Compressing SOAP Messages by using Pushdown Automata

Christian Werner, Carsten Buschmann, Ylva Brandt, Stefan Fischer
Institute of Telematics
University Lübeck
Ratzeburger Allee 160
D-23538 Lübeck, Germany
Email: {werner | buschmann | brandt | fischer}@itm.uni-luebeck.de

***Abstract*— In environments with limited network bandwidth or resource-constrained computing devices the high amount of protocol overhead caused by SOAP is disadvantageous. Therefore, recent research work concentrated on more compact, binary representations of XML data. However, due to the special characteristics of SOAP communication most of these approaches are not applicable in the field of web services. First, we give a detailed overview of the latest developments in the field of XML data compression. Then we will introduce a new approach for compressing SOAP data which utilizes information on the structure of the data from an XML Schema or WSDL document to generate a single custom pushdown automaton. This cannot only be used as a highly efficient validating parser but also as a compressor: its transitions are tagged with short binary identifiers which replace XML tags during compression. This approach leads to extremely compact data representations as well as low memory and CPU utilization.**

I. Introduction

Since SOAP is a dialect of XML, it suffers from the fact that SOAP messages are significantly bigger than binary representations. Comparisons on different approaches for realizing Remote Procedure Calls (RPC) have shown that SOAP over HTTP uses significantly more bandwidth than competitive technologies. The transmitted data is about three times as big as for Java RMI or CORBA [1].

Though today's wired networks are powerful enough to provide sufficient bandwidth even for very demanding applications, there are still some fields of computing where bandwidth is costly. In cellular phone networks, for instance, it is quite common to charge the customer according to the transmitted data volumes. Also when using dial-up connections via modem or ISDN, which are still common in many enterprise networks, bandwidth is very limited. Even more important is the economical use of bandwidth in energy-constrained environments: since the radio interface is usually a main power consumer on mobile devices, tight restrictions apply to the transmitted data volumes.

Binary representations of XML data have gained a lot of interest in the last two years. Therefore, the W3C XML Binary Characterization Working Group has been founded in March 2004. Its members conducted a detailed requirement analysis for binary XML representations and created a survey of the existing approaches in this field [2]. This working group has specified a set of properties that are important for binary XML representations. Besides compactness the main aspects that have been evaluated are support for directly reading and writing the binary XML data, independence of transport mechanisms, and processing efficiency.

As a major outcome the XML Binary Characterization Working Group created a set of 18 typical use cases for binary XML representations and analyzed their requirements. It is notable that in all use cases the property "compactness", which will be in the focus of this paper, has been rated at least as a nice-to-have feature. In ten use cases this property was even rated as mandatory.

In December 2005 another W3C working group has been established that will focus on the interoperability of binary XML: the Efficient XML Interchange Working Group [3].

Up to now both of the W3C working groups in this field have not drafted any recommendations. Currently they are still discussing the requirements of interoperable binary XML representations.

In this paper we will elaborate on how to compress SOAP messages efficiently. In particular, we will present a new compression approach that exploits the fact that XML network messages are usually described by an XML grammar which is known to both the sender and the receiver. For SOAP messages this grammar can be derived from the WSDL document of the web service. Most of the information contained in the message can be inferred from this XML grammar and hence can be omitted during transmission.

The remainder of this paper is structured as follows: In Section II we will give a survey on the compression of SOAP and XML, with a focus on techniques that make use of the availability of grammar descriptions. We then present the results of an extensive evaluation measuring the compression effectiveness of the different algorithms. In section III we will present our approach to SOAP compression, and compare the resulting compression ratios to those of related algorithms. The paper is concluded by a summary and directions for future work.

II. Survey on XML Compression

As XML documents are commonly represented as text, the naive approach for generating more compact representations is to apply well-known general purpose text compressors like

IEEE International Conference on Web Services (ICWS'06)
0-7695-2669-1/06 $20.00

gzip. Unfortunately these perform rather poorly, especially when exposed to short input files, which is a typical case in the domain of web services [1].

A. Overview of XML Compression

With the growing popularity of XML more sophisticated XML compression concepts were developed. They separate the markup from the character data in a document and compress both independently with different algorithms. This technique is used by *XMill* [4] and *xmlppm* [5]. Additionally, the XML syntax rules can be exploited for data compression: in all well-formed documents the name of any end tag can be inferred from the name of the corresponding opening tag and hence, the name of an end tag can be omitted. This technique is applied by the *Fast Infoset* [6] compressor, which produces a binary serialization that is optimized with a special focus on processing speed. A most recent approach in the field of non-Schema-aware XML compressors is *Exalt* [7]. This compressor learns about typical tag sequences in the input document and stores this information in automata structures. From these automata *Exalt* can predict the next tag at a certain stage of compression and encodes only the difference between the prediction and the read value.

Another group of XML compressors is custom-tailored for selected XML languages. By knowing the vocabulary from a fixed grammar description, it is possible to create a highly specialized compressor that maps the markup-structures of this language to shorter binary tokens using fixed built-in coding tables. WBXML [8] and Millau [9] are such tools supporting various languages in the field of mobile devices like WML or SyncML. A similar approach is the *Binary Format for MPEG-7 Metadata (BiM)* [10] which is a specialized compressor for MPEG-7 Metadata.

Although such highly specialized Schema-based XML compressors exhibit very promising compression results [1], their value for practical applications is limited. Their most severe disadvantage is that they do not support the extensibility XML has been designed for. SOAP is a typical example of this problem: The SOAP body may carry all kinds of application specific data. Hence, it cannot be encoded using fixed coding tables.

B. XML Compressors with Dynamic Schema Processing

To overcome this limitation while still leveraging the outstanding performance of Schema-based compression, researchers started to develop compressors that can be customized to application specific XML grammars.

If information on the document structure is available through an XML grammar description like XML Schema or DTD, two additional compression strategies become available. First, efficient binary content encodings can be inferred from the data type definitions in the grammar. For example, all numeric values in the XML document could be represented as 32 bit integer numbers instead of using verbose text representations. Second, the grammar prescribes how valid instance documents are structured. Therefore, parts of the structure information can be omitted in the instance document. Of course, the Schema information is also required for reconstructing the original message from the compressed representation.

Although the *XGrind* compressor [11] employs the corresponding DTD when processing an XML file, it neither omits tags that can be inferred from the grammar nor uses efficient binary encodings for numeric content like `xsd:int` or `xsd:DateTime` etc. The reason for this is that it focuses on so called context-free compression, i.e. XGrind allows for parsing and querying selected parts of the binary encoded file without decompressing it.

Anyhow, even under the constraint of context-free compression XGrind implements some features for generating compact XML representations. It uses shorter identifiers that represent the text values of tag and attribute names: It uses the DTD to identify all possible tag names, which are then mapped to compact 8 bit identifiers in the binary encoding. Additionally, the names of closing tags are omitted since in well-formed XML documents these can be inferred from the name of the corresponding opening tag. The possible values of enumeration types are also binary encoded using binary block codes. All other content is Huffman encoded in a per-tag-name fashion, i.e. for each element with a certain name a separate Huffman table is maintained.

A second approach for XML compression with optional dynamic grammar support is *Xebu* [12]. It encodes the sequence of SAX events generated by a parser. Again, on their first appearance tag names are indexed with a byte value which is then used as a short-hand pointer on repetition. Unlike XGrind, the Xebu encoder can detect numeric data and encodes it in a more compact binary format.

These two features of Xebu are also available if no grammar is present. In addition, so called omission automata can be generated from a Relax NG [13] grammar. These can then be used to omit the encoding of SAX events which can be inferred from the grammar on decoding. This approach increases compression efficiency, e.g. if the grammar prescribes a sequence with each element occurring exactly once. Unfortunately, the paper does not describe how these automata are generated from the grammar. Additionally, the Relax NG grammar can also be used for setting up the coding tables with initial values, since possible tag and attribute names are specified in the grammar. This feature is called pre-caching.

Another Schema-aware XML processor is *XML Xpress* [14]. It employs a two step processing scheme. In a first step a so called Schema Model file is generated offline from an XML Schema document and a set of sample XML files that represent typical data the compressor will be exposed to later on. Step two comprises the actual compression process and takes the Schema Model file and the XML source file to be compressed as input. The decompressor also uses the Schema Model file in order to decode the compressed representation. Because XML Xpress is a commercial product no details about the conversion processes are publicly available. Furthermore, it remains unclear if the generation of the Schema Model file

can be done automatically or requires manual intervention.

An approach which targets the special needs of SOAP is *Differential Encoding* presented in [1]. The authors employ a differential encoding algorithm to achieve compact SOAP message representations. Instead of sending the entire SOAP message only the difference between the message and a skeleton, which is previously generated from the WSDL service description, is transmitted. There are many data formats for describing differences between two XML documents. One very efficient solution is to use the Document Update Language (DUL) in combination with the xmlppm compressor [5].

A major benefit of this approach is that very high compression rates can be achieved. This holds also true for small files, i.e. smaller than 10 kBytes, which is a typical case for SOAP communication. Anyhow, the algorithmic complexity of XML differencing is quite high and therefore on resource-constrained devices this approach is only feasible with small documents.

C. Setup for the Performance Evaluation

In order to compare the SOAP compression performance of the different existing approaches, we have set up a test bed consisting of two web services which represent typical payload patterns of today's SOA applications.

We found that there is a large number of services exchanging small SOAP messages with only very little payload. The structure of the exchanged messages is described by an XML Schema, which is very restrictive. A typical example of this kind of service would be a stock quote application: Here we are sending a request message that contains the ticker symbol which is a very short string and the service then sends back a response message which holds the current quote as a numeric value. Such very short messages are hard to compress, because the compressor must not reserve much space in the output stream for transmitting large tables mapping symbols to their bit codes – such coding tables might easily need more space than the encoded data itself. As a result, a compressor that works very well on large files may perform poorly on small files.

Therefore, our first benchmark web service is a simple calculator application. We implemented four different service operations yielding messages with different amounts of payload: `void doNothing()`, `int increment(int i1)`, `int add(int i1, int i2)`, and `int add6ints(int i1, int i2, int i3, int i4, int i5, int i6)`. We have implemented this application as a literal-style web service using the Microsoft .NET platform. Each service operation was called with randomly-chosen integer values. We saved the resulting request and response messages to files, which were then passed to the compressors.

Anyhow, there are also some web services that exchange quite large messages containing a lot of string data. The structure of possible messages is described by a rather complex XML Schema definition, which heavily employs attributes to structure the transmitted information. A typical example of this kind of service is the Amazon E-Commerce Service [15].

Hence, we used the Amazon E-Commerce Service as our second test case. We issued three *ItemSearch* request messages with the search keyword "web service". In the first request, we set the ResponseGroup parameter to "small" which signals the web service to send back a non-verbose response message. In the second and third request message we set the ResponseGroup parameter to "medium" and "large" respectively which leads to much more detailed response messages. Again, we saved the resulting SOAP messages into files. They contain a fraction of 47% (small), 35% (medium) and 37% (large) `xsd:string` data while the rest consists of markup and numeric values.

In order to evaluate compressors with Schema support we also saved the WSDL descriptions of the two web services into files and extracted the enclosed XML Schema descriptions. These specify how the enclosed application specific data types in the body of the SOAP message are structured. In order to get an XML Schema description of the full recorded SOAP messages, i.e. including the SOAP Envelope and its child elements, we combined the extracted XML Schema descriptions with the XML Schema description of the SOAP 1.1 Envelope (available at: `http://schemas.xmlsoap.org/soap/envelope/`) by importing the extracted Schema definition for the application specific namespace and by modifying the data type of the body element.

Since Xebu is not able to process XML Schema files, we finally converted the XML Schema description of our SOAP messages into the Relax NG language using the tools Sun Relax NG Converter [16] and Trang [17].

D. Compression Performance on Typical SOAP Data

We analyzed the compression effectiveness of the previously described approaches when exposed to typical SOAP data. For all compressors we measured the resulting file sizes S, as well as the compression ratio λ, which is the quotient $S_{\text{compressed}}/S_{\text{uncompressed}}$. We also calculated the sum of all file sizes Σ for all compressors.

Table I shows the results. For better readability we omitted the single values for λ and denoted only the best, worst and average values here. Please note that the average compression ratio is not weighted by the file sizes and therefore this value differs from $\Sigma_{\text{compressed}}/\Sigma_{\text{uncompressed}}$.

All compressors were used in their default settings, i.e. without additional command line parameters. An exception is Xebu, which is not a ready-to-use application but a set of Java classes providing an API. We implemented a minimal compression application which makes use of both additional features that are available when an XML grammar is present: omission automata and pre-caching.

Obviously, it is not possible to evaluate WBXML, Millau and BiM in this setting since these compressors are not useable with SOAP. XML Xpress has not been experimentally evaluated because there is no publicly available implementation. The XGrind compressor could not be included,

TABLE I

COMPRESSION RESULTS FOR TYPICAL SOAP DATA. ALL FILE SIZES ARE SPECIFIED IN BYTES.

	uncompressed	gzip	XMill	xmlppm	Fast Infoset	Xebu	Diff. Enc. (DUL + xmlppm)
doNothing (Request)	336	224	338	167	210	103	21
doNothing (Response)	344	229	352	173	210	103	21
increment (Request)	381	236	334	177	246	127	52
increment (Response)	425	249	351	187	246	127	52
add (Request)	401	239	360	181	269	135	106
add (Response)	401	238	361	180	246	127	95
add6ints (Request)	559	307	436	242	397	217	155
add6ints (Response)	429	255	378	195	254	135	104
Amazon “small” (Request)	680	371	519	319	455	–	255
Amazon “small” (Response)	9,144	1,625	2,072	1,576	7,446	–	2,366
Amazon “medium” (Request)	681	373	518	321	456	–	257
Amazon “medium” (Response)	60,319	8,795	8,298	7,190	46,283	–	16,096
Amazon “large” (Request)	680	371	520	319	455	–	255
Amazon “large” (Response)	299,619	45,841	31,977	32,171	236,349	–	88,103
Σ	374,399	59,353	46,814	43,398	293,522	–	107,938
λ_{best}	1.00	0.15	0.11	0.11	0.58	0.30	0.06
λ_{worst}	1.00	0.67	1.02	0.50	0.81	0.39	0.38
$\lambda_{average}$	1.00	0.50	0.71	0.39	0.67	0.32	0.24

either, since the available implementation from July 19, 2002 (which can be downloaded at `http://sourceforge.net/projects/xgrind/`) did not perform as expected: when compressing SOAP messages, the decompressor generates invalid files from the compressed representation. Some implementation problems also occurred when using the Xebu implementation. It was not able to process the Amazon web service grammar. Hence, the Amazon test files were excluded from the Xebu evaluation.

Despite these difficulties with some implementations, which are obviously not ready for every day use, the results show that Schema-based compression yields very promising performance. Xebu and Differential Encoding feature the smallest average compression ratios. Their home turf is the efficient compression of small SOAP messages with a tight Schema description: for the calculator web service both yield the smallest message sizes. Due to the small size of the original messages gzip and XMill can hardly compress them because they have to embed extra information about the used encoding rules in addition to the actual data. For small files Fast Infoset achieves its best results, but the best ratio is only 0.58. The xmlppm approach shows the best performance possible without Schema information.

With the Amazon web service the situation is different: the Differential Encoding approach is outperformed by gzip, XMill and xmlppm when compressing the medium and large responses. These consist mainly of hardly predictable tags containing large amounts of text data. As a result, the resulting messages are not of competitive size. Fast Infoset performs even worse and again achieves only poor compression ratios.

All in all, it becomes clear that Schema-based compression seems to be the most promising way to achieve compact SOAP message encodings. However, no currently available compressor implementation can robustly yield high compression ratios for all kinds of files and file sizes.

III. USING PUSHDOWN AUTOMATA FOR SOAP COMPRESSION

Existing XML compression approaches with dynamic Schema support suffer from the limitation that they either use very expensive computations (differential encoding) or that they make use of dynamically growing data structures like coding tables (Xebu). Especially in the filed of resource-constrained devices both of these facts are disadvantageous.

Hence, our major design goal was to develop an approach that features low computational complexity and uses only very litte memory while still yielding excellent compression ratios. Existing approaches like Exalt, BiM and Xebu show that automata may be beneficial for representing structure information of XML data.

Anyhow, a plain finite-state automaton (FSA) as utilized in Exalt, BiM and Xebu cannot fully represent the grammar described by an XML Schema document, because languages described by a DTD or XML Schema are not a subset of the class of regular languages. Hence, a single FSA can only be used for processing small fragments of the XML tree. In general, it can only handle the direct children of a node. SEGOUFIN and VIANU elaborate on this problem in detail [18]. In consequence, BiM and Exalt need to provide additional mechanisms for controlling which of the generated automata is used at a certain stage of the compression process. Xebu uses a single FSA for improving compression results, but here the automaton is only an add-on for the compression algorithm and does not fully represent the Relax NG grammar.

Well-known validating XML parsers like Xerces or Expat are also based on automata. Like BiM and Exalt they cannot only rely on FSAs but need to employ additional mechanisms like tree automata [19].

A. Architecture

Our approach is to generate a single deterministic pushdown automaton (PDA) which represents the XML Schema

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="a" type="A"/>
 <xsd:complexType name="A">
  <xsd:choice>
   <xsd:element name="b" minOccurs="0">
    <xsd:complexType>
     <xsd:choice>
      <xsd:element ref="a" minOccurs="2" maxOccurs="2"/>
      <xsd:element name="c" type="xsd:int"/>
     </xsd:choice>
    </xsd:complexType>
   </xsd:element>
  </xsd:choice>
 </xsd:complexType>
</xsd:schema>
```

Fig. 1. Recursive XML Schema Description

information that is included in the commonly available WSDL description of a web service. This is done at the sender as well as the receiver side. The sender processes the SOAP message using the PDA. The path taken through the automaton represents the structure of the SOAP message (markup) and is encoded together with the leaf element (simple type) values. This information is then sent to the receiver, which uses its automaton to reconstruct the original document.

SEGOUFIN and VIANU show in [18] that it is possible to construct deterministic pushdown automata from an XML grammar yielding improved speed of parsing. However, their work considers only the by now outdated DTD grammars. We adopted the concept of using PDAs for XML processing and made it applicable to XML Schema.

In the following we present an algorithm for constructing a single PDA that fully represents an XML Schema. Due to the nature of this approach the PDA is a highly efficient parser but – as we will show – it can also be used for data compression.

Our algorithm takes an arbitrary XML Schema document as its input which describes the structure of the documents to be transmitted. Figure 1 shows an example that we will use for demonstrating our approach.

Of course, for SOAP applications this Schema document is typically much more complex and also imports other XML Schema documents for processing different namespaces.

The first construction step is to convert the XML Schema document into a so called Regular Tree Grammar (RTG) G. An RTG is a 4-tupel consisting of a set of non-terminal symbols N, a set of terminal symbols T, a set of production rules P and a set of start symbols $S \subseteq N$. MURATA ET AL. provide in [19] a detailed description of how to convert the XML Schema description into a RTG.

For our example the conversion result looks as follows:

$$G = (\{A, B, C, \text{xsd:int}\}, \{a, b, c\}, P, \{A\}) \text{ with } P = \{$$
$$\begin{array}{lcl} A & \rightarrow & a\ (B + \varepsilon), \\ B & \rightarrow & b\ (AA + C), \\ C & \rightarrow & c\ (\text{xsd:int}) \end{array}$$
$$\}$$

All used simple and complex data types result in non-terminal symbols (written in capital letters or prefixed with `xsd:`). All possible element names result in terminal symbols (small letters). The set of start symbols contains all non-terminals belonging to elements that are declared on top-level of the XML Schema document. Please follow this mapping by means of figure 1.

The generated production rules reflect the structure of valid documents. The right hand side of such a rule consists of an element name followed by a regular expression which describes the content model of this element, i.e. types of child elements it may have. Each regular expression maps to non-terminals, which occur as left hand side symbols of other rules. The rules in this example have the following meaning: an element a contains an element b or (+) nothing (ϵ), whereas b contains either two a elements or a c element, which contains an integer value. Hence, G is equivalent to the Schema in figure 1.

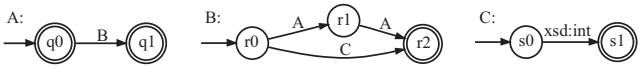


Fig. 2. Set of finite state automata generated from the regular tree grammar.

These regular expressions are then converted into a set of FSAs (see figure 2). A detailed description of the underlying conversion algorithm is given in [20]. For each element of the grammar the according FSA describes which direct child elements may follow in what sequence.

Finally, we construct the PDA. It accepts an input word by emptying the stack, i.e. initially there is a special symbol Z on the stack. If this is popped and not pushed again immediately, the PDA terminates. For each complex type two states are created, an opening one and a closing one. Additionally, one start state is created and one state for each simple type (like `xsd:int`, `xsd:string` etc.).

Then, the transitions are created. Each of them is labeled with a 3-tupel ($read$,pop,$push$) where $read$ is the tag to be read from the input and pop is the top stack symbol. Both are required for the transition to fire. The $push$ value, which can consist of zero, one or more symbols, is then written onto the stack. Note that, if multiple values are written, they are specified in inverse order.

From the start state there are transitions to all opening states that belong to an $n_i \in S$. They are labeled with the opening tag that belongs to the destination state, the stack start symbol Z and the start state of the FSA belonging to the destination state type as an additional $push$ value. Correspondingly, a transition from all closing states which belong to $n_i \in S$ to themselves is added. It reads an empty string ($\#$) from the input and pops Z from the stack, which terminates the PDA. All other transitions are created following the algorithm shown in figure 3.

The resulting PDA is depicted in figure 4. The basic idea behind this type of automaton is that the stack operations emulate the execution of the FSAs. That way the automaton checks for each element in the document if it has the correct number and sequence of child elements.

At this point, the automaton is ready for parsing. Finally, we add the core feature that is used for data compression: For all states with more than one outgoing transition we tag the

```
for all fsa ∈ FSAs do
  for all state ∈ fsa.states do

    for all t_in ∈ state.transitions_incoming do
      for all t_out ∈ state.transitions_outgoing do
        label = (
          t_out.type.tag_open,
          state,
          (FSAs.getFSA(t_out.type).startState, t_out.destState));
        createPDATransition(
          PDAStates_close.get(t_in.type),
          PDAStates_open.get(t_out.type),
          label);
      end for
    end for

    if state ∈ fsa.finalStates then
      for all t ∈ state.transitions_incoming do
        label = (fsa.type.tag_close, state, null);
        createPDATransition(
          PDAStates_close.get(t.type),
          PDAStates_close.get(fsa.type),
          label);
      end for
    end if

    if state = fsa.startState then
      for all t ∈ state.outgoingTransitions do
        label = (
          t.type.tag_open,
          state,
          (FSAs.getFSA(t.type).startState, t.destState));
        createPDATransition(
          PDAStates_open.get(fsa.type),
          PDAStates_open.get(t.type),
          label);
      end for
    end if

    if state ∈ ({fsa.startState} ⋂ fsa.finalStates) then:
      label = (fsa.type.tag_close, state, null);
      createPDATransition(
        PDAStates_open.get(fsa.type),
        PDAStates_close.get(fsa.type),
        label);
    end if

  end for
end for
```

Fig. 3. Constructing the PDA transitions from a set of FSAs.

transitions with unique binary identifiers (depicted as circled values in figure 4).

The core idea of our compression scheme is to represent a document as a sequence of traversed transitions on the path through the PDA. Of course, the code words used for tagging the transitions must be decodeable in an unambiguous manner. An advantageous scheme for generating suitable codes is the Huffman algorithm [21]. Here, for each state a Huffman table is created which maps code words to outgoing transitions. Currently, we do not infer any heuristic about transition traversal probabilities from the Schema but apply equal probabilities for each transition instead.

Besides the path through the PDA, which represents the markup of the encoded document, we must encode the simple type values. For all possible simple types our compressor provides functionalities for encoding, decoding and validating.

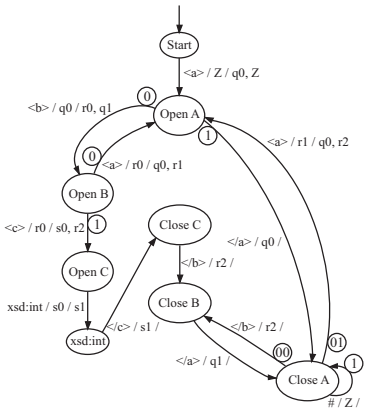
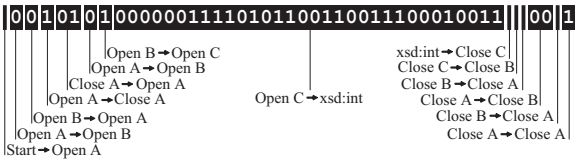


Fig. 4. Pushdown automaton constructed from the XML grammar.

For each simple type we use an appropriate binary encoding, e.g. a 32-bit value for `xsd:int`. If a transition labeled with a simple type is executed, the binary encoded value is directly placed at the current position into the output bit stream. This makes this data format ideal for XML streaming, since no data must be buffered during compression or decompression. This is a major advantage over container-based approaches like XMill.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
  <a>
    <b>
      <a>
      </a>
      <a>
        <b>
          <c>64382739</c>
        </b>
      </a>
    </b>
  </a>
```

(a) Text encoding using 119 bytes.



(b) Binary encoding using 42 bits ≈ 6 bytes.

Fig. 5. Example XML document before and after compression.

Figure 5 shows an example instance document and its encoding generated with the automaton in figure 4. The markup is encoded as the path through the PDA (using zero bits for unambiguous transitions) while simple types are inline encoded using compact binary representations.

The decompression process works vice versa: The receiver uses the bit stream generated during compression to control the path through its instance of the PDA. For each traversed transition it writes the tag parts of the according labels into the output document. For transitions labeled with simple types it reads either a fixed number of bits from the bit stream (e.g.

32 bits for `xsd:int`) or reads a number of bytes until it encounters a stop byte sequence (this is done for all character encoded datatypes like `xsd:string`).

For the sake of simplicity and compactness we did not show the handling of namespaces and attributes in our example. Anyhow, both features are supported. Namespace information is fully acquired from the XML Schema document and incorporated into the PDA, i.e. each transition label carries information about the namespace of the tag to be processed. Since we do not encode the namespace prefixes of the input document this information is lost during compression. The decompression step comprises the generation of new generic namespace prefixes. Attributes are handled just like elements but marked with a special flag avoiding naming ambiguities. For example `<a b="c">d</a>` is processed like $\texttt{<a><b}_{\texttt{att}}\texttt{>c</b}_{\texttt{att}}\texttt{>d}$.

With our approach we can address the main requirements formulated by the W3C. The binary representation can be written (or read) directly by using an API like SAX. In such a case an application interacts with the automaton instead of working with the XML text representation. In any case the generated encoding is independent of transport mechanisms and can be processed very efficiently. An evaluation of the achieved compactness is conducted in the following section.

B. Performance Evaluation

We evaluated the compression performance of our compressor, which we call "Xenia", using the two benchmark web services described in section II-C.

We evaluated Xenia with three different settings for encoding `xsd:string` data. First, we used the "none" coding style, which means that the UTF-8 representation of the string is written byte by byte into the output stream. A unique byte sequence is used for indicating the end of a string. This in particular means that string data is not compressed here. The second variant works like the first one but additionally uses the adaptive Huffman algorithm [22] for compressing the byte representation of the string. The third setting uses the Prediction by Partitial Match (PPM) algorithm which is one of the best approaches available for compressing string data. Since single strings are typically rather short and the PPM algorithm needs at least a few hundred bytes for reasonable compression results we grouped all `xsd:string` values occurring in the input document into a so called string container. In the output stream we do not encode the string data itself but an integer value which denotes the position of the particular string in the string container. After the processing of the input document has ended the container is compressed and the resulting byte sequence is appended at the end of the output stream. This technique has been introduced by LIEFKE in [4]. Note that placing the compressed container as a whole into the output stream prevents direct stream processing because the container must be decompressed prior to accessing string data. However, this is just an implementation issue. As shown in [5] it is possible to avoid this problem by splitting up the PPM output and multiplexing it with the markup stream which preserves the streaming option.

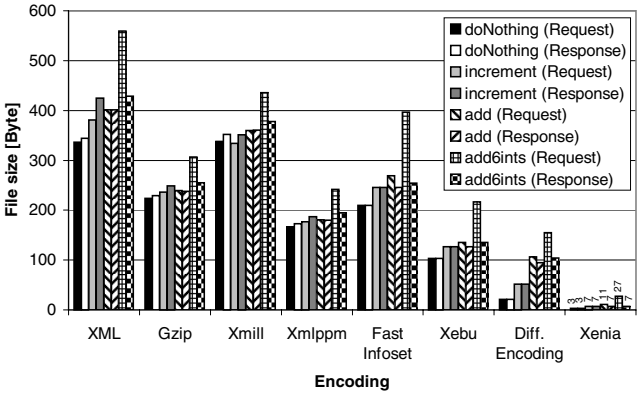


Fig. 6. Compression results for the calculator web service.

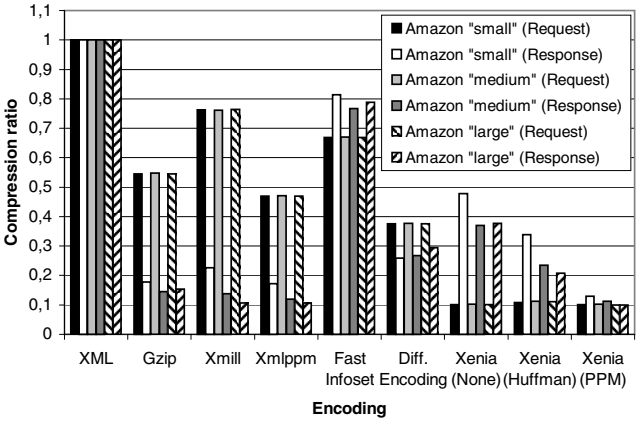


Fig. 7. Compression results for the Amazon web service.

Figure 6 shows the resulting file sizes for the different types of request and response messages exchanged by the calculator web service. It is clearly visible that Xenia performs best by far. Since the calculator web service does not take any `xsd:string` values as input, all three string encoding variants of Xenia are producing the same compression results. Hence, only one set of results is shown in this figure.

Figure 7 depicts the results for the Amazon web service. Since the file sizes of the uncompressed documents vary in a wide range from 680 to 229,619 bytes, the depiction of the file sizes would render small byte values nearly invisible. Hence, we depicted the compression ratio λ instead.

All three variants of Xenia compress the requests to about 10% of their original sizes. This is because they contain a number of short strings that generally can hardly be compressed. As a result, it does not make any difference which of the three Xenia string encoding variants is chosen. For messages that contain a lot of string data, like the three responses, it is clearly visible that the three variants perform quite differently. The "none" variant yields compression ratios of 0.48, 0.37 and 0.38 for the small, medium and the large

response respectively. Note that this implies that the markup is compressed to nearly zero since the amount of string data remains unchanged (c.f. section II-C: the fraction of string content in the original documents is 46.9%, 35.4% and 36.9%). When Huffman coding or PPM are used, the compression ratio for the large response improves to roughly 0.2 and 0.1 respectively.

IV. Conclusion and Future Work

Due to the talkative nature of XML recent research and standardization efforts concentrated on more compact binary XML representations. The main contribution of this paper is a Schema-aware binary XML encoding algorithm. It complies to the requirements that the W3C Binary XML Working Group agreed on to be most important.

We showed how to construct a pushdown automaton (PDA) from a publicly available XML Schema document (typically included in a WSDL description). This type of PDA can not only be used for parsing and validating but also enables a new compression approach. The path through the automaton during parsing fully represents the document structure and can be encoded extremely compact. Because type information is incorporated into the PDA during construction, simple type values can also be encoded using optimized binary representations. For string data we presented three different encoding variants ranging from a very simple uncompressed inline encoding to sophisticated PPM compression.

The performance evaluation of our implementation called Xenia showed that markup is compressed to nearly zero. Together with the PPM string encoding algorithm Xenia can even compress SOAP data with a high amount of character data very effectively. All XML files regardless of their size and structure were compressed to sizes between one and fifteen percent of the original.

Although our implementation does currently not fully support all features of the XML Schema language it is powerful enough to process most SOAP documents. In particular, constraint checking for the ID/IDREF feature has not been implemented, yet.

Therefore, we are currently working on the completion of our Java implementation. Because PDAs can be implemented even in resource constrained environments, SOAP processing on embedded devices is a natural application domain of Xenia. Hence, we are working on a code generation module for Xenia that produces C code, which embodies a Schema specific XML compressor. This idea can be extended to application specific XML compression hardware: To reach this goal future work might include code generation for hardware description languages like VHDL or Verilog.

Web services technology started out to address the problems resulting from heterogeneity in distributed environments. However, it has not yet found its way into the field of ubiquitous computing, which is considered to be a major development in information technology. From our point of view a main reason for this is that processing, storage and transmission of XML are very resource demanding. Hence, the advent of efficient techniques for dealing with XML is a prerequisite for bridging the gap between web services and ubiquitous information technology.

References

[1] C. Werner, C. Buschmann, and S. Fischer, "WSDL-Driven SOAP Compression," *International Journal of Web Services Research*, vol. 2, no. 1, 2005.

[2] W3C, "Working group note: XML binary characterization," Mar. 05. [Online]. Available: http://www.w3.org/TR/xbc-characterization/

[3] ——, "Charter of the efficient XML interchange working group," Nov. 2005. [Online]. Available: http://www.w3.org/2005/09/exi-charter-final.html

[4] H. Liefke and D. Suciu, "XMill: an efficient compressor for XML data," in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, Dallas, Texas, USA, 2000, pp. 153–164.

[5] J. Cheney, "Compressing XML with multiplexed hierarchical PPM models," in *Data Compression Conference*, 2001, pp. 163–173. [Online]. Available: http://citeseer.ist.psu.edu/cheney01compressing.html

[6] P. Sandoz, A. Triglia, and S. Pericas-Geertsen, "Fast infoset," jun 2004. [Online]. Available: http://java.sun.com/developer/technicalArticles/xml/fastinfoset/

[7] V. Toman, "Syntactical compression of XML data," in *Proceedings of the International Conference on Advanced Information Systems Engineering*, Riga, Latvia, June 2004.

[8] [Online]. Available: http://wbxmllib.sourceforge.net/

[9] M. Girardot and N. Sundaresan, "Millau: An encoding format for effcient representation and exchange of XML over the web," in *9th International World Wide Web Conference*, Amsterdam, Netherlands, May 2000, pp. 747–765.

[10] U. Niedermeier, J. Heuer, A. Hutter, W. Stechele, and A. Kaup, "An MPEG-7 tool for compression and streaming of XML data," in *Proceedings of the IEEE International Conference on Multimedia and Expo*, Lusanne, Switzerland, Aug. 2002, pp. 521–524.

[11] P. Tolani and J. R. Haritsa, "XGRIND: A query-friendly XML compressor," in *Proceedings of the International Conference on Data Engineering*, San Jose, California, USA, Feb. 2002, pp. 225–234.

[12] J. Kangasharju, S. Tarkoma, and T. Lindholm, "Xebu: A binary format with schema-based optimizations for xml data." in *Proceedings of the International Conference on Web Information Systems Engeneering*, New York City, New York, USA, Nov. 2005, pp. 528–535.

[13] J. Clark and M. Makoto, "Definitive specification for RELAX NG using the XML syntax," Dec. 2001. [Online]. Available: http://www.relaxng.org/spec-20011203.html

[14] Intelligent Compression Technologies, Inc., "XML Xpress." [Online]. Available: http://www.ictcompress.com/products_xmlxpress.html

[15] "Amazon E-Commerce Service." [Online]. Available: http://webservices.amazon.com/AWSECommerceService/AWSECommerceService.wsdl

[16] "Sun Relax NG converter." [Online]. Available: http://www.sun.com/software/xml/developers/relaxngconverter/

[17] "Trang: Multi-format schema converter based on RELAX NG." [Online]. Available: http://www.thaiopensource.com/relaxng/trang.html

[18] L. Segoufin and V. Vianu, "Validating streaming xml documents," in *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. New York, NY, USA: ACM Press, 2002, pp. 53–64.

[19] M. Murata, D. Lee, and M. Mani, "Taxonomy of XML schema languages using formal language theory," in *Proceedings of Extreme Markup Languages*, Montreal, Canada, Aug. 2001.

[20] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1988.

[21] D. A. Huffman, "A method for the construction of minimum-redundancy codes," *Proceedings of the Institute of Radio Engineers*, vol. 40, no. 9, pp. 1098–1101, Sept. 1952.

[22] K. Sayood, *Introduction to data compression (2nd ed.)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000.