Using XML-based Multicasting to Improve Web Service Scalability

Joe Tekli [1*]
[1] Fac. of Computer Eng., Antonine University, Baabda – 40016, Lebanon
joe.tekli@upa.edu.lb

Ernesto Damiani [2]
[2] Dept. of Info. Technology, Università degli Studi di Milano, Crema, 65 – 26013, Italy
ernesto.damiani@unimi.it

Richard Chbeir [3]
[3] LE2I Laboratory CNRS, University of Bourgogne, Dijon 21078 Cedex, France
richard.chbeir@u-bourgogne.fr

ABSTRACT:

Web services' (WS) emphasis on open standards provides substantial benefits over previous application integration techniques. A major WS feature is SOAP, a simple, robust and extensible XML-based protocol for the exchange of messages. For this reason, SOAP WS on virtual hosts are now widely used to provide shared functionalities on clouds. Unfortunately, SOAP has two major performance-related drawbacks: i) *verbosity*, related to XML, that leads to increased network traffic, and ii) *high computational burden* of XML parsing and processing, that leads to high latency. In this paper, we address these two issues and present new results regarding our framework for Differential SOAP Multicasting (DSM). The main idea behind our framework is identifying the common pattern and differences between SOAP messages, modeled as trees, so as to multicast similar messages together. Our method is based on the well known concept of Tree Edit Distance, built upon a novel *filter-differencing* architecture to reduce message aggregation time, identifying only those messages which are relevant (i.e., similar enough) for similarity evaluation. In this paper we focus on recent improvements to the *filter-differencing* architecture, including a dedicated differencing output format designed to carry the minimum amount of *diff* information, in the multicast message, so as to minimize the multicast message size, and therefore reduce the network traffic. Simulation experiments highlight the relevance of our method in comparison with traditional and dedicated multicasting techniques.

KEYWORDS:

SOAP, XML, Message Multicasting, Differential Processing, SOAP Performance, Web Service Communications.

1. INTRODUCTION

Web Services – *WS* – have been proposed as a key technology for systematic and flexible application-to-application integration. Today, WS provide a comprehensive solution for representing, discovering and invoking services in a wide variety of virtualized architectures.

Here we focus on XML Web services, i.e. the ones that utilize message formats based on XML (Bray T., Paoli J. *et al.* 2008). This technology builds on two XML schemata: WSDL – *Web Service Description Language* (Chinnici R., Moreau J.J. *et al.* 2007) supporting the machine-readable description of a service's interface, and SOAP – *Simple Object Access Protocol* (W3 Consortium 2007) dictating the messages' format, with bindings to existing protocols (e.g., HTTP, FTP, SMTP, etc.) for SOAP message negotiation and transmission. As a result, WS can rely on existing XML parsers for automatic validation of messages. Also, the easy extensibility of XML schemata allows integration mechanisms to evolve, as markets require new functionalities, without causing incompatibilities and fragmentation of protocols.

[*] Work launched during the first author's post-doctoral mission at the University of Milan, partly funded by *Fondazione Cariplo.*

Today, many SaaS – *Software-as-a-Service* – applications are based on SOAP Web services. Client applications invoke WS executed in virtualized infrastructures; this way, underlying physical servers' capacity can be dynamically assigned to services, enabling innovative pay-as-you go revenue schemes. In a multi-tenant, virtualized setting, WS receive message flows coming from multiple applications at the same time; this makes non-functional requirements even more stringent. These requirements include WS performance, security and reliability, which are closely related to XML processing. In particular, WS have inherited a major XML drawback, *verbosity*, which strongly affects WS performance. SOAP message exchanges are quite elaborate; the client program has to build the skeleton of the XML message, put the right values in it (*serialization*), and then send it to the remote service. In turn, the remote service parses the message, digs out the data it needs (*de-serialization*), and then goes through the same procedure to generate an XML reply. No wonder, then, that SOAP message processing produces considerable network traffic and causes higher latency than competing technologies (Kohlhoff C. and Steele R. 2003; Suzumura T., Takase T. *et al.* 2005). High latency becomes more critical when handling large volumes of SOAP-based communications such as with cloud-based e-science (Gannon D., Krishnan S. *et al.* 2004) and e-business (Singh G., Bharathi S. *et al.* 2003) applications.

In the context of multi-tenant virtualized applications, however, the same WS are invoked by a high number of clients executing different applications. Intuition suggests that this scenario will increase the likelihood of WS receiving large numbers of similar SOAP messages.

Hence, similarity and differential encoding have been often proposed to enhance SOAP performance, aiming to: (i) reduce processing time – in *parsing* (Makino S., Tatsubori M. *et al.* 2005; Takeuchi Y., Okamoto T. *et al.* 2005; Teraguchi M., Makino S. *et al.* 2006), in *serialization* (Devaram K. and Andersen D. 2002; Abu-Ghazaleh N., Lewis M.J. *et al.* 2004), and in *de-serialization* (Abu-Ghazaleh N. and Lewis M.J. 2005; Suzumura T., Takase T. *et al.* 2005), and to (ii) reduce network traffic – via *compression* (Werner C., Buschmann C. *et al.* 2005) and *multicasting* (Phan K.A., Tari Z. *et al.* 2008; Azzini A., Marrara S. *et al.* 2009; Phan K.A., Bertok P. *et al.* 2009). This is based on the observation that SOAP exchanges often involve highly similar messages since 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 (e.g., stock quote services (Phan K.A., Tari Z. *et al.* 2008), online booking and meteorological broadcast services (Azzini A., Marrara S. *et al.* 2009), etc.). In this paper, we focus on *SOAP multicasting*, as a technique to save network bandwidth by delivering SOAP messages to a group of destinations simultaneously (Zhang B., Jamin S. *et al.* 2002).

To our knowledge, the only approach to SOAP multicasting was described in (Phan K.A., Tari Z. *et al.* 2008), where the authors introduce SMP (Similarity-based Multicasting Protocol), identifying, indexing and routing similar SOAP messages together (cf. Section 2). SMP's main contribution consists in grouping and transmitting together similar SOAP messages, in comparison with identical-only message aggregation of traditional network-layer (e.g., IP) multicasting (Zhang B., Jamin S. *et al.* 2002). Nonetheless, careful analysis of (Phan K.A., Tari Z. *et al.* 2008) led us to pinpoint certain aspects which limit both the effectiveness and efficiency of SMP multicasting. On one hand, while SMP considers the common and distinctive parts of SOAP messages in multicast message encoding, it does not always generate minimum sized aggregate messages (and thus does not guarantee optimal network traffic) since SMP disregards similarities between the SOAP messages' *distinctive* parts (which are repeated multiple times in the aggregate message regardless of their resemblances), as we will see in our motivating examples (Section 3.1). On the other hand, SMP consists of a two-phase message aggregation process: (i) computing SOAP similarity, and (ii) identifying message common/distinct parts, inducing additional processing overhead (i.e., higher response time), which could be alleviated if both tasks could be integrated together.

This paper builds on an improved SOAP multicasting method we designed in (Tekli J., Damiani E. *et al.* 2011a) to address the limitations of SMP (Phan K.A., Tari Z. *et al.* 2008). Our framework uses Differential SOAP Multicasting (DSM), to improve multicasting effectiveness (minimizing network traffic) and efficiency (minimizing processing overhead). DSM is founded on the well known concept of Tree Edit Distance (Buttler D. 2004; Bille P. 2005) for comparing and differencing SOAP messages. It is built upon a *filter-differencing* similarity evaluation architecture, inspired by *filter-refinement* approaches used in query processing (Korn F., Sidiropoulos N. *et al.* 1998; Kailing K., Kriegel H.P. *et al.* 2004). This allows identifying SOAP messages that are relevant (i.e., similar enough) for exact tree edit computations, avoiding computing similarity when it is not necessary. In addition, we define an XML-based differencing output format, SDL (Simple Diff Language), designed to carry the minimum information (in the aggregate multicast message) necessary to regenerate original SOAP messages (at multicast end-point), hence minimizing network traffic and latency during multicast message transmission. In short, our method allows:

- Encoding the differences between SOAP messages to be multicast, including only their *distinctive* parts, so as to minimize aggregate message size, and thus network traffic,
- Integrating both SOAP similarity computation and message aggregation in one single tree edit distance measure, enhanced via a dedicated filter-differencing technique, so as to reduce multicast processing overhead.

The groundwork results and overall architecture of DSM have been described in (Tekli J., Damiani E. *et al.* 2011a). This paper's contribution extends the latter publication with a number of new results. Specifically we describe an innovative *filter-differencing* module for DSM: the filter functions, the tree edit distance measure, as well as our differencing language (SDL), which was omitted from (Tekli J., Damiani E. *et al.* 2011a).

The remainder of the paper is organized as follows: Section 2 gives some background on SOAP performance enhancement, while Section 3 provides an overview of our approach. Section 4 contains a detailed explanation of our solution for SOAP multicasting, and Section 5 provides an experimental evaluation of the approach. Section 6 includes formal proofs highlighting some of the central properties stated in Section 4. Finally, Section 7 draws the conclusion.

2. BACKGROUND

Various studies have addressed SOAP performance enhancement (Tekli J., Damiani E. *et al.* 2011b). Most build on the observation that SOAP message exchange usually involves similar messages, and exploit SOAP similarity in order to gain in performance (e.g., execution time, memory, and network traffic). They can be categorized according to the kind of SOAP processing they perform:

Serialization: It consists in converting in-memory data types into SOAP (XML-based) format. In this context, the authors in (Abu-Ghazaleh N., Lewis M.J. *et al.* 2004) identify the main bottleneck as that of transforming in-memory data of numeric types into the corresponding ASCII-based XML representation. Consequently, they, introduce a method for differential SOAP serialization, storing SOAP messages in a dedicated buffer, to be used as templates for future outcalls. The message is fully serialized and saved during the first invocation of the SOAP call. Similar subsequent calls would thus avoid a significant amount of serialization processing by requiring that only the changes to the previously sent message be serialized. The authors exploit dedicated indexing tables to track changes between in-memory data and their serialized representations. Another approach comparable to that in (Abu-Ghazaleh N., Lewis M.J. *et al.* 2004) is introduced in (Devaram K. and Andersen D. 2002), where the authors address client-side SOAP message caching and allow entire request messages to be cached and sent as is. Yet, the approach in (Devaram K. and Andersen D.

2002) does not address partial structural matches (i.e., caching messages with different structures), which is performed in (Abu-Ghazaleh N., Lewis M.J. *et al.* 2004)

Parsing: SOAP parsing usually consists in analyzing the characters in the SOAP message, extracting tokens (e.g., tags and text) and validating the underlying XML structure. This can be achieved using existing XML parsers such as DOM (W3C Consortium 2005) and SAX (Megginson D. et al. 2004). Yet, a few studies have proposed dedicated parsers, considering the particularities of SOAP messages in order to amend performance. Early approaches such as XSOAP (Slominski A. 2004) limit the validation scope to those elements specific to SOAP so as to gain in validation time. More recent methods in (Makino S., Tatsubori M. *et al.* 2005; Takeuchi Y., Okamoto T. *et al.* 2005; Teraguchi M., Makino S. *et al.* 2006) focus on differential parsing, exploiting the similarities between SOAP messages. They make use of predefined templates modeled via dedicated automatons, memorizing the basic structures of the SOAP messages. Therefore, each incoming SOAP message is matched to the template, and only those parts of the message that correspond to variable parts in the template are parsed (the invariant parts being already parsed in advance). While the approach in (Takeuchi Y., Okamoto T. *et al.* 2005) makes use of a single predefined WSDL-based template, the authors in (Makino S., Tatsubori M. *et al.* 2005) propose a more dynamic method by managing multiple templates based on SOAP message structures. If the incoming message does not match any of the templates, then parsing is undertaken via an ordinary DOM processor (W3C Consortium 2005) and a new template corresponding to the unmatched message is created and appended to the automaton. An extension of the latter approach is provided in (Teraguchi M., Makino S. *et al.* 2006), introducing more expressive automatons able to consider repeatable structures in SOAP messages, so as to reduce templates memory size and processing time.

De-serialization: It can be viewed as the symmetric function of serialization, i.e., converting parsed XML messages to in-memory application objects. Here, the main idea to improving de-serialization performance is to avoid fully de-serializing each incoming message, by exploiting already constructed objects which were de-serialized in the past. In this context, the authors in (Suzumura T., Takase T. *et al.* 2005) propose an automaton-based, two-step solution. First, they generate an automaton based on incoming SOAP messages, and conduct de-serialization in the normal way, creating a link between the defined automaton and the application object. Then, they attempt to match each incoming message with the existing automaton, and if matched, return the linked application object to the SOAP engine after partially de-serializing only the regions that differ from the past messages. Another approach is provided in (Abu-Ghazaleh N. and Lewis M.J. 2005), where the authors propose to periodically checkpoint the state of the de-serializer, and compute checksums for portions of incoming SOAP messages. Consequently, the de-serializer compares the sequence of checksums against those associated to the most recently received message, to identify those portions of the message which are different, and which require regular de-serialization. The authors discuss that checksums can be error-prone, yet argue that the possibility of changes going undetected by checksumming is low in comparison with the gain in performance.

In (Kostoulas M. G., Matsa M. *et al.* 2006), the authors introduce XML Screamer, an optimized system providing tight integration across levels of software, combining: i) schema-based XML parsing (character encoding, token extraction, and validation) and ii) de-serialization, in one single processing layer (as opposed to handling parsing and de-serialization separately such as with most existing methods discussed above) in order to avoid unnecessary data processing, namely copying (to/from memory), and data-type transformations. Experimental results in (Kostoulas M. G., Matsa M. *et al.* 2006) show that XML Screamer delivers from 2.3 to 5.3 times the throughput of traditional SOAP toolkits.

On top of processing efficiency, a major drawback of SOAP is its demand for bandwidth, critical in various domains such as mobile computing (Phan K.A., Tari Z. *et al.* 2008) and sensor networks (Werner C., Buschmann C. *et al.* 2005). This problem has been investigated on two levels: (i) SOAP compression (Werner C., Buschmann C. *et al.* 2005) to reduce

message size prior to transmission, and (ii) SOAP multicasting (Phan K.A., Tari Z. *et al.* 2008; Phan K.A., Bertok P. *et al.* 2009) to optimize SOAP network traffic.

Compression: Various methods have been proposed for classic XML compression, e.g., (Liefke H. and Suciu D. 2000; Cheney J. 2001). Nonetheless, a comparative study conducted in (Werner C., Buschmann C. *et al.* 2005) showed that existing XML compression methods might not always be appropriate in the context of SOAP. That is due to the fact that SOAP messages are of relatively smaller sizes, and might yield compression coding tables which require more space than the original SOAP messages themselves (Werner C., Buschmann C. *et al.* 2005). Following this observation, the authors in (Werner C., Buschmann C. *et al.* 2005) propose a differential SOAP compression approach. They exploit the WSDL schema definition to generate a SOAP message skeleton describing the structures of corresponding SOAP messages. Consequently, only the differences between the SOAP message and the predefined skeleton are transmitted, along with corresponding SOAP message element/attribute values. The differences and element/attribute values are consequently patched to the same skeleton at the receiver side in order to reconstruct the original message. Note that the authors do not address the differencing part itself (e.g., differencing algorithm, output format), but rather present the overall architecture of their method, and propose to use any existing XML-based tree edit distance tool.

Multicasting: Another way to reduce SOAP network bandwidth is to perform multicasting, transmitting the same information destined to multiple clients once, instead of sending multiple replicas (Zhang B., Jamin S. *et al.* 2002). As outlined above, the Similarity-based SOAP Multicasting Protocol (SMP) proposed in (Phan K.A., Tari Z. *et al.* 2008) groups and transmits together similar SOAP messages, in comparison with identical-only message aggregation with traditional (IP) multicasting (Zhang B., Jamin S. *et al.* 2002). An aggregate SMP message consists 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 SMP message is then 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. Note that SMP is built on top of SOAP unicast and does not rely on low level (IP) multicast, in order to avoid handling complex network configurations. Each midway router parses the SMP header (containing client addresses) and examines its routing table to identify the next hops for each client address. The router then splits the SMP message accordingly and forwards the appropriate information to the next hop. The authors exploit a heuristic similarity measure (Ma Y. and Chbeir R. 2005) to quantify the resemblance between SOAP messages, in order to identify the most similar candidates for aggregation and multicasting. Message aggregation (identifying common/distinctive parts) is undertaken in a subsequent dedicated process. In a recent study (Phan K.A., Bertok P. *et al.* 2009), the authors propose an enhanced similarity-based routing protocol, transmitting messages following paths such as there are more shared links between similar messages. This allowed optimizing SMP network traffic distribution. SOAP multicasting has also been recently investigated in the context of SOAP security policy evaluation (Damiani E. and Marrara S. 2008; Turkmen F. and Crispo C. 2008; Azzini A., Marrara S. *et al.* 2009), applying security rules only on distinct parts of the multicast message so as to improve policy evaluation performance.

To sum up, automaton-based techniques to SOAP message comparison (mainly used with parsing and de-serialization) (Makino S., Tatsubori M. *et al.* 2005; Takeuchi Y., Okamoto T. *et al.* 2005; Teraguchi M., Makino S. *et al.* 2006) focus on messages which strictly correspond to predefined templates. They do not produce a similarity value to quantify the resemblance between SOAP messages, but rather a Boolean result identifying whether the message is valid or not w.r.t. (with respect to) the predefined template. Other approaches usually sacrifice some quality (i.e., comparison accuracy) to gain in performance, such as the error-prone checksum-based measure in (Abu-Ghazaleh N. and Lewis M.J. 2005) exploited for SOAP de-serialization), and the heuristic SMP similarity measure in (Phan K.A., Tari Z. *et al.* 2008) used for SOAP multicasting. Moreover, neither method allows seamless SOAP

message aggregation. For further details, a comprehensive survey on SOAP performance enhancement techniques is provided in (Tekli J., Damiani E. *et al.* 2011b).

3. OVERVIEW OF THE APPROACH

Our framework addresses the tasks of similarity evaluation and differential encoding of SOAP messages, to perform SOAP multicasting. As stated previously, we develop on the SMP multicasting technique (Phan K.A., Tari Z. *et al.* 2008), which aggregates SOAP messages by identifying their *common* and *distinctive* parts. SMP disregards certain similarities, mainly between the messages' *distinctive* parts, repeated multiple times in the aggregate message regardless of their resemblances

3.1 Motivating Example

To motivate the need for a new approach, let us consider the dummy SOAP messages M_i, $_{i=1...6}$ in Fig. 1. In this example, we abstract messages to character strings for the sake of simplicity. Fig. 1.a shows the expected aggregation result, using SMP. One can see that element *'e'*, which is contained in messages M_3, M_4, M_5 and M_6, is repeated four times in the SMP message *distinctive* section, so as to regenerate the original SOAP messages, such as: $M_i = Common + D_i$.

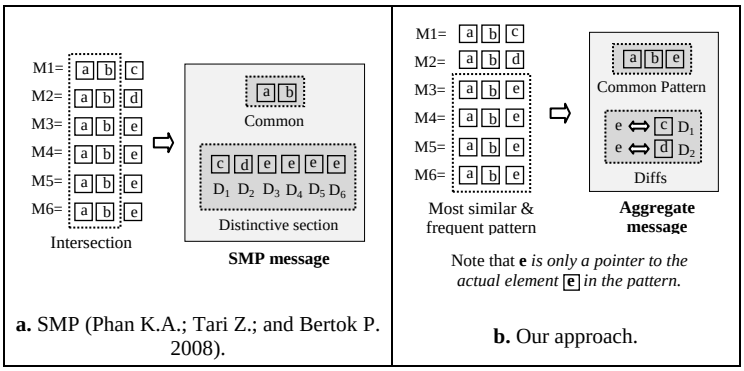


Fig. 1. Motivating example to SOAP message aggregation.

We argue that such repetitions of identical or similar elements can be eliminated in order to reduce the aggregate message size. To do so, we identify the most similar and frequent pattern among SOAP messages (instead of identifying the intersection as in SMP), and only encode the differences (*diffs*) between each message and the pattern. Hence, only the minimum amount of information needed to regenerate the original SOAP messages is encapsulated in the aggregate message, eliminating redundancies as shown in Fig. 1.b.

3.2 Underlying Technique

In order to attain our effectiveness (minimizing aggregate message size, and thus network traffic) and efficiency (reducing processing overhead) goals, we exploit the well known concept of tree edit distance (TED) (Zhang K. and Shasha D. 1989; Bille P. 2005) (also known as *tree differencing*), SOAP messages being modeled as Ordered Labeled Trees (W3C Consortium 2005). A great advantage of using tree edit distance is that along the similarity value, a *diff* is generated (i.e., *edit script*, or *delta*) providing a record of the exact differences, in terms of transformation operations, between the compared trees. This is central to achieve full integration of SOAP similarity evaluation and message aggregation (as opposed to the complex two-step process of SMP (Phan K.A., Tari Z. *et al.* 2008)). In addition, TED

methods have been widely used to compare XML-based data (Chawathe S. 1999; Nierman A. and Jagadish H. V. 2002; Dalamagas T., Cheng T. *et al.* 2006), and have been proven optimal w.r.t. less accurate (error-prone or heuristic) methods (Buttler D. 2004). This is of paramount importance to accurately identify the most common *pattern* minimizing the *diffs* among the SOAP messages being aggregated, and thus reducing overall aggregate message size.

3.3 Outline of our Proposal

We introduce a framework for Differential SOAP Multicasting (DSM), consisting of two main modules (Fig. 2): *Message Multicasting* (MM_{DSM}), and *Message Reconstruction* (MR_{DSM}). Briefly, our multicasting module starts by transforming SOAP messages into their DOM (W3C Consortium 2005) tree representations. SOAP trees are processed for similarity evaluation and aggregation simultaneously, via an integrated tree edit distance measure, to produce multicast DSM messages. Then, our message reconstruction module rebuilds the original SOAP messages. Note that each DSM multicast message consists of a message *pattern* and various *diffs*, describing the differences between the unicast SOAP messages and the multicast message *pattern*. The *pattern* comes down to the SOAP message sharing the maximum similarities to all others being processed in the same multicast, i.e., the message inducing the smallest *diffs*. Thus, message reconstruction consists in patching the *pattern* of the multicast message, with the *diff* corresponding to the SOAP message to be regenerated.

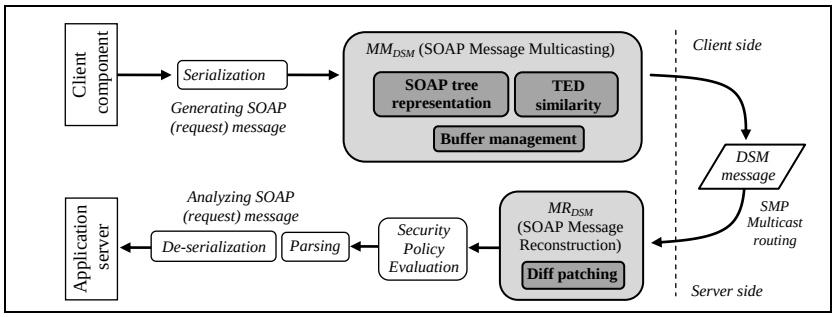


Fig. 2. Outline of our approach[1].

DSM provides an innovative multicasting technique w.r.t. the original SMP approach (Phan K.A., Tari Z. *et al.* 2008); however, our method exploits the message formatting, indexing and routing facilities provided by SMP.

4 SOAP Message Multicasting

The main idea consists in comparing SOAP messages in a pair-wise manner, generating and composing *diffs* accordingly. A DSM multicast message is generated for each group of SOAP messages such that their similarities are above a given threshold. Here, a user-defined similarity threshold $Thresh_{Sim}$ and time frame T_{Pool} are exploited. When the new outgoing SOAP message does not satisfy the threshold $Thresh_{Sim}$ w.r.t. all messages in the buffer, it is allocated a new buffer pool, for a period of T_{Pool} time, and constitutes the seed of a new DSM multicast message. When the outgoing message satisfies the similarity threshold, it is appended to the pool corresponding to the in-buffer message with which it shares maximum similarity. When the T_{Pool} expires for each buffer pool, the latter's buffer space is released and the corresponding multicast DSM message is sent over the wire. The activity diagram of our SOAP multicasting module is depicted in Fig. 3. It consists of three components: i) *SOAP*

[1] SOAP *response* message processing is similar to *request* processing, yet the *response* is generated at the server side, and transmitted toward the client.

Tree Representation, ii*) SOAP Tree Similarity Evaluation and Differencing*, and iii) *SOAP Buffer Management*.

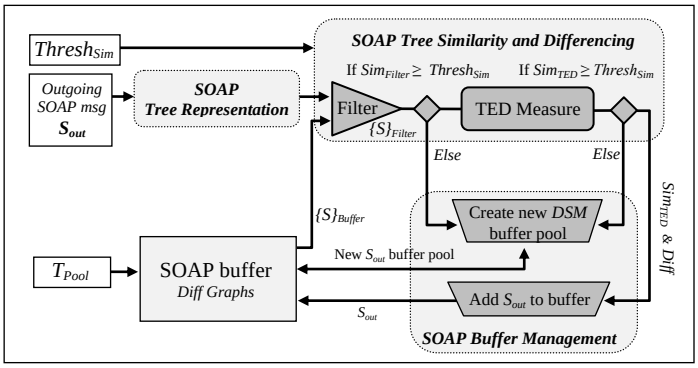


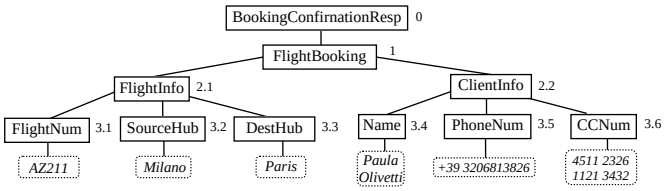
Fig. 3. Simplified activity diagram describing our SOAP message multicasting module, MM_{DSM}.

4.1 SOAP Tree Representation

Definition 1 – SOAP Message Tree: It is a rooted tree S which nodes $n_i \in S$ represent SOAP message elements, ordered and labeled following the corresponding message. Element values mark the nodes of their containing elements ●

In order to describe our tree representation, we use the same air travel booking service example we introduced in (Tekli J., Damiani E. *et al.* 2011a). This constitutes a typical scenario for SOAP multicasting since it involves a large number of similar transactions requesting booking information, confirmation and statistics. The SOAP response message in Fig. 4.a shows an answer to a booking confirmation request. Here, we only show the contents enclosed in the SOAP message body, and disregard meta-data in the header. The corresponding SOAP tree representation is depicted in Fig. 4.b.

```
<soap:Envelope xmlns:xsd= "…">
  <soap:Header> … </soap:Header>
  <soap:body>
    <BookingConfirmationResp>
      <FlightBooking>
        <FlightInfo>
          <FlightNum>AZ211</FlightNum>
          <SourceHub>Milano</SourceHub>
          <DestHub>Paris</DestHub>
        </FlightInfo>
        <ClientInfo>
          <Name>Paula Olivetti</Name>
          <PhoneNum>+39 3206813826</PhoneNum>
          <CCNum>4511 2326 1121 3432</CCNum>
        </ClientInfo>
      </FlightBooking>
    </BookingConfirmationResp>
  </soap:body>
</soap:Envelope>
```



a. Sample SOAP message S_1. **b.** SOAP tree representation.

Fig. 4. Sample SOAP message, and tree representation.

For tree node identifiers in the SOAP tree, we follow (Phan K.A., Tari Z. *et al.* 2008) in using a depth/order Dewey (like) numbering system, which allows to pinpoint the exact location of each node in the tree (central in subsequently encoding the *diffs* between SOAP trees, as we show in the following).

4.2 SOAP Tree *Filter-Differencing* Approach

We propose a two step *filter-differencing* similarity evaluation approach (cf. Fig. 3), inspired by *filter-refinement* architectures in query processing (Korn F., Sidiropoulos N. *et al.* 1998; Kriegel H.P. and Schönauer S. 2003; Kailing K., Kriegel H.P. *et al.* 2004). The main idea is

to first run a *filter* step, exploiting a fast approximation (Sim_{Filter}) of our main edit distance measure (Sim_{TED}) to compare the outgoing SOAP tree (S_{out}) to all those kept in the SOAP buffer. The filtering step identifies the set of SOAP trees in the buffer which are most similar (following Sim_{Filter}) to the outgoing tree S_{out}. Formally:

$$Filter = \{\ S \in Buffer \mid Sim_{Filter}(S_{out}, S) \geq Thresh_{Sim} \wedge \forall S' \in Buffer,\ Sim_{Filter}(S_{out}, S) \geq Sim_{Filter}(S_{out}, S')\ \} \quad \textbf{(1)}$$

The *differencing* phase consists in conducting similarity evaluation (Sim_{TED}) and *diff* generation to compare S_{out} with its most similar counterparts $S \in Filter$, identified in the filtering step.

4.2.1 Filter Similarity Measure

Three main conditions have to be satisfied for the filter step to be efficient (Kriegel H.P. and Schönauer S. 2003; Kailing K., Kriegel H.P. *et al.* 2004): (i) the filter measure has to be considerably easier to compute than the main similarity measure, (ii) a substantial part of the SOAP buffer messages has to be filtered out, and (iii) the completeness of the filter phase, w.r.t. the main similarity evaluation phase, has to be verified. While the first two criteria are intuitive, completeness in this context is less straightforward. It underlines that the filter step must not allow any false dropouts. In other words, all SOAP trees in the buffer ($S \in Buffer$), which are deemed similar to S_{out} w.r.t. the main similarity measure Sim_{TED}, should be included in the filter candidate set ($S \in Filter$).

Definition 2 – Upper Bound Function: Let Ω be a set of objects, a similarity function *Sim'* is an upper bound of function *Sim*, if $\forall p, q \in \Omega$, Sim'(p, q) $\geq$ *Sim* (p, q) (Davey B. A. and Priestley H. A. 2002) ●

Definition 3 – Filter Completeness: Given a similarity measure Sim_{TED}, and a filter characterized by similarity measure Sim_{Filter}, the filter is said to be complete w.r.t. Sim_{TED} if Sim_{Filter} is an upper bound of Sim_{TED} (Kailing K., Kriegel H.P. *et al.* 2004) ●

With our upper bound similarity measure, it is possible to safely filter out all buffer SOAP trees that have a filter similarity Sim_{Filter} less than the minimum acceptable similarity degree, i.e., $Thresh_{Sim}$ (cf. Formula (1)). In other words, our filter eliminates all candidate SOAP trees which are outside the maximum relevant similarity range, for the message aggregation and multicasting operation at hand.

Several TED-related filter similarity functions have been proposed in the context of structure query processing (Kriegel H.P. and Schönauer S. 2003; Kailing K., Kriegel H.P. *et al.* 2004). These range over very coarse functions comparing the number of edges in both structures being compared (Kriegel H.P. and Schönauer S. 2003), to more complex measures exploiting special histograms to describe the structural features of the data (distribution of the number of leaf nodes, distinct node labels, etc.) (Kailing K., Kriegel H.P. *et al.* 2004). Since existing filter methods seem either too coarse (Kriegel H.P. and Schönauer S. 2003) or somewhat complex (Kailing K., Kriegel H.P. *et al.* 2004), we propose three simple filter functions to specifically capture the main characteristics of SOAP message trees: *node edges* (*parent-child relations*) and *node order* to describe SOAP structure, and *node values* to describe SOAP message contents. Our filters are based on the vector space model widely used in information retrieval (McGill M. 1983), which performance has been accredited in a variety of applications (Salton G. 1989).

Definition 4 – Node-Edge Vector Space: Given two SOAP trees S_i and S_j, we define corresponding parent-child vectors $\overrightarrow{V_i}$ and $\overrightarrow{V_j}$ in a space which dimensions represent, each, a single edge $e_r \in (S_i \times S_i) \cup (S_j \times S_j)$, such as $1 < r < E$ where E is the number of distinct

parent-child relations in S_i and S_j. The value of a coordinate $w_{\vec{V_i}}(e_r)$ in $\vec{V_i}$ stands for the number of occurrences of edge e_r in tree S_i ●

We exploit the Manhattan distance (Krause E.F. 1987) to compute the *node edge* filter function $Sim_{n\text{-}edge}$, since it is consistent with Definitions 2 and 3, in providing a lower bound for our main TED similarity measure (the mathematical proof is provided in Section 6).

$$\mathrm{Sim}_{\text{n-edge}}\left(S_i, S_j\right) = 1 - \frac{\frac{1}{2}\sum_{r=1}^{E} \left| w_{\vec{V_i}}(e_r) - w_{\vec{V_j}}(e_r) \right|}{|S_i| + |S_j|} \quad \in [0,1] \tag{2}$$

We use similar formulas, based on the Manhattan distance, to compute both the *node order* and *node value* filter functions: $Sim_{n\text{-}order}$ and $Sim_{n\text{-}value}$, each w.r.t. its corresponding vector space defined hereunder.

Definition 5 – Node-Order Vector Space: Given two SOAP trees S_i and S_j, we define the node order vectors $\vec{V_i}$ and $\vec{V_j}$ in a space whose dimensions represent, each, the Dewey index (Phan K.A., Tari Z. *et al.* 2008) (cf. Section 4.1) associated to a single node $n_r \in S_i \cup S_j$, such as $1 < r < I$ where I is the number of distinct node index values in S_i and S_j. Vector coordinates are binary, indicating whether a node of the designated Dewey index exists or not for a given dimension n_r ●

Definition 6 – Node-Value Vector Space: Given two SOAP trees S_i and S_j, we define node value vectors $\vec{V_i}$ and $\vec{V_j}$ in a space whose dimensions represent, each, a distinct node value associated to a node $n_r \in S_i \cup S_j$, such as $1 < r < Vl$ where Vl is the number of distinct node values in S_i and S_j. Vector coordinates designate the occurrences of each node value ●

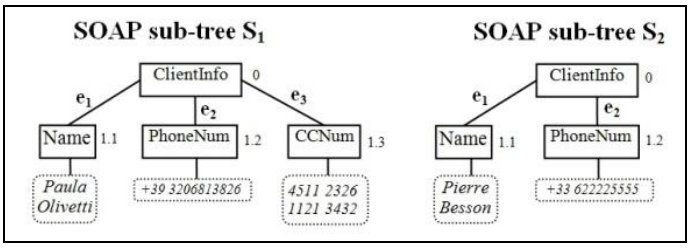


Fig. 5. Sample SOAP sub-trees.

Consider for instance the SOAP trees in Fig. 5. The corresponding filter vector representations are depicted in Fig. 6.

	e_1	e_2	e_3
V_1	1	1	1
V_2	1	1	0

a. Node edge vectors

	0	1.1	1.2	1.3
V_1	1	1	1	1
V_2	1	1	1	0

b. Node order vectors

	P. Olivetti	+39 320…	4511…	P. Besson	+33 622…
V_1	1	1	1	0	0
V_2	0	0	0	1	1

c. Node value vectors

Fig. 6.SOAP tree filter vector representations.

A classic solution to the problem of combining different filters is to apply them independently, and then intersect the resulting candidate sets (Kailing K., Kriegel H.P. *et al.* 2004). With such an approach, separate index structures for different filters have to be maintained and for each filtering task, a time-consuming intersection step is necessary. In addition, all filters functions would be equally weighted regardless of their relative importance. We follow a different approach, combining the filter functions in one integrated Sim_{Filter} measure, weighting each function based on its discriminative power over the SOAP tree candidate set. We do so by computing the variance for each filter function over all SOAP

trees within the candidate set, and normalizing each function accordingly. This brings filter similarities according to different features (parent-child relations, node order and node values) in a similar range, and assigns a larger weight to features that are a good discriminator for the specific set of candidate SOAP trees at hand. Formally:

$$\mathrm{Sim}_{\mathrm{Filter}}\left(\mathrm{S_i}, \mathrm{S_j}\right) = \frac{1}{\sum_{h \in F}\left(1-\nabla_h^{\,2}\right)} \sum_{f \in F}\left(1-\nabla_f^{\,2}\right) Sim_f\left(\mathrm{S_i}, \mathrm{S_j}\right) \quad \in [0,1] \qquad \textbf{(3)}$$

where $F=\{n\text{-}edge, n\text{-}order, n\text{-}value\}$ is the set of component filters, $Sim_f(S_i, S_j)$ is the similarity function between SOAP trees S_i and S_j for a given filter component $f \in F$, and $\nabla_f^{\,2}$ is the variance over all SOAP trees according to the f-th filter function within the SOAP tree candidate set. The combined filter measure Sim_{Filter} is consistent with Definitions 2 and 3, since each of its component filter functions is an upper bound of our main TED measure (cf. Section 6.2 for a complete mathematical proof).

4.2.2 Tree Edit Distance Similarity Measure

In our SOAP multicasting approach, we exploit a variation of the classic tree edit distance developed in (Chawathe S. 1999). Hereunder the basic definition of tree edit distance (Zhang K. and Shasha D. 1989; Chawathe S. 1999):

Definition 7 – **Tree Edit Distance:** The edit distance between two trees A and B is defined as the minimum cost of all edit scripts (*diffs*) that transform A to B, $TED(A, B)=Min\{Cost_{Diff}(A, B)\}$ ●

Definition 8 – Edit Script - Diff: It is a sequence of edit operations $Diff = \prec op_1, op_2, \ldots, op_k \succ$, transforming one tree into another. The cost of an edit script is defined as the sum of the costs of its operations: $Cost_{Diff} = \sum_{i=1}^{|Diff|} Cost_{Op_i}$ ●

The algorithm in (Chawathe S. 1999) exploits three basic edit operations: *node insertion*, *node deletion* and *node update*, disregarding more complex operations such as *move node*, *insert sub-tree*, etc., so as to increase efficiency. This algorithm has been considered as a reference point for various XML related comparison studies (Nierman A. and Jagadish H. V. 2002; Dalamagas T., Cheng T. *et al.* 2006). It is among the fastest and least complex TED algorithms available (Dalamagas T., Cheng T. *et al.* 2006; Tekli J., Chbeir R. *et al.* 2009), also, it guarantees correct results (minimal *diffs*) in comparison with existing works, e.g., (Chawathe S., Rajaraman A. *et al.* 1996; Cobéna G., Abiteboul S. *et al.* 2002) which utilize various heuristics to gain in performance. Nonetheless, the original approach described in (Chawathe S. 1999) only considers tree structures (node labels, and parent/child relationships), but not the values (since the algorithm was designed for generic hierarchical data). Hence, we redefine the set of edit operations to consider SOAP node values in the differencing process. First, we formalize the notion of SOAP tree node, necessary to define edit operation syntaxes:

Definition 9 – SOAP Tree Node: Given a SOAP ordered labeled tree, a SOAP tree node x can be represented as a triplet $x = (id, l, v)$ where $x.id$ underlines the node's identifier, $x.l$ its label, and $x.v$ its value. For an internal node, $x.v = Ø$ ●

Definition 10 – Update node: Given a node x in SOAP tree S, with label $x.l$ and value $x.v$, and given a new label l' and value v', $Upd(x, l', v')$ is an update operation applied to x resulting in tree S' identical to S except that in S', x bears l' as label and v' as value. When $l'= x.l$ or $v'= x.v$, it simplifies to $Upd(x,_{\equiv}, v')$ or $Upd(x, l',_{\equiv})$ ●

Definition 11 – Insert node: Let S be a SOAP tree with a node p having first level sub-trees $S_1, \ldots, S_m$ (i.e., sub-trees rooted at the children of node p). Given a SOAP tree node x not belonging to S, $Ins(p, i, x)$ is a node insertion applied to S, inserting x as the i^{th} child of p, thus yielding S' with first level sub-trees $S_1, \ldots, S_{i-1}, x, S_{i+1}, \ldots, S_{m+1}$ ●

Definition 12 – Delete node: Let S be a SOAP tree with node p, having a leaf node x as the i^{th} child of p, $Del(p, i, x)$[1] is a node deletion operation applied to S that yields S' where node p will have level sub-trees $S_1, \ldots, S_{i-1}, S_{i+1}, \ldots, S_m$ ●

A major question in edit distance approaches is how to choose operation cost values. In our current approach, we define operations' costs in an intuitive and natural way, by assigning identical unit costs to *insertion* and *deletion* operations ($Cost_{Ins} = Cost_{Del} = 1$), as well as to *update* operation ($Cost_{Upd}=1$) only when the newly assigned label and/or value are different from the node's current label and value (otherwise, $Cost_{Upd} = 0$, underlining that no changes are to be made to the concerned node). Note that the investigation of alternative tree operations cost models (considering for instance the semantic relatedness between SOAP node labels/values given a reference semantic network such as WordNet (Miller G. 1990) or Wikipedia) is not considered in the scope of this paper and will be addressed in a dedicated upcoming study.

Consequently, given two SOAP trees S_i and S_j, we compute their similarity based on the tree edit distance function:

$$Sim_{TED}(S_i, S_j) = 1 - \frac{TED(S_i, S_j)}{|S_i| + |S_j|} \in [0,1] \quad \textbf{(4)}$$

Consider the sample SOAP trees in Fig. **5**. $TED(S_1, S_2) = 3$, $Diff(S_1, S_2)$ consisting of three operations: i) updating the value of node *name*, ii) updating the value of *PhoneNum*, and iii) deleting node *CCNum*. Formally:

$Diff(S_1, S_2) = \prec$ Upd ({1.1, *Name*, *'Paula Olivetti'*}, ≡, *'Pierre Besson'*),
Upd ({1.2, *PhoneNum*, *'+39 32…'*}, ≡, *'+33 62…'*),
Del ({0, *ClientInfo*, Ø,}, {1.3, *CCNum*, *'4511…'*}, 3} $\succ$

a. Forward direction *diff*, transforming S_1 into S_2.

⇳

$\prec$ Upd ({1.1, *Name*, *'Pierre Besson'*}, ≡, *'Paula Olivetti'*),
Upd ({1.2, *PhoneNum*, *'+33 22…'*}, ≡, *'+39 32…'*),
Ins ({0, *ClientInfo*, ,}, {1.3, *CCNum*, *'4511…'*}, 3} $\succ$

b. Backward direction *diff*, transforming S_2 into S_1.

Fig. 7. Sample $Diff(S_1, S_2)$ example computed based on the SOAP trees in Fig. 5.

Note that the *diffs* generated following our differencing method logically encompass both the forward and the backward transformation scripts: $Diff(S_i, S_j) = \{\triangle_{ij}, \triangle_{ji}\}$, such as $\triangle_{ij}$ denotes the sequence of edit operations transforming tree S_i into S_j, whereas $\triangle_{ji}$ denotes the sequence of edit operations transforming S_j into S_i. such as $\triangle_{ji}$ can be seamlessly identified based on $\triangle_{ij}$ and vice-versa (cf. example of forward and backward *Diff* representations in Fig. 7). This is central since, at this stage, we do not know which transformation direction will be used in constructing the aggregate DSM message (to be identified in the subsequent *buffer management* phase, described in the following section). Hence, we specifically defined our edit operations' syntaxes (cf. Definitions 10-12) in a way to produce complete (bi-directional)

[1] Some parameters in the delete operation, such as the parent node p, are redundantly preserved on purpose, in order to guaranty bi-directional *diffs* (discussed in more detail in the following).

diffs at this stage so as to prevent redundant edit distance processing afterward. Yet, all redundant *diff* code information will be discarded when transforming the *diffs* into their machine-readable output format, prior to final *diff* encoding in the multicast message. The *diff* output format, which is crucial in i) minimizing multicast message size, and ii) enabling SOAP message reconstruction, is developed in Section 4.4.

4.3 SOAP Buffer Management

4.3.1 SOAP Diff Graph Representation

In order to effectively multicast buffered SOAP trees, we represent the latter as a graph-like structure, named *SOAP Diff Graph (SDG)*, connecting SOAP messages (graph nodes) via corresponding *diffs* (graph edges, Fig. 8). The buffer consists of multiple *SDG* graphs corresponding to the different buffer pools, each underlining a potential DSM multicast message.

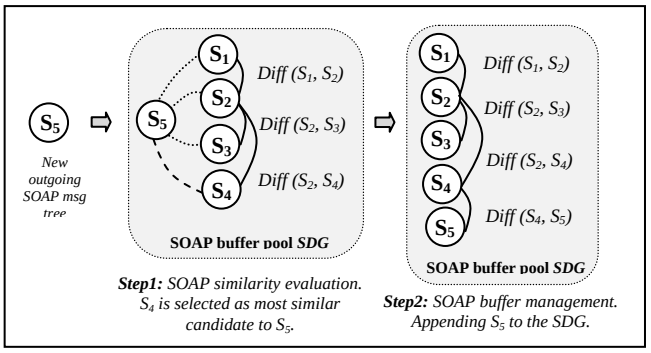


Fig. 8. An Example of SOAP buffer management.

As described previously, TED computations for similarity evaluation and *diff* generation are carried out for each new outgoing SOAP message S_{out}, w.r.t. its most similar counterparts in the buffer (i.e., the SOAP tree candidates identified via the *Filter* component). Consequently, the filter candidate S_i maximizing the main similarity measure $Sim_{TED}(S_{out}, S_i)$ is selected. If $Sim_{TED}(S_{out}, S_i) \geq Thresh_{Sim}$, then S_{out} would be appended to the corresponding *SDG* graph, connected to S_i via their common *diff*. Otherwise, if $Sim(S_{out}, S_i)$ drops below $Thresh_{Sim}$, it is allocated a new buffer pool, and constitutes the first node in a new *SDG* graph. When the buffer pool time frame T_{Pool} expires, the corresponding *SDG* is encapsulated in a *DSM* multicast message and is transmitted over the network. A simple example is depicted in Fig. 8 to show how an outgoing SOAP message tree S_5, is appended to a SOAP buffer pool *SDG*.

4.3.2 DSM Multicast Message

Encapsulating the *SDG* graph into a *DSM* multicast message requires identifying the multicast message *pattern* $S_{pattern}$, which is the most similar and frequent pattern in all messages, minimizing the different parts, i.e., the *diffs*. In other words, it consists in minimizing the multicast message size. Formally:

$$S_{pattern} = S_i \in SDG \text{ verifying } \underset{\forall S_i \in SDG}{Min} \left\{ |S_i| + \sum_j |Diff(S_i, S_j)| \right\} \quad \textbf{(5)}$$

where $|S_i|$ and $|Diff(S_i, S_j)|$ denote the cardinalities (the number of nodes) of the SOAP tree S_i and the *diff* linking S_i and S_j.

This can be performed in linear time w.r.t. the number of SOAP trees in the *SDG* graph, and is achieved by pinpointing the *SDG* node (i.e., SOAP tree) with the maximum number of edges (i.e., *diffs*). The latter, which we identify as *SDG centroid*, underlines the SOAP tree

requiring the least amount of transformation operations, i.e., the smallest *diffs*, in order to generate all its remaining counterparts in the *SDG*. In other words, the *SDG centroid* minimizes the differential parts in the *DSM* message, and thus reduces overall multicast message size. It identifies the SOAP tree with the maximum amount of commonalities w.r.t. its counterparts.

Consider the *SDG* graph in Fig. 8. Here, SOAP tree S_2 is selected as *SDG centroid*, since it is connected to its counterparts with the maximum number of minimal *diffs* (*SDG* edges). Thus, the corresponding *DSM* message consists of tree S_2 as the multicast message *pattern*, and $Diff(S_1, S_2)$, $Diff(S_2, S_3)$, $Diff(S_2, S_4)$, $Diff(S_4, S_5)$ as the differential parts corresponding to each SOAP tree. Recall that our DSM messages follow the same format as SMP messages (Phan K.A., Tari Z. *et al.* 2008) w.r.t. message header, body, indexing and routing addresses.

4.3.3 DSM Multicast Message

Our routing process is comparable to that of SMP (Phan K.A., Tari Z. *et al.* 2008) except that instead of aggregating and splitting common/different parts of the multicast message, the router patches the DSM *pattern*, i.e., *SDG centroid*, with the corresponding *diff* so as to regenerate the original SOAP tree. Consider the example in Fig. 8, such as each SOAP tree S_i is intended for a different client C_i. The DSM replicas to be sent to each client consist of:

- The pattern S_2 and $Diff(S_1, S_2)$, to regenerate SOAP tree S_1, destined to client C_1,
- The pattern S_2, destined to client C_2,
- S_2 and $Diff(S_2, S_3)$, to regenerate S_3, destined to client C_3,
- S_2 and $Diff(S_2, S_4)$, to regenerate S_4, destined to client C_4,
- S_2 and $Diff(S_2, S_4) \oplus Diff(S_4, S_5)$, to regenerate S_5, for C_5.

The $\oplus$ symbol designates the *diff* composition operator (Marian A., Abiteboul S. *et al.* 2001), which underlines the transformation of SOAP tree S_2, via two consecutive *diffs*, so as to obtain S_5. In plain terms, it consists in transforming S_2 into S_4 (using $Diff(S_2, S_4)$), and then S_4 into S_5 (via $Diff(S_4, S_5)$).

4.4 SOAP Message Reconstruction

When the DSM multicast message reaches the destined end-point client/server (or end-point router), the original SOAP message is to be reconstructed, based on the DSM common *pattern* and corresponding SOAP message *diff*, in order to be processed by the destination service component (Fig. 2). While *tree differencing* (i.e., tree edit distance) was used as an effective means to perform SOAP aggregation, we exploit its inverse process, *tree patching*, for message reconstruction.

Definition 13 – Tree Patching: It is defined as the problem and action of applying a *diff* to a tree structure (pattern) *T* in order to create a new version of the tree *T'*, incorporating all the changes encoded in the *diff* (Mouat A. 2002; Komvoteas K. 2003)●

In short, tree patching allows regenerating the original SOAP message tree at the receiver end, by applying the *diff* corresponding to the SOAP message tree, on the common DSM message *pattern*. However, a machine-readable *diff* output format is required in order to automatically perform the patching operation. Consequently, patching comes down to executing the edit operations encoded in the output *diff*, applied on the DSM *pattern*.

4.4.1 Diff Output Representation Format

Having computed the logical *diff* describing the changes between two SOAP message trees, the latter is to be outputted in a useful and machine-readable format to be encoded in the

multicast DSM message, so as to allow automatic tree patching and message reconstruction at the receiver side. In short, we aim to obtain an output *diff* representation which is:

i) Described in a simple XML encoding, to provide more flexibility and improve human readability in handling the *diffs*, and which would clearly be more suitable in the context of SOAP multicasting, since SOAP itself is XML-based,

ii) Compact in its description, including only the information necessary to regenerate the SOAP messages, in order to minimize multicast message size, and thus to optimize network traffic.

Different XML-based tree *Diff* representations exist (Laux A. and Martin L. 2000; Monsell EDM ltd. 2000; Cobena G., Abiteboul S. *et al.* 2001; Mouat A. 2002; Komvoteas K. 2003), each developed in a specific scenario, and dedicated to a specific application. While most representations provide flexibility in handling the *diffs*, they also inherit XML's verbosity. Some formats such as DeltaXML (Monsell EDM ltd. 2000) and XyDiff (Cobena G., Abiteboul S. *et al.* 2001) purposefully include additional redundant information so that the *diffs* follow the same topological structure of their source documents. This is useful for specific applications such as temporal querying and monitoring changes (Marian A., Abiteboul S. *et al.* 2001). Other formats, such as DUL (Mouat A. 2002) and EDUL (Komvoteas K. 2003), include additional context information concerning the siblings and parents of the nodes affected by each edit operation (e.g., number of siblings and their labels, parent node siblings, etc.), in order to generate *diff* descriptions which would be independent of the document trees based on which they were generated, to be patched with any arbitrary document tree. In short, none of the existing formats seems adapted to our simple and specific needs, mainly human readability and compactness.

4.4.2 Simple Diff Language (SDL)

Thus, we introduce a dedicated *diff* representation format: *SDL* (*Simple Diff Language*), which allows encoding edit operations as follows (recall operation Definitions 10-12 in Section 4.2.2):

The update operation, *Upd*(*x, l', v'*):
- General case: <Upd node_id='*x.id*' label='*l'*'> *v'* </Upd>
- When *l'* = *x.l* : <Upd node_id='*x.id*'> *v'* </Upd>
- When *v'* = Ø or *v'* = *x.v* : <Upd node_id='*x.id*' label='*l'*' />

The insertion operation, *Ins*(*p, i, x*):
- <Ins parent_id = '*p.id*' pos= '*i*' label= '*x.l*'> *x.v* </Ins>
- When *v* = Ø: <Ins parent_id = '*p.id*' pos= '*i*' label= '*x.l*' />

The deletion operation, *Del*(*p, i, x*):
- General case: <Del node_id = '*x.id*'/>

One can clearly see that the old node label (*x.l*) and value (*x.v*) are not preserved in the SDL representation of the update operation. Similarly, the parent node (*p*), node sibling order (*i*), node label (*x.l*) and value (*x.v*) do not appear in the SDL representation of the deletion operation, since the SDL *diff* representations are only required to carry the minimum necessary information needed to apply the corresponding edit operations. In fact, at this stage, we already know the *diff* transformation direction to be used in the aggregate DSM message (following the corresponding *SOAP Diff Graph*, cf. Section 4.3), and thus can seamlessly (with no additional edit distance processing) eliminate all redundant information in operation syntaxes. Hence, w.r.t. the update operation, the old node label (*x.l*) and value (*x.v*) become dispensable since only the new ones are actually required. Likewise for the deletion operation

where the only information required to delete a node is its structural position, known via its identifier (*x.id*).

For instance, the output SDL representation corresponding to *Diff*(S_1, S_2) developed in Section 4.2.2 (and reported hereunder), is shown in Fig. 9.

Logical *diff*	*Diff*(S_1, S_2) = ≺ Upd ({1.1, *Name*, *'Paula Olivetti'*}, ≡, *'Pierre Besson'*), Upd ({1.2, *PhoneNum*, *'+39 32…'*}, ≡, *'+33 62…'*), Del ({0, *ClientInfo*, Ø,}, {1.3, *CCNum*, *'4511…'*}, 3} ≻
SDL *diff* representation	*<Diff Source=* 'S$_1$' *Dest=* 'S$_2$'*>* *<Upd node_id=* '1.1'*> Pierre Besson </Upd>* *<Upd node_id=* '1.2'*> +33 62…</Upd>* *<Del nod_id=* '1.3'*>* *</Diff>*

Fig. 9. An Example of SDL encoding.

Output *diffs* are encoded following SDL in order to include only the minimum amount of necessary information in the DSM multicast message, to be transmitted over the wire, and then patched with the DSM message *pattern* at the end-point client/server to regenerate corresponding original SOAP messages.

4.5 Complexity Analysis

The time complexity of our approach simplifies to $O(N \times |S|^2)$ where N is the maximum number of in-buffer SOAP messages, and $|S|$ the cardinality of the largest SOAP message tree. This includes both the complexities of *SOAP Message Multicasting* at the sender side (MM_{DSM}, cf. Fig. 2), and *SOAP Message Reconstruction* at the receiver side (MR_{DSM}), and can be evaluated as follows.

The complexity of *SOAP Message Multicasting* comes down to $O(N \times |S|^2)$:

- *SOAP Tree Similarity Evaluation and Differencing* is of $O(|S|^2)$:
  - The *Filter* component is of $O(|S|)$ time, each of the filter similarity functions, being evaluated in average linear time w.r.t. SOAP message size,
  - The complexity of the tree edit distance algorithm (i.e., the main TED similarity measure) adapted from (Chawathe S. 1999), is of $O(|S|^2)$.
- *SOAP Buffer Management* is of worst $O(N \times |S|^2)$, and comes down to the complexity of running the *SOAP Tree Similarity Evaluation and Differencing* module (cf. Fig. 3) to compare the new outgoing message, to each in-buffer message tree (recall that N is the maximum number of SOAP message trees in the buffer),

The complexity of the *SOAP Message Reconstruction* operation comes down to $O(N \times (2 \times |S|))$ time, since:

- The complexity of the *tree patching* operation is of worst $O(2 \times |S|)$ time, where $2 \times |S|$ underlines the maximum possible *diff* size (corresponding to the deletion and insertion of every node in SOAP tree S),
- *Tree patching* is performed for each of the N *diffs* corresponding to the SOAP messages encapsulated in the DSM multicast message. Note that the number of *diffs* corresponding to the DSM message is at most N, i.e., when the buffer consists of one single pool corresponding to the DSM message at hand, grouping all N in-buffer SOAP messages.

Likewise, space complexity also simplifies to $O(N \times |S|^2)$ in the worst case, considering RAM space to store: i) the SOAP message trees being evaluated for multicasting (i.e., in-buffer messages), which is of $O(N \times |S|)$, ii) the filter vectors corresponding to each SOAP tree, which is of worst $O(N \times |S|)$, and iii) the distance matrixes and *diffs* computed during similarity evaluation, requiring $O(|S|^2)$ space for each of the N in-buffer message trees, hence: $O(N \times |S|^2)$. Space complexity analysis is straight forward, and in various ways similar to time analysis. Thus, details have been omitted for clearness of presentation.

5 EVALUATION

We conducted several new simulation experiments to test the performance of our approach, and compare it to *SMP*, *traditional multicast* (aggregating identical messages only), and *unicast*. We evaluated two main criteria: i) network traffic (multicasting effectiveness), and ii) processing time (multicast efficiency).

5.1 Network Traffic

We adopt a single sender/receiver scenario such as the messages are multicast at the sender end-point, and reconstructed at the receiver end-point, disregarding intermediate routers. Hence, network traffic amounts to the sum of the sizes of all SOAP messages over the client/server link. As for the test data, two sets of 500 SOAP messages (each) were generated (of average 4KB per message), based on Google's web service SOAP request and response WSDLs[1], using an adaptation of IBM's XML document generator[2].

We varied three main parameters and evaluated network variation accordingly: the amount of Non-Identical Messages (*NIM %*) sent to the client/server, the amount of pair-wise modifications (*Modifs %*) between non-identical messages (which we tuned via the IBM generator), and the number of messages considered for multicasting (*NbMsg*).

5.1.1 Network Traffic when varying NIM% and Modifs %

First, we fixed the total number of SOAP messages to be multicast, *NbMsg = 500*, and evaluated network traffic w.r.t. *NIM %* and *Modifs %*.

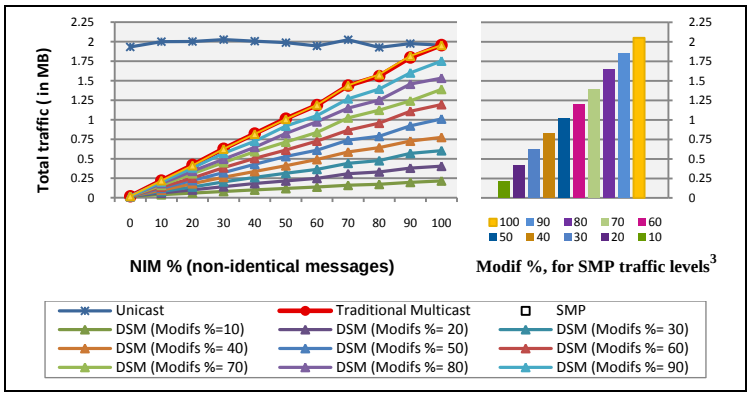


Fig. 10. Variation of network traffic w.r.t. the amount of modifications between messages.

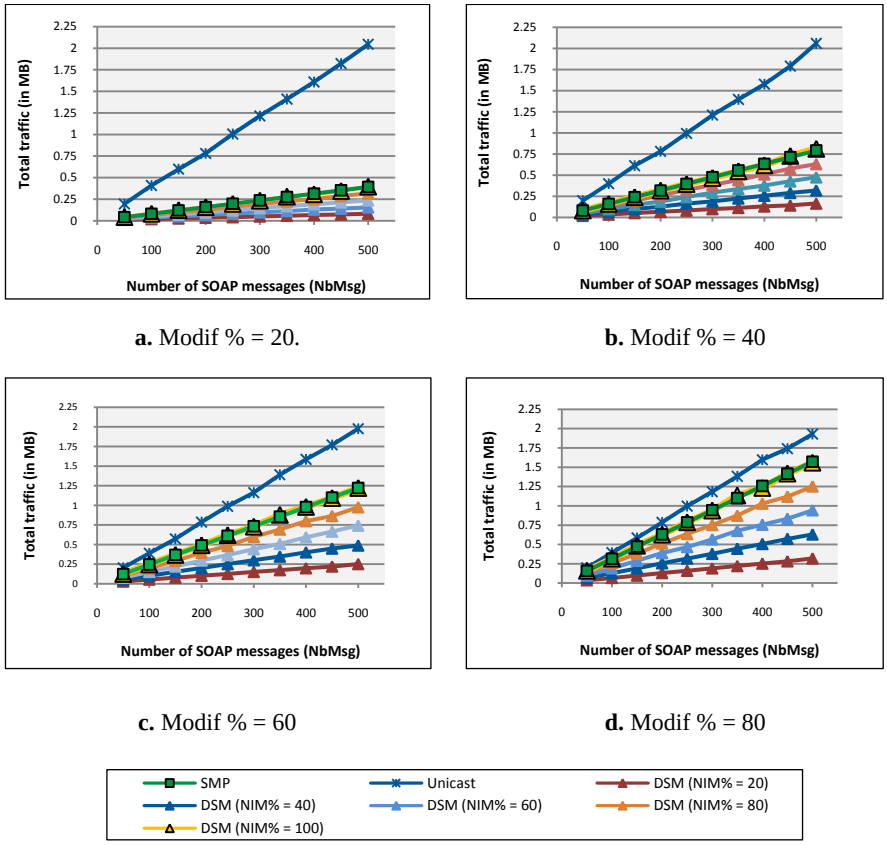
Results in Fig. 10 show that our approach (DSM) reduces traffic proportionally to the amount of differences (both *NIM %* and *Modifs %*) among messages. SMP reduces traffic w.r.t. the amount of pair-wise message modifications (*Modif %*), regardless of the amount of non-identical messages (*NIM %*), and thus produces the same 'worst case' results that are obtained via DSM (DSM's upper traffic limit) when none of the messages to be multicast are

[1] http://www.w3.org/2004/06/03-google-soap-wsdl.html
[2] http://www.alphaworks.ibm.com.
[3] SMP traffic levels are invariant w.r.t. NIM %, and are thus represented separately.

identical (*NIM %=100*). That happens because SMP only considers the intersection between messages when generating the aggregate multicast, regardless of the largest or most frequent message pattern. Traditional multicast reduces traffic w.r.t. the amount of non-identical messages (*NIM %*), but does not consider partially similar messages (*Modif %*) since it only aggregates identical messages. It produces the 'worst' results obtained using DSM, when messages are completely different (*Modif%=100*). The largest traffic is constantly produced via unicast, since the latter transmits messages regardless of their similarities (despite *NIM%* and *Modifs%*).

5.1.2 Varying the Number of SOAP Messages to be Multicast

Fig. 11 depicts network traffic when varying the number of SOAP messages considered for multicasting (*NbMsg*), such as the number of non-identical messages (*NIM %*) varies linearly w.r.t. the amount of pair-wise message modifications (*Modifs%*).



a. Modif % = 20.

b. Modif % = 40

c. Modif % = 60

d. Modif % = 80

Fig. 11. Comparing network traffic variation between DSM and SMP, when varying the number of messages to be multicast, *NbMsg*.

Results confirm those of the previous experiment (Fig. 10): i) unicast yields the highest network traffic levels, which remain unwavering w.r.t. the number of identical and/or similar messages (*NIM %* and/or *Modif %*), ii) traditional multicast only considers identical messages and thus varies w.r.t. *NIM %*, iii) traffic with SMP varies w.r.t. *Modif%*, regardless of the amount of non-identical messages *NIM %*, while ii) DSM optimizes traffic w.r.t. both *NIM%* and *Modifs%*. A compact representative depiction of network traffic variation in Fig. 12, based on the graphs in Fig. 11, for fixed average *NIM %* and *Modifs %* values, shows that the traffic gap between DSM, SMP, traditional multicast, and most evidently unicast, grows noticeably with the increasing number of messages. Results show that DSM underlines an average 20% traffic reduction in comparison with SMP.

5.2 Processing Time

Timing experiments were carried out on a PC with an Intel Xeon 2.66 GHz processor with 4GB RAM. Here, we evaluate the time complexity of DSM's core message aggregation and reconstruction operations, and compare it to SMP's message aggregation process (Phan K.A., Tari Z. *et al.* 2008), traditional multicast (automaton-based component, e.g., (Takeuchi Y., Okamoto T. *et al.* 2005), for identifying identical/different messages), and unicast (simple tree automaton for verifying SOAP message integrity prior to transmission).

On one hand, timing results in Fig. 13.a show that our approach is linear in the size of each SOAP message tree, which equally underlines a polynomial (quadratic) dependency on the combined size of both trees being compared (Fig. 13.b), confirming our complexity analysis. On the other hand, results in Fig. 13.b also show that our method induces an average 30% reduction in processing overhead in comparison with SMP. Results similar to those in Fig. 13.b are obtained when fixing message size and varying the total number of SOAP messages being processed.

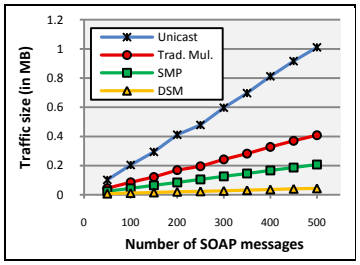
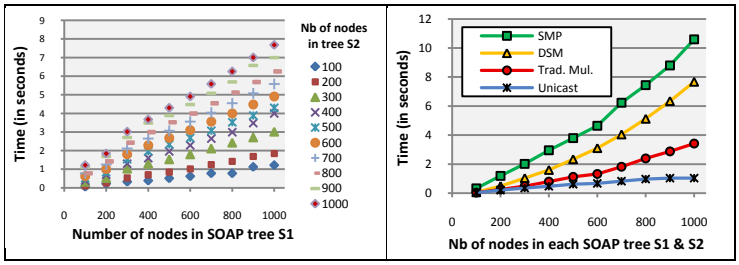


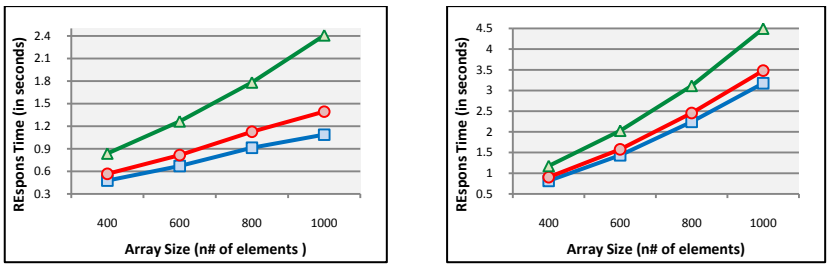
Fig. 12. Network traffic, with NIM%=20 and Modif%=10.



a. DSM time results. **b.** Comparative time results

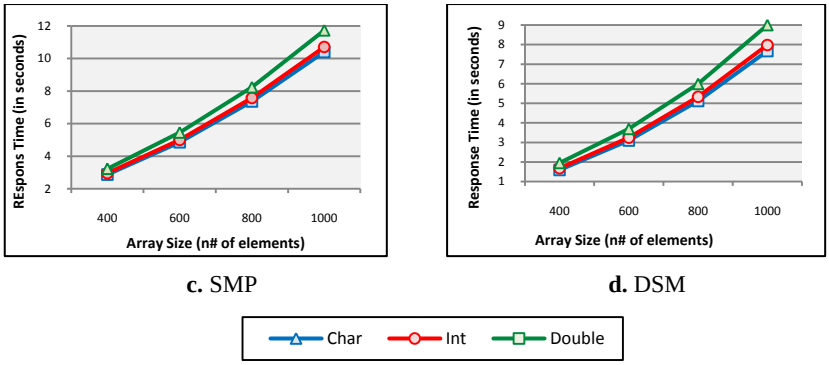
Fig. 13. Timing analysis.

In addition, we conducted several tests to evaluate processing time when exchanging different kinds of SOAP messages handling different data-types. Synthetic SOAP messages made of Character, Integer and Double arrays of varying sizes (ranging from 400 to 1000 SOAP elements per array) were utilized in this simulation experiment. We evaluated end-to-end response time (SOAP latency), i.e., the time perceived by a client to obtain a reply for a SOAP request for a web service. This includes (in addition to multicast processing): serialization time at the sender side, as well as parsing and de-serialization time at the receiver side (cf. overall architecture in Fig. 2[1]). We disregard network delays and service execution time in this evaluation (considering the scenario where the sender/receiver are run on the same host), in order to solely depict the results corresponding to SOAP latency.



a. Unicast **b.** Traditional Multicast

[1] We do not consider security policy evaluation in our current tests.



c. SMP **d.** DSM

Fig. 14. Response time, when manipulating different data-types

Fig. 14 shows the average time results normalized based on two traditional SOAP toolkits, including the SOAP Microsoft (Visual Basic) toolkit (Davis D. and Parashar M. 2002) and gSOAP (Van Engelen R.A. and K. Gallivan K. 2002), coupled with each of the DSM, SMP, traditional multicast, and unicast methods.

On one hand, results in Fig. 14 show that the time performance gap increases consistently when exchanging numeric data of type Double, in comparison with Integer and Character-based SOAP messages. The time gap is most significant with unicast (Fig. 14.a, where Double arrays induce an average 118% processing overhead w.r.t. Char) and traditional multicast (Fig. 14.b, with an average 43% overhead over Char). This is probably due to the expensive process of converting in-memory numeric data of type Double to-and-from ASCII-based (XML) SOAP format (as discussed in previous studies on SOAP performance, cf. Background in Section 2, e.g., (Chiu K., Govindaraju M. *et al.* 2002) where the translation between in-memory numeric data of type Double and the ASCII-based XML representation format has been shown to consume over 90% of the end-to-end SOAP message processing time). Note that the time performance gap decreases with SMP (Fig. 14.c, with an average 12.5% overhead between Double and Char) and DSM (Fig. 14.b, with an average 17.2% overhead) due to the extra processing involved in both protocols in performing similarity-based multicasting (i.e., similarity evaluation, multicast message aggregation, and SOAP reconstruction), which tends to reduce the impact of data-type conversion.

On the other hand, results in Fig. 14 concur with the timing results in Fig. 13.b where unicast and traditional multicast supersede SMP and DSM in time performance, whereas DSM (our approach, Fig. 14.d) induces an average reduction of 34% processing overhead (with all three data-types) in comparison with SMP (Fig. 14.c)

We are currently conducting experiments to fine-tune (optimize) DSM's performance, varying: i) the SOAP message aggregation similarity threshold $Thresh_{Sim}$, ii) the number and sizes of multicast buffer pools, and iii) the buffer pool time frame T_{Pool}. We aim to identify the set of input parameter values most adapted for different kinds of SOAP messages (encoding numeric data-types, type arrays of varying sizes, etc.), using different SOAP-based benchmarks, e.g., (Head M.R., Govindaraju M. *et al.* 2005; Head M.R., Govindaraju M. *et al.* 2006), and exploring various multicast scenarios (w.r.t. the n# of clients, routing algorithm, and network topology, such as in (Phan K., Bertok P. *et al.* 2009)).

6 PROOFS & PROPERTIES OF THE PROPOSED SIMILARITY AND FILTER FUNCTIONS

6.1 Component Filter Functions

6.1.1 Node-Edge Filter Function

Lemma 1. Given two SOAP trees S_i and S_j, and corresponding node-edge (parent-child) relations vectors $\overrightarrow{V_i}$ and $\overrightarrow{V_j}$ (cf. Definition 4), the node-edge filter function $Sim_{n\text{-}edge}(S_i, S_j)$ is an upper bound of our tree edit distance similarity measure (cf. Section 4.2.2), $Sim_{n\text{-}edge}(S_i, S_j) \geq Sim_{TED}(S_i, S_j)$, having:

$$\mathrm{Sim}_{\text{n-edge}}\left(\mathrm{S_i}, \mathrm{S_j}\right) = 1 - \frac{\frac{1}{2}\sum_{r=1}^{E} \left| \mathrm{w}_{\overrightarrow{\mathrm{V_i}}}(\mathrm{e_r}) - \mathrm{w}_{\overrightarrow{\mathrm{V_j}}}(\mathrm{e_r}) \right|}{|\mathrm{S_i}| + |\mathrm{S_j}|} \qquad \text{cf. formula } \mathbf{(2)}$$

$$\text{with } Manh(\overrightarrow{V_i}, \overrightarrow{V_j}) = \sum_{r=1}^{E} \left| \mathrm{w}_{\overrightarrow{\mathrm{V_i}}}(\mathrm{e_r}) - \mathrm{w}_{\overrightarrow{\mathrm{V_j}}}(\mathrm{e_r}) \right| \quad \text{and} \quad \mathrm{Sim}_{\mathrm{TED}}(\mathrm{S_i}, \mathrm{S_j}) = 1 - \frac{TED(\mathrm{S_i}, \mathrm{S_j})}{|\mathrm{S_i}| + |\mathrm{S_j}|} \qquad \text{cf. formula } \mathbf{(4)}$$

In other words, the Manhattan distance $Manh(\overrightarrow{V_i}, \overrightarrow{V_j})$ divided by 2, underlines a lower bound of the edit distance:

$$\frac{Manh(\overrightarrow{V_i}, \overrightarrow{V_j})}{2} \leq TED(S_i, S_j) \qquad \mathbf{(6)}$$

Proof. Consider an edit script $Diff = \prec op_1, op_2, \ldots, op_k \succ$ transforming S_i to S_j following our tree edit distance measure (cf. Section 4.2.2). We proceed by induction over the length $k = |Diff|$. If $k=0$, i.e., $Diff = \varnothing$, then $Manh(\overrightarrow{V_i}, \overrightarrow{V_j}) = TED(S_i, S_j) = 0$ ($Sim_{n\text{-}edge} = Sim_{TED} = 1$). When extending *Diff* by a new edit operation op_n, the edit distance *TED* is increased by $Cost(op_n)=1$, i.e., the cost of the edit operation. Yet, the Manhattan distance varies as follows. The edit operation in our approach may be a leaf node insertion, leaf node deletion or a node update:

- The insertion operation increases the occurrence frequency of the parent-child edge being inserted (i.e., the corresponding edge vector weight), by a value of 1. As a result, $Manh(\overrightarrow{V_i}, \overrightarrow{V_j})$ increases by a value of 1.
- The deletion operation decreases the occurrence frequency of the parent-child edge being depleted by a value of 1. This increases $Manh(\overrightarrow{V_i}, \overrightarrow{V_j})$ by 1.
- The update operation affects the occurrence frequency of two parent-child edges: decreasing by 1 the weight of the edge corresponding to the old node, and increasing by 1 the weight of the edge corresponding to the new node. In other words, $Manh(\overrightarrow{V_i}, \overrightarrow{V_j})$ increases by a value of 2.

Based on the three points above, it follows that the Manhattan distance $Manh(\overrightarrow{V_i}, \overrightarrow{V_j})$ changes by at most 2 for each edit operation affecting a SOAP message trees, whereas TED changes exactly by a value of 1. Therefore, the inequality of formula (6) holds, and thus *Lemma 1* is proved □

6.1.2 Node-Order Filter Function

Lemma 2. Given two SOAP trees S_i and S_j, and corresponding node-order vectors $\overrightarrow{V_i}$ and $\overrightarrow{V_j}$ (cf. Definition 5), the node order filter function $Sim_{n\text{-}order}(S_i, S_j)$ is an upper bound of our tree edit distance similarity measure, $Sim_{n\text{-}order}(S_i, S_j) \geq Sim_{TED}(S_i, S_j)$, having:

$$\mathrm{Sim}_{\text{n-order}}\left(S_i, S_j\right) = 1 - \frac{\sum_{r=1}^{E} \left| w_{\overrightarrow{V_i}}(e_r) - w_{\overrightarrow{V_j}}(e_r) \right|}{|S_i| + |S_j|} \quad \textbf{(7)}$$

with $Manh(\overrightarrow{V_i}, \overrightarrow{V_j}) = \sum_{r=1}^{E} \left| w_{\overrightarrow{V_i}}(e_r) - w_{\overrightarrow{V_j}}(e_r) \right|$ and $\mathrm{Sim}_{\text{TED}}(S_i, S_j) = 1 - \frac{TED(S_i, S_j)}{|S_i| + |S_j|}$ cf. formula **(4)**

In other words, the Manhattan distance $Manh(\overrightarrow{V_i}, \overrightarrow{V_j})$ is a lower bound of the edit distance:

$$Manh(\overrightarrow{V_i}, \overrightarrow{V_j}) \leq TED(S_i, S_j) \quad \textbf{(8)}$$

Proof. Consider an edit script $Diff = \prec op_1, op_2, \ldots, op_k \succ$ transforming S_i to S_j following our tree edit distance measure. We proceed by induction over the length $k = |Diff|$. If $k=0$, i.e., $Diff = \varnothing$, then $Manh(\overrightarrow{V_i}, \overrightarrow{V_j}) = TED(S_i, S_j) = 0$ ($Sim_{n\text{-}order} = Sim_{TED} = 0$). When extending *Diff* by a new edit operation op_n, the edit distance *TED* is increased by $Cost(op_n)=1$, i.e., the cost of the edit operation. Yet, the Manhattan distance varies as follows. The edit operation following our approach may be a leaf node insertion, leaf node deletion or a node update:

- The insertion operation increases the occurrence frequency of the node order score being inserted (i.e., the corresponding node order vector weight), by a value of 1. As a result, $Manh(\overrightarrow{V_i}, \overrightarrow{V_j})$ increases by a value of 1.
- The deletion operation decreases the occurrence frequency of the node order score being depleted by a value of 1. This increases $Manh(\overrightarrow{V_i}, \overrightarrow{V_j})$ by 1.
- The update operation does not affect node order occurrence frequency, the latter remaining the same before and after the operation takes place. In other words, $Manh(\overrightarrow{V_i}, \overrightarrow{V_j})$ does not change.

Based on the three points above, it follows that the Mahattan distance $Manh(\overrightarrow{V_i}, \overrightarrow{V_j})$ changes by at most 1 for each edit operation affecting a SOAP message trees, whereas TED changes exactly by a value of 1. Therefore, the inequality of formula (8) holds, and thus *Lemma 2* is proved □

6.1.3 Node-Value Filter Function

Lemma 3. Given two SOAP trees S_i and S_j, and corresponding node-value vectors $\overrightarrow{V_i}$ and $\overrightarrow{V_j}$ (cf. Definition 6), the node value filter function $Sim_{n\text{-}value}(S_i, S_j)$ is an upper bound of our tree edit distance similarity measure, $Sim_{n\text{-}value}(S_i, S_j) \geq Sim_{TED}(S_i, S_j)$, having:

$$\mathrm{Sim}_{\text{n-value}}\left(S_i, S_j\right) = 1 - \frac{\frac{1}{2}\sum_{r=1}^{E} \left| w_{\overrightarrow{V_i}}(e_r) - w_{\overrightarrow{V_j}}(e_r) \right|}{|S_i| + |S_j|} \quad \text{cf. formula } \textbf{(2)}$$

with $Manh(\overrightarrow{V_i}, \overrightarrow{V_j}) = \sum_{r=1}^{E} \left| w_{\overrightarrow{V_i}}(e_r) - w_{\overrightarrow{V_j}}(e_r) \right|$ and $\mathrm{Sim}_{\text{TED}}(S_i, S_j) = 1 - \frac{TED(S_i, S_j)}{|S_i| + |S_j|}$ cf. formula **(4)**

In other words, the Manhattan distance $Manh(\overrightarrow{V_i}, \overrightarrow{V_j})$ is a lower bound of the edit distance:

$$\frac{Manh(\overrightarrow{V_i}, \overrightarrow{V_j})}{2} \leq TED(S_1, S_2) \quad \text{cf. formula } \textbf{(6)}$$

Proof. Consider an edit script $Diff = \prec op_1, op_2, \ldots, op_k \succ$ transforming S_i to S_j following our tree edit distance measure. We proceed by induction over the length $k = |Diff|$. If $k=0$, i.e.,

Diff=∅, then $Manh(\vec{V_i}, \vec{V_j}) = TED(S_i, S_j) = 0$ ($Sim_{n\text{-}value} = Sim_{TED}=0$). When extending *Diff* by a new edit operation op_n, the edit distance *TED* is increased by $Cost(op_n)=1$, i.e., the cost of the edit operation. Yet, the Manhattan distance varies as follows. The edit operation following our approach may be a leaf node insertion, leaf node deletion or a node update:

- The insertion operation increases the occurrence frequency of the node value being inserted (i.e., the corresponding node value vector weight), by a score of 1. As a result, $Manh(\vec{V_i}, \vec{V_j})$ increases by 1.
- The deletion operation causes a decrease in the occurrence frequency of the node value vector weight corresponding to the node value being deleted, by a score of 1. This increases $Manh(\vec{V_i}, \vec{V_j})$ by 1.
- The update operation affects the occurrence frequency of two node value weights: decreasing by 1 the weight of the node value corresponding to the old node, and increasing by 1 the weight of the node value corresponding to the new node. In other words, $Manh(\vec{V_i}, \vec{V_j})$ increases by a score of 2.

Based on the three points above, it follows that the Manhattan distance $Manh(\vec{V_i}, \vec{V_j})$ changes by at most 2 for each edit operation affecting a SOAP message trees, whereas TED changes exactly by a value of 1. Therefore, the inequality of formula (6) holds, and thus *Lemma 3* is proved □

6.2 Combined Filter Function

Recall the combined filter similarity function in formula (3):

$$\mathrm{Sim}_{\mathrm{Filter}}(\mathrm{S}_1, \mathrm{S}_2) = \frac{1}{\sum_{h \in F}\left(1-\nabla_h^{\,2}\right)} \sum_{f \in F}\left(1-\nabla_f^{\,2}\right) Sim_f(\mathrm{S}_1, \mathrm{S}_2) \quad \in [0,1]$$

Lemma 4. Given two SOAP trees S_i and S_j, the combined filter function Sim_{Filter} in formula (3) is an upper bound of our tree edit distance similarity measure, $Sim_{Filter}(S_i, S_j) \geq Sim_{TED}(S_i, S_j)$.

Proof. The combined filter similarity comes down to the weighted sum of its component filter measures:

$$Sim_{Filter}(\mathrm{S_i}, \mathrm{S_j}) = \sum_{f \in F} w_f \; Sim_f(\mathrm{S_i}, \mathrm{S_j}) \quad \in [0,1] \qquad \textbf{(9)}$$

where *F={n-edge, n-order, n-value}* is the set of component filters, and:

$$w_f = \frac{\left(1-\nabla_f^{\,2}\right)}{\sum_{h \in F}\left(1-\nabla_h^{\,2}\right)} \quad \text{such as} \quad \sum_{f \in F} w_f = 1 \;\text{ and }\; \forall w_f \geq 0 \qquad \textbf{(10)}$$

where $\nabla_f^{\,2}$ is the variance over all SOAP trees according to the *f*-th filter function within the SOAP tree candidate set.

Consequently, for all SOAP trees S_i and S_j in the candidate set:

$$\text{For each } f \in F,\ Sim_f(\mathrm{S_i}, \mathrm{S_j}) \geq Sim_{TED}(\mathrm{S_i}, \mathrm{S_j}) \;\Rightarrow$$

$$\sum_{f \in F} w_f \; Sim_f(\mathrm{S_i}, \mathrm{S_j}) \;\geq\; Sim_{TED}(\mathrm{S_i}, \mathrm{S_j}) \;\Rightarrow$$

$$Sim_{Filter}(S_i, S_j) \geq Sim_{TED}(S_i, S_j)$$

Hence, *Lemma 4* is proved □

7 CONCLUSION

In this paper, we describe a new framework for Differential SOAP Multicasting (DSM). It consists in identifying the common pattern and differences between SOAP messages, and multicasting those messages that are most similar. The groundwork and overall architecture of DSM have been described in (Tekli J., Damiani E. *et al.* 2011a). This paper's contribution extends the latter publication, developing DSM's *filter-differencing* module: the filter functions and the tree edit distance measure. Also, we describe our machine-readable differencing language *SDL* (*Simple Diff Language*), which was omitted from (Tekli J., Damiani E. *et al.* 2011a), and present additional simulation experiments. Results show that our approach outperforms its alternative, SMP (Phan K.A.; Tari Z.; and Bertok P. 2008), and minimizes network traffic in comparison with traditional multicast and unicast. Our technique readily lends itself to seamless integration with well-known optimizations of underlying protocols, e.g. by sending SOAP multicasts over persistent HTTP connections on high-latency networks (Kangasharju J.; Tarkoma S. and Raatikainen K. 2003).

In our future work, we also plan to make use of tight software integration architectures, such as in (Kostoulas M. G., Matsa M. *et al.* 2006; Zhang W. and Van Engelen R. A. 2006), so as to avoid repeated/unnecessary data processing (in serialization and multicasting, and in parsing and de-serialization, minimizing interference between the different techniques involved in SOAP message exchange), copying to/from memory buffers, and expensive data-type transformations (ASCII/UTF to in-memory types, and vice-versa). This would prove essential in improving overall multicast time response, namely when handling bulky scientific data such as arrays of integers and doubles. We also plan to investigate multicasting of secure SOAP messages, to improve performance in the evaluation of WS security policies (Turkmen F. and Crispo C. 2008), which remains a virtually unexplored topic to this date.

REFERENCES

Abu-Ghazaleh N. and Lewis M.J. (2005). *Differential Deserialization for Optimized SOAP Performance*. Proceedings of the ACM/IEEE Conference on Supercomputing pp. 21-31, Seattle.

Abu-Ghazaleh N., Lewis M.J., et al. (2004). *Differential Serialization for Optimized SOAP Performance*. Proceedings of the 13th International Symposium on High Performance Distributed Computing (HPDC'04) pp. 55-64.

Azzini A., Marrara S., et al. (2009). *Extending the Similarity-Based XML Multicast Approach with Digital Signatures*. Proc. of the 2009 ACM Workshop on Secure Web Services (SWS'09) pp. 45-52, Chicago.

Bille P. (2005). *A Survey on Tree Edit Distance and Related Problems*. Theoretical Computer Science 337(1-3):217-239.

Bray T., Paoli J., et al. (2008). *Extensible Markup Language (XML) 1.0 - 5th Edition. W3C recommendation, 26 Novembre 2008*. Retrieved November 2010, from http://www.w3.org/TR/REC-xml/.

Buttler D. (2004). *A Short Survey of Document Structure Similarity Algorithms*. Proceedings of the International Conference on Internet Computing (ICOMP) pp. 3-9.

Chawathe S. (1999). *Comparing Hierarchical Data in External Memory*. Proceedings of the International Conference on Very Large Data Bases (VLDB) pp. 90-101.

Chawathe S., Rajaraman A., et al. (1996). *Change Detection in Hierarchically Structured Information*. Proc. of the ACM International Conference on Management of Data (SIGMOD) pp. 26-37. Montreal.

Cheney J. (2001). *Compressing XML with Multiplexed Hierarchical PPM Models*. In Proceedings of the Data Compression Conference pp. 163-173.

Chinnici R., Moreau J.J., et al. (2007). *Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language, W3C Recommendation 26 June 2007*. Retrieved 25 August 2010, from http://www.w3.org/TR/wsdl20/.

Chiu K., Govindaraju M., et al. (2002). *Investigating the Limits of SOAP Performance for Scientific Computing*. Proceedings of ACM International Symposium on High Performance Distributed Computing (HPDC) pp. 246-254, Edinburgh, Scotland.

Cobena G., Abiteboul S., et al. (2001). *Xydiff, tools for detecting changes in XML documents*. http://wwwrocq.inria.fr/˜cobena/XyDiffWeb/.

Cobéna G., Abiteboul S., et al. (2002). *Detecting Changes in XML Documents*. Proceedings of the IEEE International Conference on Data Engineering (ICDE) pp. 41-52.

Dalamagas T., Cheng T., et al. (2006). *A Methodology for Clustering XML Documents by Structure*. Information Systems 31(3):187-228.

Damiani E. and Marrara S. (2008). *Efficient SOAP Message Exchange and Evaluation Through XML Similarity*. Proceedings of the 2008 ACM workshop on Secure Web Services (SWS'08) pp.29-36.

Davey B. A. and Priestley H. A. (2002). *Introduction to Lattices and Order (2nd Edition).* Cambridge University Press, pp. 310.

Davis D. and Parashar M. (2002). *Latency Performance of SOAP Implementations*. Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid pp. 407-412.

Devaram K. and Andersen D. (2002). *SOAP Optimization via Parameterized Client-Side Caching* Proc. of the IEEE/ACM 2nd International Symposium on Cluster Computing and the Grid (CCGRID'02) pp.439-312.

Gannon D., Krishnan S., et al. (2004). *On Building Parallel and Grid Applications: Component Technology and Distributed Services*. In Proc. of the Second International Workshop on Challenges of Large Applications in Distributed Environments (CLADE ’04) IEEE Computer Society, p. 44, Washington DC, USA.

Head M.R., Govindaraju M., et al. (2005). *A Benchmark Suite for SOAP-based Communication in Grid Web Services*. In Proceedings of the ACM/IEEE Conference on Supercomputing (SC'05) pp. 19.

Head M.R., Govindaraju M., et al. (2006). *Benchmarking XML Processors for Applications in Grid Web Services*. In Proceedings of the ACM/IEEE Conference on Supercomputing (SC'06) pp. 30, Seatle, WA.

Kailing K., Kriegel H.P., et al. (2004). *Efficient Similarity Search for Hierarchical Data in Large Databases*. Proceedings of the International Conference on Extending Database Technology pp. 676-693.

Kangasharju J.; Tarkoma S. and Raatikainen K. (2003). *Comparing SOAP Performance for Various Encodings, Protocols, and Connections.* Proceedings of the IFIP 8th International Conference, PWC'03, LNCS 2775/03 pp. 397-406.

Kohlhoff C. and Steele R. (2003). *Evaluating SOAP for High Performance Business Applications: Real-Time Trading Systems*. Proceedings of the World Wide Web (WWW) Conference Budapest, Hungary.

Komvoteas K. (2003). *XML Diff and Patch Tool*. MS in Distributed Multimedia and Information Systems Dissertation, Edinburgh, Scotland: Heriot-Watt University.

Korn F., Sidiropoulos N., et al. (1998). *Fast and Effective Retrieval of Medical Tumor Shapes. I*. IEEE TKDE 10, pp. 889–904.

Kostoulas M. G., Matsa M., et al. (2006). *XML Screamer: An Integrated Approach to High Performance XML Parsing, Validation and Deserialization*. In Proceedings of the 15th International Conference on World Wide Web (WWW ’06) pp. 93–102.

Krause E.F. (1987). *Taxicab Geometry - An Adventure in Non-Euclidean Geometry*. Dover Publications - NY, pp. 88.

Kriegel H.P. and Schönauer S. (2003). *Similarity Search in Structured Data*. Proceedings of the 5th International Conference on Data Warehousing and Knowledge Discovery (DaWaK) pp. 309-319.

Laux A. and Martin L. (2000). *XUpdate Working Draft.* XML:DB Initialtive.

Liefke H. and Suciu D. (2000). *XMill: An Efficient Compressor for XML Data.* University of Pennsylvania Technical Report MSCIS-99-26.

Ma Y. and Chbeir R. (2005). *Content and Structure Based Approach for XML Similarity*. Proceedings of the IEEE International Conference on Computer and Information Technology (CIT) pp. 136-140.

Makino S., Tatsubori M., et al. (2005). *Improving WS-Security Performance with a Template-Based Approach*. Proceedings of the IEEE INternational Conference on Web Services (ICWS'05) pp. 581-588.

Marian A., Abiteboul S., et al. (2001). *Change-Centric Management of Versions in an XML Warehouse*. Proceedings of the International Conference on Very Large Data Bases (VLDB) pp. 581-590.

McGill M. (1983). *Introduction to Modern Information Retrieval*. McGraw-Hill, New York.

Megginson D. et al. (2004). *The Simple API for XML - SAX 2.0.2*. Retrieved February 2011, from http://www.megginson.com/SAX/.

Miller G. (1990). *WordNet: An On-Line Lexical Database*. International Journal of Lexicography 3(4).

Monsell EDM ltd. (2000). *DeltaXML*. http://deltaxml.com Retrieved December 2009.

Mouat A. (2002). *XML Diff and Patch Utilities*. CS4 Dissertation. Edinburgh Scotland: Heriot-Watt University.

Nierman A. and Jagadish H. V. (2002). *Evaluating structural similarity in XML documents*. Proceedings of the ACM SIGMOD International Workshop on the Web and Databases (WebDB) pp. 61-66.

Phan K.A., Bertok P., et al. (2009). *Minimal Traffic-Constrained Similarity-Based SOAP Multicast Routing Protocol*. OTM Confederated International Conferences LNCS 4803, pp. 558-576.

Phan K.A., Tari Z., et al. (2008). *Similarity-Based SOAP Multicast Protocol to Reduce Bandwidth and Latency in Web Services*. IEEE Transactions on Services Computing Vol 1, No 2, pp. 88-103.

Phan K.A.; Tari Z.; and Bertok P. (2008). *Similarity-Based SOAP Multicast Protocol to Reduce Bandwidth and Latency in Web Services*. IEEE Transactions on Services Computing Vol 1, No 2, pp. 88-103.

Salton G. (1989). *Automatic text processing: the transformation, analysis, and retrieval of information by computer*. Addison-Wesley Longman, Boston, MA, USA pp. 530.

Singh G., Bharathi S., et al. (2003). *A Metadata Catalog Service for Data Intensive Applications*. In Proc. of the ACM/IEEE Conference on Supercomputing. IEEE Computer Society, 2003, p. 33, Washington DC, USA.

Slominski A. (2004). *XSOAP*. Retrieved February 2010, from http://www.extreme.indiana.edu/xgws/xsoap/.

Suzumura T., Takase T., et al. (2005). *Optimizing Web Services Performance by Differential Deserialization* Proceedings of the IEEE International Conference on Web Services (ICWS'05) Vol. 1, pp.185- 192.

Takeuchi Y., Okamoto T., et al. (2005). *A Differential-Analysis Approach for Improving SOAP Processing Performance*. Proceedings of the IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE'05) pp. 472-479.

Tekli J., Chbeir R., et al. (2009). *An Overview of XML Similarity: Background, Current Trends and Future Directions*. Elsevier Computer Science Review 3(3):151-173.

Tekli J., Damiani E., et al. (2011a). *Differential SOAP Multicasting*. Proceedings of the 9th IEEE International Conference on Web Services (ICWS'11) Washington DC, USA.

Tekli J., Damiani E., et al. (2011b). *SOAP Processing Performance and Enhancement*. To appear in IEEE Transactions on Service Computing (IEEE TSC).

Teraguchi M., Makino S., et al. (2006). *Optimized Web Services Security Performance with Differential Parsing*. Proceedings of the 4th International Conference on Service-Oriented Computing (ICSOC'06) pp. 277-288.

Turkmen F. and Crispo C. (2008). *Performance evaluation of XACML PDP implementations*. Proceedings of the ACM workshop on Secure Web Services (SWS), Alexandria, Virginia, USA. pp. 37-44.

Van Engelen R.A. and K. Gallivan K. (2002). *The gSOAP Toolkit for Web Services and Peer-To-Peer Computing Networks.* In Proceedings of the 2nd IEEE International Symposium on Cluster Computing and the Grid (CCGrid2002) pp. 128-135, Berlin, Germany.

W3 Consortium. (2007). *SOAP Version 1.2*. W3C Recommendation (Second Edition) Retrieved February 2010, from http://www.w3.org/TR/soap/.

W3C Consortium. (2005). *The Document Object Model*. Retrieved 28 May 2009, from http://www.w3.org/DOM.

Werner C., Buschmann C., et al. (2005). *WSDL-Driven SOAP Compression*. International Journal of Web Services Research Vol. 2, Issue 1, pp. 18-35.

Zhang B., Jamin S., et al. (2002). *Host Multicast: A Framework for Delivering Multicast to End Users*. Proceedings of the IEEE Conference on Computer Communications (INFOCOM'02) pp. 1366-1375.

Zhang K. and Shasha D. (1989). *Simple Fast Algorithms for the Editing Distance between Trees and Related Problems*. SIAM Journal of Computing 18(6):1245-1262.

Zhang W. and Van Engelen R. A. (2006). *A Table-Driven Streaming XML Parsing Methodology for High-Performance Web Services*. In Proceedings of the IEEE International Conference on Web Services (ICWS'06) pp. 197–204.

ABOUT THE AUTHORS

Joe Tekli is an assistant professor at the Faculty of Engineering - Antonine University (UPA), Lebanon (since Sept. 2011). He has completed three post-doctoral missions: in the Institute of Computer Science and Statistics - University of Sao Paulo (USP), Brazil (Sept. 2010 – Aug. 2011), in the Department of Computer Science - University of Shizuoka, Japan (spring 2010), and in the Department of Science and Technology - University of Milan (UniMi), Italy (Fall 2009). He holds a holds a PhD in CS from the University of Bourgogne, LE2I UMR-CNRS, France, awarded (in Oct. 2009) with Highest Honors. He also holds a Research Masters in CS from the University of Bourgogne (July 2006), and a Masters of Engineering in Telecommunications from the Antonine University, Lebanon (July 2005), both acquired with Honors (ranked top of his class in both programs). He has been awarded various prestigious postdoctoral fellowships, of the FAPESP (Brazil), JSPS (Japan), and Fondazione Cariplo (Italy). He was also awarded a PhD Fellowship of the Ministry of Education (France), and a Masters Scholarship of the AUF (France-Lebanon). His research activities cover XML processing, web services, data semantics and taxonomies, data clustering and classification, RSS integration, and multimedia fragmentation. He is a member of IEEE and ACM SIGAPP French Chapter. He is an organizing member of various international conferences such as ICDIM, SITIS, MEDES and ACM SAC'06. His research results have been published in various international journals and conferences (e.g., IEEE TSC, Computer Science Review, WWW Journal, IEEE ICWS, ER, SBBD, WISE, ADBIS, COMAD, etc.)

Ernesto Damiani is a professor at Università degli Studi di Milano and the director of the same University 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 has served and is serving in all capacities on many congress, conference, and workshop committees. He is a senior member of the IEEE. In 2008 he was nominated ACM distinguished scientist and he received the Chester Hall Award from the IEEE Societty on Consumer Electronics. Web page www.dti.unimi.it/~damiani.

Dr. Richard Chbeir received his PhD in Computer Science from the University of INSA-FRANCE in 2001. The author became a member of IEEE since 1999. He is currently an Associate Professor in the Computer Science Department of the Bourgogne University, Dijon-France. His research interests are in the areas of distributed multimedia database management, XML similarity and rewriting, spatio-temporal applications, indexing methods, multimedia access control models, security and watermarking. Dr. CHBEIR 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, etc.), conferences (ER, WISE, SOFSEM, EDBT, ACM SAC, Visual, IEEE CIT, FLAIRS, PDCS, etc.), and has served on the program committees of several international conferences (ICDIM, IEEE SITIS, ACM SAC, IEEE ISSPIT, EuroPar, SBBD, etc.). He has been organizing many international conferences and workshops (ICDIM, CSTST, SITIS, etc.). He is currently the Chair of the French Chapter ACM SIGAPP and the vice-chair of ACM SIGAPP.