A Context-Dependent XML Compression Approach to Enable Business Applications on Mobile Devices

Yuri Natchetoi[1], Huaigu Wu[1], and Gilbert Babin[2*]

[1] SAP Labs
Montréal, Québec, Canada
Yuri.Natchetoi@sap.com
Huaigu.Wu@sap.com
[2] Information Technologies, HEC Montréal
Montréal, Québec, Canada
Gilbert.Babin@hec.ca

Abstract. As the number of mobile device users increases, the need for mobile business applications development increases as well. However, such development is impeded by the limited resources available on typical mobile phones. This paper presents a context-dependent XML compression approach that enables the deployment of business applications on mobile devices. That is, the compressed XML document is not self-contained and cannot be de-compressed without using information shared between the sender and the recipient. By relying on shared information, we obtain a better compression ratio than existing context-free compression algorithms.

1 Introduction

Nowadays, mobile devices, especially cell phones are everywhere. According to a recent study [1], there are almost 2.5 billions of connected mobile devices in the world. Furthermore, the capabilities of the available phones are also increasing, making it possible to envision complex mobile applications. However, business-related applications running on cell phones are still rare. The major obstacles are the following: 1) limited data storage capability, 2) limited network access capability, 3) limited computation capability, and 4) limited display capability.

Most business applications require the client side to process large amounts of data either locally or through high-speed networks. Currently, most existing cell phones cannot fulfill these requirements. Typical solutions for large data issues in the world of desktop applications are compression and caching. Data are compressed in order to achieve fast transmission, and then de-compressed for access or caching. However, for mobile devices, the compressed data might still be too large for efficient transmission. More importantly, mobile devices lack an efficient way to de-compress the compressed data, and process or store

* On sabbatical leave at SAP Labs, Montréal

the de-compressed data. Hence, business application development for mobile devices requires an innovative approach to compress and de-compress data. In this paper, we propose to use context-dependent compression on business data. That is, the compressed data does not need to contain a lot of additional data for the de-compressor to interpret. De-compression relies on the presence of shared information between the sender and the recipient of the data. This approach was developed at SAP labs to enable building business applications on regular J2ME-enabled cell phones [2, 3]. Data objects are transformed into XML format to be compressed on the server side, and clients de-compress the compressed XML file to restore data objects. The main contributions of our approach are that the context-dependent compression provides a compression rate for business objects exchange higher than existing compression algorithms, and the de-compression algorithm is very simple, which can easily adapt to the capabilities of most cell phones.

XML is frequently used to serialize objects and exchange them through the network. Also, many tools are currently available to compress XML to facilitate transferring and querying. These tools can be categorized as follows:

1. *General compression tools* like Winzip[4], Gzip[5], etc., which compress XML files as regular text files without considering the XML structure,
2. *XML compression tools* such as XMill [6], XMLPPM [7], XAUST [8], which provide better compression ratios than general tools by using the XML structure to optimize the compression results, and
3. *Query-able XML compression tools* including XPress [9], TREECHOP [10], and XGrind [11], which adopt a homomorphic transformation strategy [12] for compressing the XML structure and supporting queries without de-compression at the cost of compression efficiency reduction.

All the aforementioned compression approaches are context-free. The compressed XML document contains all the information required to perform the de-compression, and every transformation is therefore independent. However, in many business applications, object structures are usually predefined, shared on both server and client sides, and rarely changed during run-time. Based on this shared knowledge, the de-compressor can interpret a compressed file even when the XML structure information is not contained by the compressed file. Furthermore, the coding schema for the compression could be optimized to get better compression ratios, because the de-compressor already has some knowledge. The proposed solution combines coding XML file by multiple coding schemata and de-coupling transmission of XML structure information from data transmission. By using multiple coding schemata, we are able to reduce the number of bits required to represent the different symbols transmitted. De-coupling of structure transmission and data transmission drastically reduces the bits transferred, especially for short messages.

This paper is organized as follows. We describe the principles supporting XML compression in Section 2. In Section 3, we describe the proposed compression mechanism. Results from a comparative study with other compression approaches are presented in Section 4. We conclude in Section 5.

```xml
<PurchaseOrder no="1456">
  <Date>06/05/05</Date>
  <CustomerID>765345</CustomerID>
  <Order>
    <Item>
      <ProductNo>P-4534</ProductNo>
      <Quantity>2</Quantity>
    </Item>
    <Item>
      <ProductNo>P-9182</ProductNo>
      <Quantity>1</Quantity>
    </Item>
  </Order>
</PurchaseOrder>
```

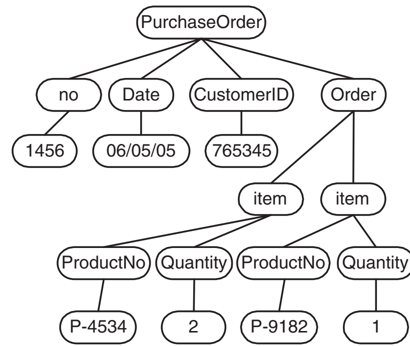

Fig. 1. XML Encoding of a Simple Object **Fig. 2.** Parse Tree for the XML Encoding

2 Principles Supporting XML Compression

Kropf *et al.* [13] have claimed that the information to be transmitted between two systems may be reduced when considering the knowledge (semantic, syntactic, etc.) shared by both systems. Consequently, only the information that is not shared needs to be exchanged, which leads to context-dependent compression.

2.1 Object-Oriented XML

Our compression approach is mostly used to exchange application data in XML format between server and client; messages are relatively small and usually contain a single data object or a collection of data objects. We use a simple algorithme to transform objects to XML files (many tools exist to perform this task, see SOX [14] and XStream [15]). First, the name of the object class becomes the root element of the XML file. Each simple attribute (number or text) of the object is transformed into an attribute of the root element. Each complex attribute (subordinate object) is transformed to a child element. For collection objects (vectors, hash tables, etc.), each contained object is transformed to a child of the wrapping element representing the collection object. This process continues recursively until the whole object has been transformed (see Fig. 1).

2.2 Structured Compression

All general-purpose compression algorithms, such as gzip [5], do not take into account knowledge of internal file structure, business-specific information or data types. They assume that the file to be compressed is composed of plain text with symbol (ASCII characters or their combinations) occurrence frequencies unknown in advance. These algorithms collect their statistics based on a limited data set, build an ad-hoc encoding dictionary, and attach this dictionary to the compressed file. These algorithms assume that every file/message sent is unique

(in terms of frequency distribution), is unstructured, and that all parts of the message are equally important. Because of these assumptions, most general-purpose compression algorithms do not work well on short messages and do not achieve the highest possible compression ratios.

If we consider compression in terms of information entropy [16], we can then apply the additional knowledge of business objects structure to reduce the entropy of the message, making it possible to provide a better compression mechanism that yields a better compression ratio. This is the case for specialized business applications; we have additional information about business objects exchanged between the server and the client. For instance, since most business applications use XML as a universal data exchange format, we can use some knowledge of the file structure such as DTD or XML schema. The DTD provides us with all possible states we may encounter when parsing the XML file: tag names, attribute names, and values inside every attribute and inside every tag. The DTD also limits the type of information that may be encountered in each of these states. Consequently, we can determine the probability of meeting certain information in certain states, and can use this knowledge to reduce message entropy. For example, we know that inside a lexical unit delimited by symbols '<' and '>' there is a "very high" probability to encounter one of the tag names listed in the DTD. In fact, most XML compression algorithms use this approach (XMill [6], XGrind [11], Xpress [9]). To further reduce entropy, we build multiple encoding dictionaries, one for each state. This way, we can use longer lexical items (sequences of characters) and their occurrence frequency distribution. Every state has its own set of lexical items and therefore its own dictionary. If more than one state have the same data domain, we can reuse the dictionary. The decision to reuse a dictionary depends on the expected gain in compression.

The advantage of using state-specific dictionaries is that each dictionary is then relatively small, and the resulting encoding requires fewer bits per symbol. Indeed, in many situations, the number of available lexical items per state is finite. We can often list all possible values for these states. In other situations where some items do not exist in a dictionary, we can list the most frequent items, collected by statistical processing. In this latter case, we can introduce "escape" symbols, which are used to inform the parser that the following sequence of symbols is not a part of the dictionary, but a combination of lower-level symbols, like, for example decimal numbers or ASCII characters.

2.3 Combining XML parser and compressor

The XML parser implements a finite state machine that navigates the XML document tree (Fig. 2), as determined by an XML schema or DTD. This approach is somewhat similar to XAUST [8], where the compressor uses a recursive finite state machine, one machine per syntactic element. In the approach we propose, we replace the recursive finite state machines by multiple dictionaries with a single finite state machine. Every state corresponds to a specific syntactic unit (XML tag, attribute, value) where symbol combinations have different

Tree Path (key)	*Dictionary elements (bits: value)*
/	1 : PurchaseOrder
/PurchaseOrder	00: Date 01: CustomerID 10: Order 111: no 1101: ; 1100: \
/PurchaseOrder/Date	1: 06/05/05 0: \
/PurchaseOrder/CustomerID	1: 765345 0: \
/PurchaseOrder/Order/	1: item 0: \
/PurchaseOrder/Order/Item/	0: Quantity 11: ProductNo 101: ; 100: \
/PurchaseOrder/Order/Item/ProductNo	0: P-4534 11: P-9182 10: \
/PurchaseOrder/Order/Item/Quantity	0: 1 11: 1 10: \
/PurchaseOrder/no	1: 1456 0: \

Fig. 3. DTD-Specific Dictionaries of the PurchaseOrder Object

occurrence probabilities. The transitions correspond to the passage from one syntactic unit to the next. The machine can be presented as a graph with nodes corresponding to states and arcs corresponding to transitions. The graph can be automatically created from the DTD or from any formal syntax description such as BNF. Formally, the parser is defined as $F = (A, V, T)$, where A is the set of symbols read by the parser, V are vertices or states and T are transitions $(T : V \times A \rightarrow V)$.

Every state $(v_i \in V)$ has its own alphabet of lexical items (i.e., its own dictionary A_i). In fact, the dictionaries are built according to the parse tree following the DTD syntax. Lexical items can be single characters, words, tag names or possible choice options in the data field. Every lexical item $(a \in A)$ has a probability of appearance in the current state $P(a|v_i)$. This probability is determined by collecting statistics on a set of messages, which provides better results than using a single file, as many implementations of gzip and similar compression utilities do. In XML, symbols like: '<', '>', '"', and '&' play the role of delimiters which force a transition in the state machine. To enable the transition from one state to the next, the state-specific alphabet contains not only the set of possible lexical items but also the set of transition delimiters. Formally, $A_i = \{L_i\} \cup \{E_i\} \subset A$, where L_i are possible lexical items for the state v_i and E_i are symbols marking possible transitions leaving state v_i. These state transition delimiters also occur with a certain probability that can be calculated using statistical analysis. Then, every symbol from A_i can be encoded. Figure 3 shows the dictionaries generated based on the tree of Figure 2, where ';' is the end-of-state symbol and '\' is an escape symbol.

The compression module replaces every lexical item in the business object with a bit sequence. Figure 4 shows the resulting bit sequence to encode the XML stream of Figure 1. The length of the sequence depends on the encoding algorithm used (e.g., Huffman encoding [17, 18], arithmetic compression [19]). At the end of a state, the encoder adds an end-of-state symbol (';'), which is also encoded in the same dictionary. If it can be determined from the data schema that there is only one lexical item in the state and that the next state

```
1 111 1 00 1 01 1 10 1 11 0 0 11 101 1 11 11 0 0 101 1101
```

Fig. 4. Compressed XML Stream for the PurchaseOrder Object

is predetermined, then the end-of-state symbol may be omitted. For example, XML attribute encoding presumes that each attribute name is always followed by an attribute value, and hence the transition system between the attribute name and its value can be omitted.

2.4 De-Coupling Dictionaries From Business Messages

One characteristic of business applications is that many messages of the same type will have the same frequency distribution with high probability and therefore can share the same dictionary. This assumption allows the de-coupling of the dictionary from the message. De-coupling helps both with compression ratio and performance as dictionaries can be sent once for many messages, hence making messages shorter. Context-free compression techniques cannot make this assumption, and therefore cannot gain from past messages. Dictionary construction is the most time consuming part of the compression. By using a single dictionary for multiple messages, dictionary construction does not need to be performed for every message. Furthermore, it can be done on the server where the CPU is much more powerful that on the (mobile) client. Once compression dictionaries are compiled, they can be deployed to the client and be ready for a fast streamline compression and de-compression. It is also possible to collect frequency statistics during business operations and periodically optimize and re-compile dictionaries using new statistical data.

When comparing our compression approach to others, we must consider the fact that the dictionary is not sent with every message, but once in a while. Consequently, we must determine the average number of messages that are sent between dictionary updates. This will allow us to compute the *adjusted compression rate* (C_a) as:

$$C_a = \frac{N + pD}{M},$$

where N is the number of bytes sent in the compressed message, D is the number of bytes of the dictionary, p is the probability that the dictionary is sent with the message, and M is the number of bytes of the uncompressed message. Furthermore, if changes in the dictionary are not too significant, i.e. one additional option has been added to the possible attribute selections, then we can send differential updates for the dictionary, including only updated symbols. This is possible since dictionaries are only dependent on states. A change in one dictionary will not change others. Differential updates for business objects will be described in another paper. Every time we change the dictionary, we update its version. When the mobile client initializes the session with the server, it includes the version of local dictionaries into the session handshake. If the version is out of date, the server sends differential dictionary updates first.

3 Implementation

The compression algorithm consists of a compressor and a de-compressor. The compressor is implemented using J2SE and is running on the server. The de-compressor is implemented using J2ME and is running on mobile phones. The compressor's operations can be broken up in two stages. During the first stage, it collects frequency of occurrence statistics and builds a set of dictionaries. This is done by analyzing a sample set of XML files used in the business process. The compressor first determines the frequency of occurrence of all XML elements, attributes, and tag values by state, where every tag, attribute, and value constitutes a separate state. The alphabet for a specific state consists of all the lexical items allowable by the XML schema (L_i) plus two transition symbols (E_i): a generic end-of-state symbol (';') and an escape symbol ('\') for adding new values to the state without rebuilding the whole set of dictionaries. For a given state, we can choose a single generic dictionary to encode symbols not present in the state-specific dictionary, which requires a single escape character. These symbols are also considered in collecting the frequency distribution statistics.

Once the alphabet for each state (A_i) is defined and the frequency of each symbol is determined, the compressor builds a Huffman binary tree [17, 18] for each state. These trees are in turn compressed using generic dictionaries also built by using the Huffman compression algorithm.

During the second stage of the compression, actual messages are compressed for transmission to the mobile device. This process requires the compressed dictionaries to be already deployed on mobile phones. The compressed message is wrapped into a standard XML header that has some attributes to identify the XML schema, the compression schema, the encoding, the encryption method, and some other parameters. The attributes can be easily extended because of the flexibility of XML. Clearly, the wrapping schema can also be compressed using the proposed approach, the compression dictionary required being pre-deployed with the compressor.

When de-compressing, the parser is combined with the de-compressor and therefore it has knowledge of the current state. The transitions are determined from the XML schema used. The de-compression algorithm is very simple and allows to uncompress and to de-serialize an XML stream into a business object in a single run, bypassing the conversion phase from binary format to text format. The general algorithm is illustrated as follows:

```
decompress(state,input)
    tree := findDictionaryTree(state)
    do
        token := getToken(tree,input)
        if (token <> end-of-state)
            nextState := getNextState(state,token)
            decompress(nextState,input)
        end-if
    while (token <> end-of-state)
end
```

The de-compression starts at the root element of the XML document (the initial state). Function `findDictionaryTree()` can readily identify the dictionary for this element as dictionaries are organized in a tree structure that follows the DTD syntax. As we move from state to state, we are moving from one dictionary to another. The de-compressor then reads bits from the input stream and immediately converts them into lexical items using the Huffman binary three for the current state (`getToken()`). This is accomplished by reading one byte from the stream and then traversing the Huffman tree one node at a time by shifting single bits. If we reach the end of the current byte, we retrieve the next byte from the input stream. When a leaf node in the Huffman tree is reached, we determine the corresponding lexical item and return it to the parser. The parser changes its state depending on the lexical item identified (`getNextState()`). This method simply implements the transition table T for the current state. We then call the `decompress()` method recursively from this new state. This process is repeated sequentially until the end-of-state symbol for the root element is reached.

This algorithm only requires 1) one encoded tree that reflects the structure of the XML file (based on the DTD or XML schema) and 2) a number of small binary trees that determine Huffman codes for every possible lexical item in each state. The transition from one node to another in the Huffman code tree is similar to the transition in the XML structure and therefore the decoding module is very compact and efficient. It is suitable for the computation capability of mobile phones.

The encoding algorithm is a little bit more complex; however in many business cases the amount of information transferred from server to the mobile device and from the device to the server is highly asymmetric. In this case, instead of encoding the whole XML file, the client can only send back the values modified by the user.

4 Comparison with Other Algorithms

In order to assess the quality of the proposed context-dependent compression algorithm, we compared it with a number of existing context-free compression algorithms, namely: bzip2 [20], gzip, and XMill. We executed these compression algorithms on a number of test cases on a Pentium 4 machine (CPU 3.4 GHz and 2 GB RAM) running Windows XP. The test cases included 7 typical messages used to exchange data between SAP mobile business applications and the SAP mobile infrastructure middle-ware. Each message contains a list objects; the smallest message contains 20 business objects ($3,742$ bytes) and the biggest message contains 436 business objects ($100,029$ bytes). For each test case, we measured the adjusted compression ratio (C_a). Better compression mechanisms will show smaller compression ratios. The results are presented in Table 1, which shows message sizes for the smallest (m_{min}) and largest (m_{max}) messages, and the average (μ) and standard deviation (σ) of C_a over all the test cases.

When comparing gzip and XMill, we see that gzip performed better for smaller messages, but worse for bigger messages. These results are consistent

Table 1. Comparison of compression algorithms; m_{min} is the smallest message, m_{max} is the largest message. μ and σ are the average and standard deviation (σ) of the adjusted compression ratio C_a, p is the probability of sending the dictionary with the message

	p	*Size of* m_{min} *(Bytes)*	*Size of* m_{max} *(Bytes)*	μ	σ
Original file		3742	100029	1	0
Context-dependent compression	0.00	127	4108	0.0422	0.0088
	0.10	535	4516	0.0732	0.0333
	0.33	1489	5470	0.1457	0.1260
	0.50	2169	6150	0.1974	0.1923
bzip2		772	8417	0.1520	0.0725
gzip		648	10864	0.1603	0.0617
Xmill		919	9189	0.1690	0.0903

with results from [21]. Table 1 shows that gzip has a better average compression rate than XMill since our test cases tend to be small, which is typical for object exchange in mobile applications. The results also support our claim that context-dependent compression performs better than other algorithms. When all the dictionaries are sent once every 10 messages ($p = 0.1$), the average compression rate is two times better than other compression algorithms. Only when the dictionaries are sent once every 3 messages ($p = 0.33$) is the compression rate close to the other algorithms. Considering the nature of business applications where multiple messages of the same nature are sent between client and server in a short time interval, p is much closer to 0 than to 0.33.

5 Conclusion

The context-dependent compression method provides a high compression ratio and a simple de-compression algorithm that works reasonably on very simple mobile devices. This compression approach performs very well for mobile business applications where an intensive exchange between server and client takes place, using short XML messages, and when the structure of the XML documents is known in advance. This method will probably not prove as efficient for unstructured files or even for XML files having diverse schemata and frequency distributions.

One of the important advantages of this method is a simple de-compression algorithm combined with a simple XML parser. This approach better suits mobile devices, which neither have powerful CPUs to de-compress data nor enough memory to keep all the files uncompressed. Our proposed method requires more resources on the compressing side and a more complex infrastructure for dictionary management, however these requirements are justified for an enterprise environment where the servers normally have high CPU resources. In addition such an environment typically has a business process work-flow that is already in place which makes it simple to manage the synchronization of dictionaries.

Overall, there is an increasing demand for the efficient compression methods for mobile client-server applications, which is growing rapidly with the growth of the mobile application market. The context-dependent compression method described in this paper provides good results for this niche and potentially can be used in many applications.

References

1. Taylor, C.: Global mobile phone connections hit 2.5bn (September 2006) http://www.electricnews.net/frontpage/news-9792607.html.
2. Wu, H., Natchetoi, Y.: Mobile shopping assistant: Integration of mobile applications and web services. In: WWW 2007, Banff, Alberta, Canada (May 2007) 1259–1260
3. Dagtas, S., Natchetoi, Y., Wu, H., Shapiro, A.: An integrated wireless sensing and mobile processing architecture for assisted living and healthcare applications. In: HealthNet 2007, Puerto Rico, USA (June 2007)
4. : Winzip (last visit in Jan. 2007) http://www.winzip.com/.
5. Gailly, J., Adler, M.: gzip 1.2.4. (last visit in Jan. 2007) http://www.gzip.org/.
6. Liefke, H., Suciu, D.: XMill: An efficient compressor for XML data. In: Proc. of the ACM SIGMOD Int'l Conf. on Management of Data. (2000) 153–164
7. Cheney, J.: Compressing XML with multiplexed hierarchical PPM models. In: Proc. of the IEEE Data Compression Conf. (2000) 163–172
8. Hariharan, S., Shankar, P.: Evaluating the role of context in syntax directed compression of XML documents. In: IEEE Data Compression Conf. (2006) 453
9. Min, J.K., Park, M.J., Chung, C.W.: XPRESS: A queriable compression for XML data. In: Proc. of the ACM SIGMOD Int'l Conf. on Management of Data. (2003)
10. Leighton, G., Müldner, T., Diamond, J.: TREECHOP: A tree-based query-able compressor for XML. In: Proc. 9th Canadian Conf. on Inf. Theory. (2005) 115–118
11. Tolani, P.M., Haritsa, J.R.: XGRIND: A query-friendly XML compressor. In: IEEE Proc. of the 18th Int'l Conf. on Data Engineering. (2002)
12. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley (1979)
13. Kropf, P., Babin, G., Hulot, A.: Réduction des besoins en communication de corba. In: Colloque Int'l sur les Nouvelles Technologies de la Répartition (NOTERE'98), Montréal, Canada (October 1998) 73–84
14. : Sox: Schema for object-oriented xml (last visit in Jan. 2007) http://www.w3.org/TR/NOTE-SOX/.
15. : Xstream (last visit in Jan. 2007) http://xstream.codehaus.org/.
16. Shannon, C.: A mathematical theory of communication. Bell Syst. Tech. Journal **27** (1948) 398–403
17. Huffman, D.: A method for the construction of minimum-redundancy codes. In: Proc. of the I.R.E. (1952)
18. Cormen, T.H., Leiserson, C.E., Rivest, R.E., Stein, C.: Introduction to Algorithms. Second edition edn. Volume Section 16.3. MIT Press and McGraw-Hill (2001)
19. MacKay, D.J.: Information theory, inference and learning algorithms. CUP (2003)
20. : bzip2 (last visit in Jan. 2007) http://www.bzip.org/.
21. Ng, W., Yeung, L., Cheng, J.: Comparative analysis of XML compression technologies. World Wide-Web Journal **9**(1) (March 2006) 5–33