as grammar productions rules in tabular form, and then to utilize a runtime streaming parsing engine to break up the SOAP message into a token stream, to be processed for well-formedness verification and validation at once. The authors in [87] show that their approach is more efficient than existing XML and SOAP toolkits where validation is enforced separately [5], [65], [77] (e.g., it runs six times faster than gSOAP [77]). Yet, TDX's performance is shown to be comparable (and even lower) when evaluated against a nonvalidating schema-specific SOAP parsing approach [74].

Instead of focusing on a specific phase of SOAP parsing, such as lexical analysis, or limiting the range of SOAP elements validation, more recent proposals in [45], [70], [71] focus on differential parsing, exploiting the similarities between SOAP messages, in order to skip unnecessary parsing altogether (including character encoding, lexical analysis, and validation) as depicted in Fig. 6b. In the following, we discuss the main approaches to differential SOAP parsing.

Template-based. T-SOAP [70] makes use of a predefined template, modeled via a finite state automaton (FSA), memorizing the basic structure of the SOAP messages, extracted from the corresponding WSDL definition schema.[2] It allows the identification of invariant and variable tag parts in the SOAP messages. Consequently, each incoming SOAP message is matched to the predefined template, and only those parts of the message, which correspond to variable parts in the template, are parsed (the invariant parts being already parsed in advance). While it induces a significant gain in processing time, in comparison with classic SAX [47] and DOM [84] parsers, a major limitation of T-SOAP [70] is its restriction to messages conforming to the same basic structure. In other words, a SOAP message with a structure different than that underlined in the predefined template would not benefit from T-SOAP [70] and would have to be parsed from scratch.

Multiple templates. In [45], the authors propose a more dynamic approach by managing multiple templates based on actual SOAP message structures, instead of using a single predefined schema structure. Incoming messages are first matched against the automaton, describing multiple message templates merged together. If the message matches any of the templates, then parsing is undertaken w.r.t. the variable parts of the corresponding template, similarly to [70]. Otherwise, parsing is undertaken via an ordinary DOM-based processor [84], and a new template corresponding to the unmatched message is created and appended into the automaton, to be exploited in upcoming parsing operations. While this technique provides more flexibility than T-SOAP [70], the authors in [45] underline that their method requires more memory for storing the combined automaton, and additional processing time for updating the latter with new message templates. Experimental results in [45] show however that the proposed approach performs better, in time and memory usage, than classic SAX [47] and DOM [84] parsers.

Detecting repeatable structures. An extension to the approach in [45] is provided in [71]. The authors in [71] introduce an improved automaton, able to consider repeatable structures in SOAP messages, which are not considered in [45]. That is because the automaton in [45] is string based and processes SOAP messages as a series of invariant and variable sections of string characters (i.e., byte sequences), whereas the new automaton in [71] considers the XML syntax (e.g., XML tagging) in its definition of states and state transitions. Detecting repeatable structures allows reducing the number of templates to be appended to the automaton, the latter becoming more expressive. Consequently this allows reducing memory and processing time needed for storing and updating the automaton, respectively, thus further enhancing parsing performance. Experimental results in [71] show improved memory usage and time performance w.r.t. the approach in [45], as well as a classic DOM parser [84].

Note that both methods described in [45], [71] have been developed in the context of WS-Security processing. Their main objective is therefore to improve security policy evaluation performance, by repetitively applying security rules only on those parts of SOAP messages which are different, processing the common parts only once. Yet, other methods aimed at improving security policy evaluation performance have been proposed in the context of SOAP message multicasting [6], [14] (which is discussed subsequently). Thus, for clearness of presentation, we disregard security aspects in this section, and provide a unified view of SOAP security policy evaluation performance, covering all related methods, in Section 3.3.

3.1.3 SOAP Deserialization

Deserialization is the process of converting XML messages to in-memory application objects, to be processed by the service executor. It can be viewed as the symmetric function of serialization. Recall that with serialization, the SOAP message is the target for recycling, whereas with deserialization, the target is an application object.

Approaches to improving SOAP deserialization performance build on the observation that memory object creation, based on SOAP XML messages, is an expensive task (mainly due to data type transformation—conversion from ASCII-based textual representation to in-memory numeric types, and the processing of the XML tree hierarchy [68]). Hence, the main idea is to avoid fully deserializing each incoming message, by exploiting already constructed objects which were deserialized previously. In other words, deserialization is differential and is only applied to those portions of the SOAP messages which have not been deserialized previously. To our knowledge, two studies have been developed in this direction, which we identify as *automaton-based* [68] and *checksum-based* [1]. We also stumbled on a more recent approach, *XML Screamer* [39], which promotes tight integration between software layers to avoid unnecessary deserialization processing.

Automaton-based. The authors in [68] propose an automaton-based approach, consisting of two main functions. The first consists in generating an automaton based

2. A FSA is usually modeled as $(P, \Sigma, p_s, F, \delta)$, where P is a set of states, Σ is the set of labels, $p_s \in P$ is the start state, $F \subset P$ is a set of final states, and $\delta : e \times R \rightarrow p$ is a transition function, where $e \in \Sigma, R$ is an expression over P and $p \in P$ [34]. Standard procedures for producing automatons and testing the membership of data instances w.r.t. automatons have been thoroughly studied in language theory [34].

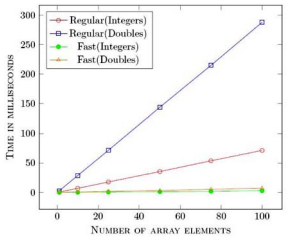


Fig. 7. Comparing regular deserialization and full differential deserialization time [1].

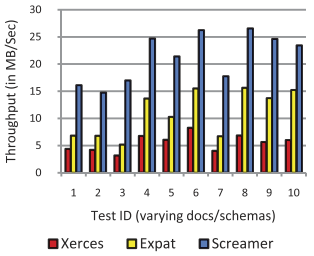


Fig. 8. Comparing XML Screamer [39] with traditional SOAP toolkits [5], [65].

on incoming SOAP messages (similarly to SOAP parsing approaches in [45], [70]), and then conducting deserialization in the usual way, creating a link between the defined automaton and the application object. The second function is to match an incoming message with the existing automaton, and if matched, return the linked application object to the SOAP engine after partially deserializing only the portions that differ from previous messages. The deserialization approach described in [68] could exploit the methods in [45], [70], [71] in building the deserialization automaton. Recall that SOAP parsing and deserialization are complementary operations, and allow SOAP message analysis (Fig. 1).

Checksum based. In [1], the authors propose to periodically checkpoint the state of the deserializer and to compute checksums[3] for portions of the incoming SOAP messages. In short, the deserializer runs in one of two modes: *regular* and *fast*. In regular mode, the deserializer processes SOAP message tags and contents as a normal SOAP deserializer, creating checkpoints and corresponding message portion checksums along the way. It switches to fast mode once it recognizes that the parser state is the same as one that has been saved in a checkpoint. In fast mode, the deserializer compares the sequence of checksums against those associated with the most recently received message. If the checksums match, then the already deserialized objects corresponding to the portions of the SOAP message at hand are exploited in a straightforward manner, without additional processing. Otherwise, when a checksum mismatch occurs, the system switches from fast to regular mode, where it processes SOAP tags and contents as a normal deserializer.

The authors discuss and experimentally validate the performance of their approach, considering the relation between 1) the amount of similarity between incoming messages, which otherwise determines the percentage of time the deserializer spends in fast mode, 2) how quickly the system can recognize the need to switch modes (from fast to regular, and vice versa), and 3) the overhead of creating checkpoints, and comparing checksums.

On one hand, if the new message is completely different from the previous one (which is the worst case scenario), the differential deserializer runs slightly slower than a normal deserializer since it does the same work, plus the added work of calculating and comparing checksums. On the other hand, when all checksums match, i.e., when the new message is identical to the previous one (which is the best case scenario), the cost of deserialization is replaced by that of computing and comparing checksums, which is significantly faster (speedups up to 41 times have been recorded by the authors, cf. Fig. 7). The authors also mention that using checksums to match portions of SOAP messages can be error prone, (since checksums themselves are not perfect by definition), but the possibility of changes going undetected is extremely low, in comparison with the substantial gain in performance.

Note that both methods in [1], [68] have not been evaluated w.r.t. each other, so as to compare their relative improvements in SOAP deserialization performance.

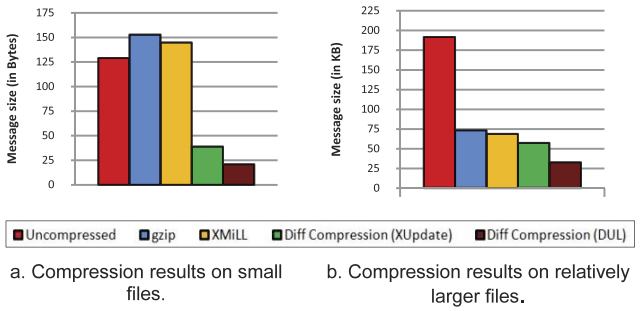
XML screamer. In a more recent study, the authors introduce XML Screamer [39], an optimized system providing tight integration across levels of software, combining: 1) schema-based XML parsing (character encoding, token extraction, and validation) and 2) deserialization, in one single processing layer (as opposed to separate layers—Fig. 6a), in order to avoid unnecessary data processing, copying (to/from memory), and data type transformations. The authors adopt a design principle requiring that each character and/or string in the input document be "visited" only once (if possible), so as to reduce repeatable scans of the same data and corresponding unnecessary overhead (e.g., tests to verify whether a character is an angle bracket ">", or an expected element name character, are performed only once following [39], whereas such tests are repeated multiple times—during parsing, and deserialization—in traditional XML/SOAP toolkits). Experimental results in [39] show that XML Screamer delivers from 2.3 to 5.3 times the throughput of traditional SOAP toolkits [5], [65] (cf. Fig. 8).

Note that the combination of software layer integration optimization [39], with similarity-based SOAP parsing [45], [70], [71] and deserialization [1], [68], has not been investigated to date. We believe this to be a very interesting research topic which could yield promising performance improvements in the near future.

3.2 Methods for Reducing Network Traffic

Another major drawback of using SOAP is its voracity for bandwidth, compared to competing solutions such as CORBA [54] and Java RMI [66]. Even though today's networks can be powerful enough to provide sufficient bandwidth, the latter remains crucial in several applications, namely in mobile computing [59] (e.g., wireless and cellular platforms), as well as sensor networks [81]. In this context, the problem of SOAP bandwidth reduction has

3. A checksum is a fixed size datum computed from a block of digital data (of fixed and/or variable size) to detect accidental errors that may occur during transmission or storage [50].



a. Compression results on small files.

b. Compression results on relatively larger files.

Fig. 9. Comparing the effectiveness of differential SOAP compression, in comparison with alternative text-based (gzip [20] and XML-based (XMILL [42]) techniques.

been investigated on two levels: 1) SOAP compression [81] in order to reduce message size prior to transmission, and 2) SOAP multicasting [58], [59] so as to optimize SOAP traffic traveling on the wire.

3.2.1 SOAP Compression

Various methods have been proposed for classic text and XML compression, namely gzip [20], WBXML [46], XMILL [42], and ESAX [9]. Text compression techniques (e.g., gzip) could be exploited with XML-based data (e.g., SOAP), since the latter are usually stored as ASCII-based text files. Nonetheless, a comparative study conducted in [81] showed that existing compression methods for classic XML documents might not always be appropriate in the context of SOAP. That is due to the fact that SOAP messages are of relatively smaller sizes (a few kilobytes), in comparison with other kinds of XML-based documents (e.g., SVG [85], MPEG-7 [52], etc., usually in the order of hundreds of kilobytes). Hence, existing compression methods might yield coding tables (i.e., tables mapping symbols to their bit codes) which require more space than the original SOAP messages themselves [81] (cf. Fig. 9a). In other words, compression results for large files are not necessarily transferable to small files, which is the case of SOAP messages. Following this observation, the authors in [81] propose a differential compression framework specifically aimed toward SOAP messages, exploiting the similarities between SOAP messages sent or received by the same service. The approach is based on XML differential encoding, which basically means that only the differences between SOAP messages should be sent over the wire. In brief, the authors exploit the WSDL schema definition to generate a SOAP message skeleton (the same would be available at the sender/receiver sides) describing the structure and tagging of corresponding SOAP messages (i.e., SOAP element/attribute names and corresponding parent/child relations, disregarding values). Consequently, only the differences between the SOAP message and the predefined skeleton are transmitted, along with corresponding SOAP message element/attribute values. The differences in structure and tagging, as well as element/attribute values, are consequently patched to the same skeleton at the receiver side in order to reconstruct the original message.

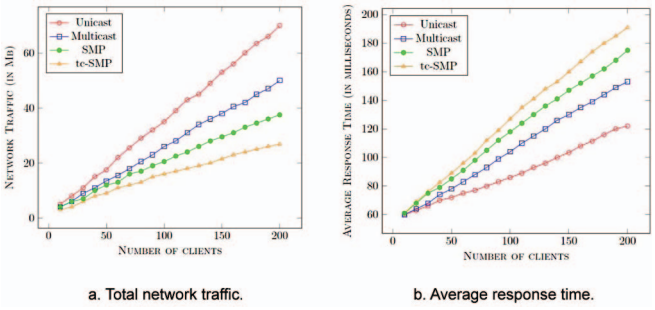
The authors argue that the effectiveness of their method depends on the degree of resemblance between the generated skeleton and the actual SOAP messages, which strictly influences compression rate: a higher resemblance yields smaller difference files, which in turn underlines a higher compression rate. They test two existing implementations of XML diff encoding tools (XUpate [41] and DUL [51]) in their experimental evaluation, proving that their approach yields better compression rates than existing XML-based compression techniques (Fig. 9).

The authors evaluate the execution speed of their approach, and show that it is slower that gzip [20], which introduces a major computational burden w.r.t. service execution time. In fact, gzip itself has been shown to be computationally expensive, exceeding the combined cost of XML serialization and data transport over LANs [28], [73]. Thus, while SOAP compression seems central in reducing network traffic, particularly when network bandwidth is very limited, its execution time underlines an equally serious drawback, which (to our knowledge) remains an open problem.

3.2.2 SOAP Multicasting

Another approach to reduce SOAP network bandwidth consumption would be to perform multicasting, a well-known technique that allows to conserve network bandwidth in applications where the same data are to be transmitted to multiple clients [86]. The main idea is to avoid sending replicated unicast messages over the wire by simultaneously delivering identical messages to a group of destinations, in a single aggregate message, only creating copies when the network links to the multiple destinations split [59], [86]. In general, multicasting would be effective when the number of receivers for a given service is sufficiently large and there is sufficient commonality in their interests, which happens to be the usual case with SOAP [59].

In this context, the authors in [59] put forward SMP, a Similarity-based SOAP Multicasting Protocol. It is built on top of SOAP unicast, and does not rely on low level (IP) multicast, in order to avoid complex network configurations at intermediate nodes (hubs and routers). In addition, SMP's main contribution and originality consists in grouping and transmitting together similar SOAP messages, and not only identical messages such as with traditional (IP) multicasting. An SMP message consists of two parts: SMP header and SMP body. The SMP header stores the addresses of destinations to which the messages should be sent. The SMP body is composed, in turn, of two parts: the *common* part section containing common values of the messages, and *distinctive* part section containing the different parts of each message. The aggregate SMP message is consequently encapsulated within the body of a classic SOAP message, which header encompasses the address of the next router along the path to all intended recipients. Each midway router would parse the SMP header and examine its routing table to decide the next hops for each client address. The router then separates client addresses into groups, splits the SMP message accordingly, and forwards the appropriate information to the next hop. The SMP message is split so that only relevant information (i.e., information destined to the designated clients) is sent down the stream path. During splitting, multiple copies of the input message are first produced, one for each downstream link that the router



a. Total network traffic. b. Average response time.

Fig. 10. Comparing network traffic and average response time with tc-SMP [58], SMP [59], traditional multicasting and unicast (reported from [58]).

connects to. The client list in each newly generated message header includes only those destinations that will be routed through that hop. *Distinctive* items in the original SMP message are analyzed and removed if they are not intended for clients beyond the next hop. The *common* part is obviously replicated in all outgoing messages. If the next hop connects directly to an end-point service, a standard SOAP unicast message is extracted from SMP and sent to the client service component.

The authors exploit an XML-based similarity measure [44] to quantify the resemblance between SOAP messages, so as to only aggregate the most similar ones. In addition, a dedicated indexing technique is also introduced to reduce SOAP message size by omitting full tag names and leveraging the organization of common and distinct parts in the SMP message.

In a subsequent study [58], the authors propose an enhanced routing protocol to further improve the performance of their SMP multicasting approach. In their original proposal [59], they used Dijkstra's Open Shortest Path First (OSPF) routing algorithm, which routes the message using the shortest path from a source to a destination. In their later study [58], the authors introduce traffic constrained SMP (tc-SMP) exploiting a similarity-based routing algorithm for transmitting messages following paths which maximize shared links between highly similar messages. This allows optimizing SMP network traffic distribution and thus further reducing overall network traffic (cf. Fig. 10a).

The authors also evaluate the performance penalty, in response time, of tc-SMP and SMP over traditional multicasting (simply multicasting identical messages) and unicast transmissions (cf. Fig. 10b). It is mainly due to the processing overhead required to measure the similarity between messages and aggregate similar ones (for both tc-SMP and SMP), as well as setting up the routing tree (in the case of tc-SMP). In short, results show that tc-SMP induces an average 3.5 to 5 times reduction in network traffic, compared to an average 2.5 times increase in average response time, which is considered acceptable by the authors, particularly in scenarios where bandwidth is limited such as with wireless and sensor networks.

In addition to network traffic optimization with classic SOAP message communications, differential SOAP multicasting (SMP) has been recently investigated in the context of secure SOAP message exchange [6], [14], in order to improve SOAP security policy evaluation performance.

```
1 <subject><role>BookingAgency</role></subject>
  <object>//BookingConfirmation/CreditCardNb</object>
  <rule>
     <Access>Allowed</Access>
     <Encrytption>AES</Encryption>
  </rule>

2 <subject><role>Customer</role></subject>
  <object>//BookingConfirmation/CreditCardNb</object>
  <rule>
      <Access>Denied</Access>
  </rule>
```

Fig. 11. Sample SOAP security policy rules (expressed in XML).

3.3 Improving SOAP Security Policy Evaluation Performance

In the past few years, the growing demand on mission-critical WS applications (e.g., financial transactions, stock market, etc.), has underlined an urgent need to provide trustworthy and secure services [48]. Nonetheless, security provision may introduce a substantial additional overhead, which has motivated researchers to start investigating the impact of security policy evaluation on WS performance.

WS-Security policy evaluation [19] consists in checking and verifying the access and usage security constraints defined on SOAP messages. It is performed both at the client and server application end points, each w.r.t. its own policy rules (cf. Fig. 1). A WS-Security policy usually underlines a set of rules (actions), specifying security constraints (e.g., authorizations, signatures, encryption, etc.) on particular SOAP elements and contents [6], [15]. A security policy rule can be characterized in a 3-tuple entity: *(subject, object, rule)*, where *subject* identifies the users to whom the rule applies, *object* identifies to which messages, or portions of messages, the corresponding policy rule applies, and *rule* specifies the actions (e.g., access, signature, or encryption [6]) authorized for the policy *subject* (user), on the policy *object*. Consider for instance the XML-based security rules in Fig. 11. The first rule allows service points with role "booking agency" to access encrypted credit card numbers of client requests, whereas the second rule denies subjects with role "customer" from accessing credit card numbers of other clients.

The need for evaluating WS-Security policies may introduce additional overhead, which in some cases dwarfs the latency of standard SOAP message processing. The results of [37] show that WS-Security policy evaluation can cause: 1) an increase in SOAP response time by a factor of 3 on average, 2) a substantial increase in network traffic (SOAP messages size) by a factor 6.9 in overall (regardless of the type of data, e.g., integer, double, string, etc., being exchanged). In this context, a few proposals have addressed the issue of improving SOAP security policy evaluation performance through improving other underlying techniques, namely parsing [45], [71], caching [76] and multicasting [6], [14]. Methods for improving SOAP parsing performance, e.g., [45], [71], consist in parsing and simultaneously processing the SOAP message for security evaluation, providing the deserializer module with the parsed output message (or parts of the message) the destination client is allowed to access. Simultaneous parsing and security policy evaluation is undertaken via automatons (cf. Section 3.1.2) which consider both the parser

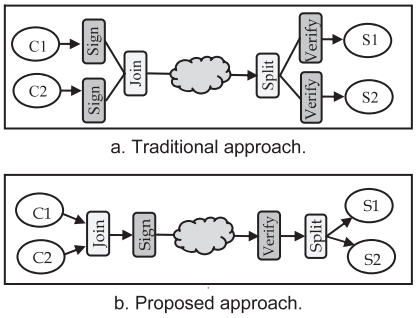


Fig. 12. Different scenarios to security policy evaluation.

context and security context, at the same time, for each incoming SOAP message. In other words, security-enabled parser automatons identify SOAP events (e.g., opening element tag, element text, etc.) which correspond to classic parsing events, as well as their corresponding policy rules (e.g., authorization, signature, or encryption schemes, allowing security processing), so as to process SOAP messages accordingly. These methods have been discussed in Section 3.1.2.

In [76], the authors investigate various techniques for WS-Security performance optimization, including digest-based caching, prehashing, and on-demand canonicalization. They propose to store the deserialized objects of digitally signed XML messages in cache, and then match the IDs and digest hash values of inbound elements to the objects in the cache, to be retrieved and utilized in case of a cache hit. Similarly, the digest hash value for each signed element in the outbound message is stored in the cache, along with its serialized content, so as to reserialize and rehash (in subsequent message exchanges) only those objects which are different. The authors show that the digest-caching and prehashing methods reduce overhead by a factor of 3 to 4 [76], at the expense of increased memory use (which they do not experimentally quantify). The authors also investigate on-demand canonicalization [75] (i.e., recanonicalizing contents only when the signature verification fails), and show that it effectively improves performance when more than 88 percent of the WS-Security messages need not be recanonicalized (otherwise, it might introduce additional overhead) [76].

Approaches in [6], [14] discuss and compare different scenarios where SOAP multicasting, namely SMP [59], could improve policy evaluation performance. In [14], the authors focus on a single sender/receiver SOAP message exchange scenario. They discuss how policy evaluation could be performed on an aggregate SMP message so as to only repeat policy evaluation processing on the SMP common part section once. Following the authors, security policy evaluation would be only repeated on those parts of the SOAP messages which are distinctive, inducing a substantial gain in processing time. In a subsequent study [6], the authors extend their discussion to multiple scenarios, with multiple senders/receivers, and investigate different approaches to improve SOAP signing/encryption through multicasting. They discuss different strategies for achieving optimal ordering of signing and multicasting operations, such as *Sign-Join-Split-Verify* and *Join-Sign-Split-Verify*. Fig. 12 depicts the classic approach, and the one ultimately adopted by the authors. They conclude that the best strategy, minimizing processing time and thus maximizing the gain in performance, would be to

1. first aggregate the SOAP messages (*Join*),
2. process the aggregate SMP message for signing/encryption (*Sign*),
3. transmit the signed/encrypted aggregate message to the receiver where it is first checked w.r.t. the latter's policy rules and processed for signature recognition and decryption (*Verify*), and then
4. decompose the SMP message to reconstruct the original SOAP messages (*Split*, cf. Fig. 12b).

Experimental results to quantify the actual gain in performance are not provided in [6], the corresponding prototypical implementation being under development. Indeed, research on the interplay between WS-Security policy evaluation and SOAP multicasting is still at a preliminary stage.

3.4 Parallelization and Hardware Approaches

Despite of the various kinds of software optimizations to improve SOAP and XML processing performance, no parser software can process input faster than its supporting hardware accesses data. With most current XML software toolkits, the maximum processing rate usually attains a best of tens of clock cycles per character [39] (a simple character-scanning loop runs at about 100 Mbytes/second on a 1 GHz Pentium processor, which amounts to 10 cycles/byte [39]), and that for many XML applications can result in processing rates of the order of hundreds of clock cycles per character (traditional parsers, e.g., [5], [65], perform in the range of 2.5-6 Mbytes of input per second or 160-400 cycles/byte, with a penalty of between 16× and 40× [39]). Recent benchmarking works in [32], [33] demonstrate that most existing implementations of WS do not scale well when the size of the SOAP/XML document being processed is increased. The authors in [32], [33] argue that most existing software toolkits are typically designed to process small-sized XML data sets, and thus are not suited for large-scale computing applications, e.g., [25], [62]. Hence, recent studies have attempted to alleviate the limitations of XML software performance bottlenecks by applying nontraditional parallel processor architectures, e.g., [8], [23], [30], [36], [55], [78]. On one hand, general-purpose (scalar) processors are characterized by the sequential nature of instruction execution, where instructions are selected based on their sequential memory addresses, conditions being evaluated one at a time. On the other hand, XML processing usually requires the evaluation of multiple conditions of various types that can occur simultaneously, namely during XML string and character parsing (e.g., verifying character integrity, whether an end tag matches a previously processed start tag, whether an attribute name is unique for a given element, and so on). Hence, the nature and frequency at which XML processing conditions occur result in a less predictable instruction flow, which calls for higher processing parallelism to improve performance [8], [78].

Parallel processing solutions can be roughly classified according to the level at which the hardware supports

parallelism [13], namely: bit-level, data-level, and instruction-level. In addition to single-node parallelism, a.k.a. microparallelism (achieved on a single computer system, with multiple processing units connected via the same bus and sharing the same memory), recent XML-related studies [23], [30], [31] have addressed cluster computing, a.k.a. macroparallelism (i.e., distributed computing on large data sets of computer clusters). In the following, we provide a concise overview of the most prominent XML and SOAP parallel processing methods in the literature, roughly organized following the type of parallelism they achieve.

Bit-level parallelism. It consists in increasing the processor word size (i.e., the amount of bits the processor can manipulate per cycle) and optimizing the inner processor architecture so as to reduce the number of instructions the processor must execute to perform operations on variables whose sizes are greater than the length of the word, and thus gain in processing rate. In this context, the authors in [78] introduce ZUXA, an XML accelerator engine which provides a processing model optimized for conditional execution in combination with dedicated instructions for XML character and string-processing functions. It is based on a programmable XML Finite State Machine technology, B-FSM, specifically tailored to provide high XML processing performance (a processing rate of one state transition per clock cycle), wide input and output vectors (with words of at least 64 bits for each transition), storage efficiency (to allow cost-efficient use of fast on-chip memory technologies), as well as full programmability (supporting fast incremental updates, allowing dynamic addition/removal of states and transitions), and scalability to tens of thousands of states and state transition rules. Related hardware solutions have been developed in the industrial arena, e.g., Datapower [16], which exploits Just-In-Time virtual machine technology [40] and ASICs customized for XML processing.

Data-level parallelism. Also known as Simple Instruction Multiple Data (SIMD), data-level parallelism describes computer systems with multiple processing elements that perform the same operation on multiple data simultaneously. An application that may take advantage of data-level parallelism is one where the same operation is being executed on a large number of data points, which is a common operation in many multimedia applications (e.g., image/video rendering and filtering), as well as in XML parsing and lexical analysis (e.g., reading input characters, and identifying string tokens). Parabix [8] is an XML parser designed to exploit the data-level parallelism capabilities of modern processors to deliver performance improvements over the traditional byte-at-a-time parsing technology. A byte-oriented character data are first transformed to a set of eight parallel bit streams, each stream comprising one bit per character code unit. Character validation, transcoding, and lexical item stream formation are all then carried out in parallel using bitwise logic and shifting operations. Byte-at-a-time scanning loops in the parser are replaced by bit scan loops that can advance by as many as 64 positions with a single instruction. Experimental results in [8] show that Parabix performs substantially better than traditional XML parsers: ranging from twice as fast as Expat [65], to an order of magnitude faster than Xerces [5].

Instruction-level parallelism. It is a processing paradigm which underlines the reordering and combination of instructions into instruction sets, which are then executed in parallel without affecting the result of the program. Instruction-level parallelism could be achieved in a number of ways to improve XML parsing performance, namely through 1) pipelining, and/or 2) multiprocessing (a.k.a. superscalar computing) [13]. On one hand, pipelining allows splitting the processing of an instruction into a series of independent steps, executed in parallel by different threads. On the other hand, multiprocessing allows the execution of more than one instruction during a clock cycle, by simultaneously dispatching multiple instructions to redundant execution units on the processor. Superscalar processors are identified as *multicore* when their constituent processing units are embedded in the same processor chip. While pipelining may provide significant speedup, XML software pipelining is often hard to implement due to synchronization and memory access bottlenecks, and to the difficulties of balancing the pipeline stages [55]. Hence, most studies in the context of XML and WS have focused on multiprocessing solutions. One prominent approach is the Meta-DFA project [43], [56], introducing a parallelization method that uses a two-stage DOM parser. The main idea is to divide the XML document into chunks, such as multiple threads would work on the chunks independently. The first stage consists in *preparsing* the XML document, to determine its logical tree structure (made of start and end tag node references). This structure is then used in a subsequent stage to divide the XML document such that the divisions between the chunks occur at well-defined points in the XML grammar. As the chunks are parsed, the results are then merged. In a following study [55], the authors investigate static partitioning and load balancing in order to minimize thread synchronization overhead. The authors in [56] show that their technique is effective and scales to large numbers of cores (up to 30 cores). Nonetheless, the authors discuss that while DOM-style parsing can be intuitive and convenient with applications requiring random access/manipulation of XML-based data, nonetheless, it can also be memory-intensive, both in the amount of memory used (to store the DOM structure), and in the high overhead of memory management [43], [55].

In a related project by Head et al., the Piximal toolkit [23], [30], [31] presents a parallelized SAX parsing solution, focusing on a different class of applications than the DOM-based Meta-DFA project, tailored around event-streams and fast sequential access of XML-based data. Piximal conducts parsing dynamically, and generates as output a sequence of SAX events. It results in a larger number of parser states and state transitions, underlining more opportunities for parallelization optimization, and scaling well with increasing numbers of processing cores. Experimental results demonstrate that the level of speedup obtainable using Piximal's microlevel parallelization techniques can be limited due to: 1) memory bandwidth, which could become a bottleneck [31], and 2) the amount of computation required to parse the input, which would induce little performance gain if the computation required is small in comparison to the time required to access the bytes of the input in memory [23].

Hence, the authors in [23], [30], [31] also address macrolevel parallelism. They investigate the distributed processing of large-scale XML data stored in a cluster, by applying Google's MapReduce processing paradigm [18]. The simplicity and robustness of the MapReduce model, as well as its relaxed synchronization constraints, tend to work favorably for large-scale XML data sets and WS computing environments [23]. Experimental results on Piximal's macrolevel parallelization technique show that securing additional resources for each thread by distributing the workload to a cluster of machines using MapReduce can increase performance [23], [30], [31]. Nonetheless, the authors also show that if not enough processing is taking place on each cluster, the latter would be burdened with redundancy checks and network traffic for just small chunks of input. The authors conclude that when computation is not sufficient enough to offset communication latencies due to the number of running computers, a single node, which minimally suffers from the same condition, would perform better than a cluster of computers.

4 Ongoing Challenges

Despite the wide array of techniques proposed to enhance SOAP processing performance, yet various challenges and limitations remain unaddressed. Three major hurdles remain to the wide adoption of similarity-based techniques.

First, while similarity-based methods have been shown in many cases to produce a significant gain in speed-up when many similar messages are involved [69], as well as a noticeable reduction in network traffic [58], nonetheless, similarity computations can sometimes introduce additional overhead on their own (as shown with SOAP compression [81] and multicasting [58], [59]), especially when the SOAP messages being processed are fairly different (i.e., not similar to the documents processed before). Hence, a comprehensive empirical analysis addressing the tradeoff between: 1) the amount of additional processing overhead, and 2) the amount of processing time and network traffic reduction, induced by similarity-based approaches, is required in order to identify and better understand each method's optimum usage constraints (e.g., percentage of similar SOAP messages, amount of inner message similarities, number of messages, and so on).

Second, interference and synergy between different similarity-based techniques is not yet completely understood. One can realize that the various techniques covered in the paper are not mutually exclusive, but are rather complementary. For instance, similarity-based methods to SOAP serialization, parsing, and deserialization could very well exploit XML parallel processing architectures so as to better improve their clock cycle character processing rates. In addition, software-based methods could make use of tight integration architectures, such as in [39], so as to avoid repeated/unnecessary data processing, copying to/from memory buffers, and expensive data-type transformations (ASCII/UTF to in-memory types, and vice versa). In this context, recent efforts have been made toward combining efficient SOAP multicasting, on one hand, with fast security policy evaluation on the other hand (as discussed in Section 3.3). Nonetheless, corresponding techniques are still in their preliminary stages. Comparative theoretical and experimental studies are required to better understand the interplay and actual gain in performance between WS-Security policy evaluation and SOAP multicasting.

Third, and perhaps more importantly, interference may arise between SOAP similarity-based multicasting described in this paper and attempts at boosting SOAP performance via custom protocol bindings.

Several commercial SOAP engines, including Noemax and Sun Metro, are based on custom protocol bindings that exploit information on the XML stream data to improve the performance of transport layer protocols. In these implementations of SOAP, HTTP binding has been dropped altogether in favor of an integrated SOAP/TCP transport where each message sent during a communication session is accompanied only by new entries (if any) to the XML Infoset vocabulary [67]. The vocabulary is a table that associates string values with identifiers. In this context, the technique used to reduce the size of the XML text encoding is to enter string values (such as XML markup) in the vocabulary and substitute all occurrences of these string values in the document with their corresponding identifier. This vocabulary-based technique is sometime coupled with GZIP compression [20] of messages, and is a major competitor of similarity-based multicasting when nonstandard protocol bindings are acceptable—e.g., on clusters or grids [80] when no firewall traversal is required. However, the effect of using similarity-based SOAP multicasting in the context of custom SOAP/TCP bindings is still largely unexplored, but, great potential have been shown by enhancements in the underlying HTTP transport protocol (particularly in the context of HTTP 1.1) to reduce the overhead of creating a new connection for every SOAP message (with persistent connections and message chunking [12], [28]), as well as by ongoing investigations in XML-based binary encodings for SOAP [57], [64], [83]. In short, techniques to SOAP performance enhancement are yet to be further improved and perfected, promising further performance improvements in the near future, which presents an overwhelming motivation to do research in this field.

5 Conclusion

In this survey paper, we have given an overview of current research related to SOAP processing performance enhancement, focusing on similarity-based approaches, as well as WS-Security optimizations, and XML parallel processing architectures. We provide a concise, yet comprehensive review of how different techniques have been exploited to enhance SOAP performance in almost every phase of SOAP processing, ranging over message parsing [45], [70], [71], serialization [4], [21], de-serialization [1], [68], compression [81], multicasting [6], [58], [59], security evaluation [6], [14], and data/instruction-level processing [8], [55], [78] (cf. Tables 1 and 2). Most methods build on the observation that SOAP message exchange usually involves highly similar messages (messages created by the same implementation usually have the same structure, and those sent from a server to multiple clients tend to show similarities in structure and content). The main idea is then to identify the common parts of SOAP messages, to be processed once, only repeating the processing for parts

TABLE 1
Characteristics of Existing (Similarity-Based) SOAP Performance Enhancement Approaches

Performance	SOAP Processing	Approach	Features
Reducing *Response time* and increasing *Throughput*	Serialization	Abu-Ghazaleh *et al.* [4]	bSOAP, differential serializer: - DUTs (Data Update Tracking), tracking between in-memory data, and their serialized representations. - *Dirty bits* to identify fields whose values changed, recognizing parts to be reused.
		Abu-Ghazaleh *et al.* [2, 3]	bSOAP buffer management: - Padding and chunk overlaying to allow on-the-fly message expansion.
		Devaram and Andersen [21]	Client-side SOAP message caching: - Indexing structures to detect correspondences between cached and outgoing messages. - Does not address partial structural matches (only caches identical structures).
	Parsing	Zhang and Van Engelen [87]	TDX: Table Driven XML parsing - Combining the lexical analysis and validation - Pre-recording parser states as grammar productions in tabular form, and breaking up the SOAP message into a token stream
		Takeuchi *et al.* [70]	T-SOAP, template-based differential parser: - Predefined template, modeled via a finite state automaton (FSA). - Identification of invariant/variable tag parts in the SOAP messages. - Variable parts are only parsed.
		Makino *et al.* [45]	Multi-template differential parser: - Appending new templates to the FSA, - More flexible than T-SOAP [70] (bound to one single template), - Requires more memory that T-SOAP.
		Teraguchi et al. [71]	Detecting repeatable structures: - Improved XML-based automaton, to consider repeatable structures in SOAP messages, in comparison with string-based ones in [45, 70], - More expressive automaton, reducing memory and time consumption.
		Kostoulas et al. [39]	XML Screamer: - Tight integration across software levels, - Combines parsing and de-serialization in one layer, so as to avoid unnecessary data processing, copying (to/from memory), and data-type transformation.
	De-Serialization	Suzumura *et al.* [68]	Automaton-based approach: - Classic de-serialization and automaton creation, - Matching messages to automaton and only de-serialising those different portions (could complement parsers in [45, 70, 71])
		Abu-Ghazaleh and Lewis [1]	Checksum-based approach: - Regular mode, periodically checkpointing de-serialiser state, - Compare checkpoints, and switches to fast mode, when parser state is similar to state saved in previous checkpoint, - Checksumming is fast, yet error prone.
		Makino *et al.* [45], Teraguchi *et al.* [71]	Security-based SOAP message parsing: - Automatons to consider both the parser context and security context, - Identifying SOAP events (tags, text…) and their corresponding policy rules (authorizations, signatures…)
Reducing *Network traffic*	Security Policy Evaluation	Damiani and Marrara [14]	Security-based SOAP multicasting: - Single sender-receiver scenario, - Policy evaluation on aggregate SMP message [59], - Policy evaluation repeated only on those parts of SOAP messages which are different.
		Azzini *et al.* [6]	Security-based SOAP multicasting: - Multiple senders/receivers scenario - Different approaches to improve SOAP signature/encryption (*Sign-Join-Split-Verify*, *Join-Sign-Split-Verify*…), - Best strategy is *join-sign-verify-split*.
		Van Engelen and Zhang [76]	WS-Security performance optimization: - Digest-based cashing, storing and using de-serialized digitally signed objects, - Pre-hashing, storing and using digest values of digitally signed objects, - On-demand canonicalization, re-canonicalizating contents only when the signature verification fails.
	Com-pression	Werner *et al.* [81]	Differential compression: - XML differential encoding (tree edit distance), - Identifying differences between SOAP messages and predefined WSDL-based SOAP templates, - Only differences are transmitted, - Patching differences with the same skeleton at the receiver side, to reconstruct the original message.
	Multicasting	Phan *et al.* [59]	SMP, Similarity-based SOAP Multicasting Protocol: - Built on top of IP unicast (avoiding complex network configurations), - Grouping and transmitting together similar SOAP messages (not only identical ones such as with classic multicasting), - SMP message encapsulated in classic SOAP message, with common and distinct parts.
		Phan *et al.* [58]	tc-SMP, traffic constrained SMP: - Enhanced routing protocol for transmitting messages following paths which maximize shared links between highly similar messages, - Reducing traffic in comparison with the OSPF-based SMP [59].

TABLE 2
Characteristics of SOAP and XML-Based Parallellization and Hardware Related Approaches

Performance	SOAP Processing	Approach	Features
Micro-Parallelism	Bit-level	Van Lunteren *et al.* [78]	ZUXA XML Accelerator Engine: - Increasing processor word size, i.e., the amount of bits the processor can manipulate per cycle, - Optimized for conditional execution with dedicated instructions for XML character processing, - Based on a programmable State Machine technology, B-FSM, tailored to provide high XML processing performance, wide input/output vectors, storage efficiency, as well as full programmability.
	Data-level	Cameron *et al.* [8]	PARABIX: - Designed to exploit the data-level parallelism, - Byte-oriented character data is first transformed to a set of 8 parallel bit streams, each stream comprising one bit per character code unit, - Character validation, transcoding, and lexical item stream formation are all then carried out in parallel using bitwise logic and shifting operations.
	Instruction-level	Pan et al. [43, 56]	Meta-DFA: - Two-stage DOM parser : i) *pre-parsing* to determine its logical XML tree structure, and then ii) dividing the XML document such that the divisions between the chunks occur at well-defined points in the XML grammar, - Merges results as the chunks are parsed, - Exploits static partitioning and load-balancing to minimize thread synchronization overhead, - Considerably scalable (up to of 30 cores).
		Head *et al.* [23, 30, 31]	Piximal: - Introduces a parallelized SAX parser, tailored around event-stream XML data (different class of applications than the DOM-based Meta-DFA), - Larger number of parser states, thus more opportunity for parallelization and scalability with increasing numbers of cores (in comparison with Meta-DFA), - Speed-up could be limited due to: i) memory bandwidth, and ii) the amount of computation required to parse the input (if the computation required is small in comparison to the time required to access the bytes of the input in memory).
Macro-Parallelism	Head *et al.* [23, 30, 31]		Piximal, with cluster computing: - Exploits distributed processing of large-scale XML data stored in a cluster, by applying Google's MapReduce processing paradigm [18], - Introduces relaxed synchronization constraints, which tend to work favorably for large-scale XML data sets and WS computing environments, - Experiments show that macro-parallelism can increase performance (in comparison with micro-parallelism). Yet, if not enough processing is taking place on each cluster, the latter would be burdened with redundancy checks and network traffic for just small chunks of input, and could perform worst than a single node, - Examining computation costs to determine the best computation strategy.

which are different, and substantially reducing SOAP processing overhead. Other approaches investigate non-traditional processor architectures, including micro- and macrolevel parallel processing solutions, so as further increase the processing rates of SOAP/XML software toolkits. In addition, we have also discussed some of the main challenges and possible future research directions, covering SOAP software and parallel architecture integration, as well as custom protocol bindings.

We hope that the unified presentation of SOAP-related performance enhancement techniques in this paper will foster further research on the subject.

ACKNOWLEDGMENTS

This work was supported in part by the Fondazione Cariplo 2007 Capitale Umano di Eccellenza research grant and the Japan Society for the Promotion of Science (JSPS) 2010 research fellowship no. PE10006.

REFERENCES

[1] N. Abu-Ghazaleh and M.J. Lewis, "Differential Deserialization for Optimized SOAP Performance," *Proc. ACM/IEEE Conf. Supercomputing*, pp. 21-31, 2005.

[2] N. Abu-Ghazaleh, M.J. Lewis, and M. Govindaraju, "Performance of Dynamic Resizing of Message Fields for Differential Serialization of SOAP Messages," *Proc. Int'l Symp. Web Services and Applications*, pp. 783-789, 2004.

[3] N. Abu-Ghazaleh, M. Govindaraju, and M.J. Lewis, "Optimizing Performance of Web Services with Chunk-Overlaying and Pipelined-Send," *Proc. Int'l Conf. Internet Computing (ICIC)*, pp. 482-485, 2004.

[4] N. Abu-Ghazaleh, M.J. Lewis, and M. Govindaraju, "Differential Serialization for Optimized SOAP Performance," *Proc. 13th Int'l Symp. High Performance Distributed Computing (HPDC '04)*, pp. 55-64, 2004.

[5] Apache Foundation, *Xerces XML Parser*, http://xerces.apache. org, Nov. 2010.

[6] A. Azzini, S. Marrara, M. Jensen, and J. Schwenk, "Extending the Similarity-Based XML Multicast Approach with Digital Signatures," *Proc. ACM Workshop Secure Web Services (SWS '09)*, pp. 45-52, 2009.

[7] T. Bray, J. Paoli, C. Sperberg-McQueen, Y. Mailer, and F. Yergeau, *Extensible Markup Language (XML) 1.0*, fifth ed., W3C, http://www.w3.org/TR/REC-xml, Nov. 2008.

[8] R.D. Cameron, K.S. Herdy, and D. Lin, "PARABIX: High Performance XML Parsing Using Parallel Bit Stream Technology," *Proc. Conf. Center for Advanced Studies on Collaborative Research: Meeting of Minds (CASCON '08)*, vol. 17, pp. 222-235, 2008.

[9] J. Cheney, "Compressing XML with Multiplexed Hierarchical PPM Models," *Proc. Data Compression Conf.*, pp. 163-173, 2001.

[10] R. Chinnici, J.J. Moreau, A. Ryman, and S. Weerawarana, *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language*, W3C recommendation, http://www. w3.org/TR/wsdl20, Aug. 2009.

[11] K. Chiu and W. Lu, "A Compiler-Based Approach to Schema-Specific XML Parsing," *Proc. Workshop High Performance XML Processing*, 2004.

[12] K. Chiu, M. Govindaraju, and R. Bramley, "Investigating the Limits of SOAP Performance for Scientific Computing," *Proc. ACM Int'l Symp. High Performance Distributed Computing (HPDC)*, pp. 246-254, 2002.

[13] D.E. Culler, J.P. Singh, and A. Gupta, *Parallel Computer Architecture - A Hardware/Software Approach,* p. 1100. Morgan Kaufmann, 1999.

[14] E. Damiani and S. Marrara, "Efficient SOAP Message Exchange and Evaluation Through XML Similarity," *Proc. ACM Workshop Secure Web Services (SWS '08),* pp. 29-36, 2008.

[15] E. Damiani, V. De Capitani di Vimercati, S. Paraboschi, and P. Samarati, "Securing SOAP E-Services," *Int'l J. Information Security,* vol. 1, pp. 100-115, 2001.

[16] Datapower, http://www.datapower.com, Nov. 2010.

[17] D. Davis and M. Parashar, "Latency Performance of SOAP Implementations," *Proc. IEEE/ACM Second Int'l Symp. Cluster Computing and the Grid,* pp. 407-412, 2002.

[18] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Comm. ACM,* vol. 51, no. 1, pp. 107-113, 2008.

[19] G. Della-Libera et al., *Web Services Security Policy Language (WS-SecurityPolicy),* V1.1 Specification, http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-secpol/ws-secpol.pdf, July 2005.

[20] L.P. Deutsch, "GZIP File Format Specification Version 4.3," IETF RFC 1952, 1996.

[21] K. Devaram and D. Andersen, "SOAP Optimization via Parameterized Client-Side Caching," *Proc. IEEE/ACM Second Int'l Symp. Cluster Computing and the Grid (CCGRID '02),* pp. 439-312, 2002.

[22] R. Elfwing, U. Paulsson, and L. Lundberg, "Performance of SOAP in Web Service Environment Compared to CORBA," *Proc. Ninth Asia-Pacific Software Eng. Conf. (APSEC '02),* pp. 84-94, 2002.

[23] Z. Fadika, M.R. Head, and M. Govindaraju, "Parallel and Distributed Approach for Processing Large-Scale XML Data Sets," *Proc. IEEE/ACM 10th Int'l Conf. Grid Computing (GRID '09),* pp. 105-112, 2009.

[24] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext Transfer Protocol - HTTP/1.1," IETF RFC 2616, http://www.ietf.org/rfc/rfc2616.txt, May 2010.

[25] D. Gannon, S. Krishnan, L. Fang, G. Kandaswamy, Y. Simmhan, and A. Slominski, "On Building Parallel and Grid Applications: Component Technology and Distributed Services," *Proc. Second Int'l Workshop Challenges of Large Applications in Distributed Environments (CLADE '04),* pp. 44-51, 2004.

[26] J.Z. Gao, H.S.J. Tsao, and Y. Wu, *Testing and Quality Assurance for Component-Based Software,* p. 439. Artech House, 2003.

[27] A. Ginige and S. Murugesan, "Web Engineering: An Introduction," *IEEE Multimedia,* vol. 8, no. 1, pp. 14-17, Jan.-Mar. 2001.

[28] M. Govindaraju, A. Slominski, K. Chiu, P. Liu, R. Van Engelen, and M.J. Lewis, "Toward Characterizing the Performance of SOAP Toolkits," *Proc. IEEE/ACM Fifth Int'l Workshop Grid Computing (GRID '04),* pp. 365-372, 2004.

[29] M. Gudgin et al., *Simple Object Access Protocol 1.1,* W3C recommendation, http://www.w3.org/TR/SOAP, June 2003.

[30] M.R. Head and M. Govindaraju, "Parallel Processing of Large-Scale XML-Based Application Documents on Multi-core Architectures with PiXiMaL," *Proc. IEEE Fourth Int'l Conf. E-Science,* pp. 261-268, 2008.

[31] M.R. Head and M. Govindaraju, "Performance Enhancement with Speculative Execution Based Parallelism for Processing Large-Scale XML-Based Application Data," *Proc. Int'l Symp. High Performance Distributed Computing (HPDC '09),* pp. 21-30, 2009.

[32] M.R. Head, M. Govindaraju, A. Slominski, P. Liu, N. Abu-Ghazaleh, R. Van Engelen, K. Chiu, and M.J. Lewis, "A Benchmark Suite for SOAP-Based Communication in Grid Web Services," *Proc. ACM/IEEE Conf. Supercomputing (SC '05),* p. 19, 2005.

[33] M.R. Head, M. Govindaraju, R. Van Engelen, and W. Zhang, "Benchmarking XML Processors for Applications in Grid Web Services," *Proc. ACM/IEEE Conf. Supercomputing (SC '06),* p. 30, 2006.

[34] J.E. Hopcroft, R. Motwani, and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation,* second ed. Addison Wesley, 2001.

[35] M. Horstmann and M. Kirtland, *DCOM Architecture,* Microsoft MSDN, http://msdn.microsoft.com/en-us/library/ms809311.aspx, Jan. 2010.

[36] Intel Corporation, *Intel Core i7-800 Processor Series and the Intel Core i5-700 Processor Series,* download.intel.com/products/processor/corei7/319724.pdf, Nov. 2010.

[37] M.B. Juric, I. Rozman, B. Brumen, M. Colnaric, and M. Hericko, "Comparison of Performance of Web Services, WS-Security, RMI, and RMI—SSL," *J. Systems and Software,* vol. 79, no. 5, pp. 689-700, 2006.

[38] C. Kohlhoff and R. Steele, "Evaluating SOAP for High Performance Business Applications: Real-Time Trading Systems," *Proc. World Wide Web (WWW) Conf.,* 2003.

[39] M.G. Kostoulas, M. Matsa, N. Mendelsohn, E. Perkins, A. Heifets, and M. Mercaldi, "XML Screamer: An Integrated Approach to High Performance XML Parsing, Validation and Deserialization," *Proc. 15th Int'l Conf. World Wide Web (WWW '06),* pp. 93-102, 2006.

[40] E. Kuznetsov, "Method and Apparatus of Data Exchange Using Runtime Code Generator and Translator," US Patent 6772413, 2004.

[41] A. Laux and L. Martin, *XML:DB Initiative: XUpdate,* Working Draft, 2000.

[42] H. Liefke and D. Suciu, "XMill: An Efficient Compressor for XML Data," Technical Report MSCIS-99-26, Univ. of Pennsylvania, 2000.

[43] W. Lu, K. Chiu, and Y. Pan, "A Parallel Approach to XML Parsing," *Proc. IEEE/ACM Seventh Int'l Conf. Grid Computing (Grid '06),* pp. 223-230, 2006.

[44] Y. Ma and R. Chbeir, "Content and Structure Based Approach for XML Similarity," *Proc. Int'l Conf. Computer and Information Technology (ICCIT),* pp. 136-140, 2005.

[45] S. Makino, M. Tatsubori, K. Tamura, and Y. Nakamura, "Improving WS-Security Performance with a Template-Based Approach," *Proc. IEEE Int'l Conf. Web Services (ICWS '05),* pp. 581-588, 2005.

[46] B. Martin and B. Jano, *WAP Binary XML Content Format,* W3C recommendation, Feb. 2010.

[47] D. Megginson et al., *The Simple API for XML,* http://www.megginson.com/SAX, Feb. 2010.

[48] D.A. Menascé and V.A.F. Almeida, *Capacity Planning for Web Services - Metrics, Models and Methods,* p. 556. Prentice Hall, 2002.

[49] D.A. Menascé, V.A.F. Almeida, and L.W.L. Dowdy, *Capacity Planning and Performance Modeling: From Mainframes to Client-Server Systems.* Prentice Hall, 1994.

[50] T.K. Moon, *Error Correction Coding: Math. Methods and Algorithms,* p. 756. John Wiley & Sons, 2005.

[51] A. Mouat, "XML Diff and Patch Utilities," CS4 dissertation, Heriot-Watt Univ., 2002.

[52] Moving Pictures Experts Group, *MPEG-7,* http://www.chiariglione.org/mpeg/standards/mpeg-7, June 2010.

[53] M.L. Noga, S. Schott, and W. Lowe, "Lazy XML Processing," *Proc. ACM Symp. Document Eng. (DocEng '02),* 2002.

[54] Object Management Group, *The Common Object Request Broker: Architecture and Specification,* Version 3.0.3, http://www.omg.org/technology/documents/formal/corba_2.htm, Jan. 2010.

[55] Y. Pan, W. Lu, Y. Zhang, and K. Chiu, "A Static Load-Balancing Scheme for Parallel XML Parsing on Multicore CPUs," *Proc. IEEE Seventh Int'l Symp. Cluster Computing and the Grid (CCGrid '07),* pp. 351-362, 2007.

[56] Y. Pan, Y. Zhang, K. Chiu, and W. Lu, "Parallel XML Parsing Using Meta-DFAs," *Proc. IEEE Third Int'l Conf. eScience and Grid Computing (eScience '07),* pp. 237-244, 2007.

[57] P. Sandoz et al., *Fast Web Services,* java.sun.com/developer/technicalArticles/WebServices/fastWS, 2003.

[58] K.A. Phan, P. Bertok, A. Fry, and C. Ryan, "Minimal Traffic-Constrained Similarity-Based SOAP Multicast Routing Protocol," *Proc. OTM Confederated Int'l Conf.,* pp. 558-576, 2009.

[59] K.A. Phan, Z. Tari, and P. Bertok, "Similarity-Based SOAP Multicast Protocol to Reduce Bandwidth and Latency in Web Services," *IEEE Trans. Services Computing,* vol 1, no 2, pp. 88-103, Apr.-June 2008.

[60] D. Rud, A. Schmietendorf, and R. Dumke, "Product Metrics for Service-Oriented Infrastructures," *Proc. Int'l Workshop Software Metrics and DASMA Software Metrik Kongress (IWSM/MetriKon '06),* A. Abran, M. Bundschuh, G. Buren, R. Dumke, eds., pp. 161-174, 2006.

[61] A. Sahai and V. Machiraju, "Enabling fo the Ubiquitous E-Services Vision on the Internet," Technical Report HPL-2001-5, Hewlett-Packard Laboratories, 2001.

[62] G. Singh, S. Bharathi, A. Chervenak, E. Deelman, C. Kesselman, M. Manohar, S. Patil, and L. Pearlman, "A Metadata Catalog Service for Data Intensive Applications," *Proc. ACM/IEEE Conf. Supercomputing,* p. 33, 2003.

[63] A. Slominski, *XSOAP,* http://www.extreme.indiana.edu/xgws/xsoap, 2004.
[64] SourceForge.NET, *XML Binary Information Set (XBIS),* http://xbis.sourceforge.net, 2010.
[65] SourceForge.NET, *The Expat XML Parser,* http://expat.sourceforge.net, 2010.
[66] Sun, *Java Remote Message Invocation (RMI),* http://java.sun.com/j2se/1.5.0/docs/guide/rmi, Jan. 2010.
[67] Sun Microsystem, *SOAP/TCP Specification v.1.0.* http://java.sun.com/webservices/reference/apis-docs/soap-tcp-v1.0.pdf, May 2007.
[68] T. Suzumura, T. Takase, and M. Tatsubori, "Optimizing Web Services Performance by Differential Deserialization," *Proc. IEEE Int'l Conf. Web Services (ICWS '05),* vol. 1, pp. 185-192, 2005.
[69] T. Takase, H. Miyashita, M. Tatsubori, and T. Suzumura, "An Adaptative, Fast and Safe XML Parser Based on Byte Sequence Memorization," *Proc. World Wide Web Conf.,* pp. 692-701, 2005.
[70] Y. Takeuchi, T. Okamoto, K. Yokoyama, and S. Matsuda, "A Differential-Analysis Approach for Improving SOAP Processing Performance," *Proc. IEEE Int'l Conf. e-Technology, e-Commerce and e-Service (EEE '05),* pp. 472-479, 2005.
[71] M. Teraguchi, S. Makino, K. Ueno, and H.V. Chung, "Optimized Web Services Security Performance with Differential Parsing," *Proc. Fourth Int'l Conf. Service-Oriented Computing (ICSOC '06),* pp. 277-288, 2006.
[72] H.L. Truong, S. Dustdar, and T. Fahringer, "Performance Metrics and Ontologies for Grid Workflows," *Future Generation Computer Systems,* vol. 23, pp. 760-772, 2007.
[73] R. Van Engelen, "Pushing the SOAP Envelope with Web Services for Scientic Computing," *Proc. Int'l Conf. Web Services (ICWS),* pp. 346-352, 2003.
[74] R. Van Engelen, "Constructing Finite State Automata for High Performance XML Web Services," *Proc. Int'l Conf. Internet Computing (ICIC),* pp. 975-981, 2004.
[75] R. Van Engelen, "A Framework for Service-Oriented Computing with C and C++ Web Service Components," *ACM Trans. Internet Technology,* vol. 8, no. 3, pp. 1-25, 2008.
[76] R. Van Engelen and W. Zhang, "An Overview and Evaluation of Web Services Security Performance Optimizations," *Proc. IEEE Int'l Conf. Web Services (ICWS),* pp. 137-144, 2008.
[77] R.A. Van Engelen and K.A. Gallivan, "The gSOAP Toolkit for Web Services and Peer-to-Peer Computing Networks," *Proc. IEEE Second Int'l Symp. Cluster Computing and the Grid (CCGrid '02),* pp. 128-135, 2002.
[78] J. Van Lunteren, J. Bostian, B. Carey, T. Engbersen, and C. Larsson, "XML Accelerator Engine," *Proc. First Int'l Workshop High Performance XML Processing,* 2004.
[79] J. Viega, M. Messier, and P. Chandra, *Network Security with OpenSSL.* O'Reilly, 2002.
[80] N. Wang, M. Welzl, and L. Zhang, "A High Performance SOAP Engine for Grid Computing," *Social Informatics and Telecomm. Eng.,* vol. 2, pp. 1-8, 2009.
[81] C. Werner, C. Buschmann, and S. Fischer, "WSDL-Driven SOAP Compression," *Int'l J. Web Services Research,* vol. 2, no. 1, pp. 18-35, 2005.
[82] Word Wide Web Consortium, *SOAP Version 1.2,* W3C recommendation, second ed., http://www.w3.org/TR/soap, Feb. 2010.
[83] World Wide Web Consortium, *XML Binary Characterization Working Group,* http://www.w3.org/XML/Binary, 2010.
[84] World Wide Web Consortium, *The Document Object Model,* http://www.w3.org/DOM, May 2009.
[85] World Wide Web Consortium, *Scalable Vector Graphics (SVG),* http://www.w3.org/Graphics/SVG, 2009.
[86] B. Zhang, S. Jamin, and L. Zhang, "Host Multicast: A Framework for Delivering Multicast to End Users," *Proc. IEEE INFOCOM,* pp. 1366-1375, 2002.
[87] W. Zhang and R.A. Van Engelen, "A Table-Driven Streaming XML Parsing Methodology for High-Performance Web Services," *Proc. IEEE Int'l Conf. Web Services (ICWS '06),* pp. 197-204, 2006.

Joe M. Tekli received the master's of engineering degree in telecommunications from the Antonine University, Lebanon, in July 2005, and the research master's degree in computer science from the University of Bourgogne, France, in July 2006, both with honors. He received the PhD degree in computer science from the University of Bourgogne, LE2I CNRS, France, in October 2009 with highest honors. He has been a visiting researcher in the Department of Computer Science and Statistics (ICMC), University of Sao Paulo, Brazil, since 2011, and has completed two postdoctoral missions in the Department of Science, University of Shizuoka, Japan (Spring 2010), and in the Department of Information Technology, University of Milan, Italy (2009). He has been awarded various postdoctoral fellowships from the FAPESP (Brazil), JSPS (Japan), and Fondazione Cariplo (Italy). He was also awarded a PhD Fellowship from the Ministry of Education (France), and a Masters Scholarship from the AUF (France). His research activities cover XML, web services, data semantics, data mining, and multimedia retrieval. He is an organizing member of various international conferences such as SITIS, ICDIM, MEDES, and ACM SAC 2006. His research results have been published in various international journals and conferences (e.g., *CS Review*, *WWW Journal*, *JWS*, ER, IEEE ICWS, SBBD, WISE). He is a member of the IEEE and the ACM SIGAPP French Chapter.

Ernesto Damiani is a professor at the Università degli Studi di Milano and director of that university's PhD program in computer science. He has held visiting positions at a number of international institutions. He has done extensive research on advanced network infrastructure and protocols, taking part in the design and deployment of secure high-performance networking environments. His areas of interest include business process representation, web services security, processing of semi and unstructured information, and semantics-aware content engineering for multimedia. He is interested in models and platforms supporting open source development. He serves in all capacities on many congress, conference, and workshop committees. In 2008, he was nominated as an ACM distinguished scientist and received the Chester Hall Award from the IEEE Society on Consumer Electronics. His webpage is http://www.dti.unimi.it/~damiani. He is a senior member of the IEEE and the IEEE Computer Society.

Richard Chbeir received the PhD degree in computer science from the University of INSA-FRANCE in 2001. He is currently an associate professor in the Computer Science Department of Bourgogne University, Dijon-France. His research interests are in the areas of distributed multimedia database management, XML similarity and rewriting, spatiotemporal applications, indexing methods, multimedia access control models, security, and watermarking. He has published more than 80 peer-reviewed publications in international journals and books (*IEEE Transactions on SMC, Information Systems, Journal on Data Semantics, Journal of Systems Architecture*), conferences (ER, WISE, SOFSEM, EDBT, ACM SAC, Visual, IEEE CIT, FLAIRS, PDCS), and has served on the program committees of several international conferences (ICDIM, IEEE SITIS, ACM SAC, IEEE ISSPIT, EuroPar, SBBD). He has organized many international conferences and workshops (ICDIM, CSTST, SITIS). He is currently the chair of the French Chapter ACM SIGAPP and the vice-chair of ACM SIGAPP. He has been member of the IEEE since 1999.

Gabriele Gianini received the PhD degree. He is an assistant professor at the Department of Information Technology at the University of Milan, where he is a lecturer of probability and statistics, and has been a visiting professor at the Free University of Bolzano since 2005. From 1990 and 2000, he worked at the Fermi National Accelerator Laboratory (Fermilab) in Chicago and at the CERN in Geneva. He is involved in several research projects funded by the Italian Ministry of Research and by the European Union. He is a member of the IEEE.