

Xebu: A Binary Format with Schema-based Optimizations for XML Data

Jaakko Kangasharju, Sasu Tarkoma, and Tancred Lindholm

Helsinki Institute for Information Technology

PO Box 9800, 02015 TKK, Finland

Tel: +358 50 384 1518, Fax: +358 9 694 9768

`jkangash@hiit.fi, sasu.tarkoma@hiit.fi, tancred.lindholm@hiit.fi`

Abstract. XML is currently being used as the message syntax for Web services. To enable small mobile devices to use Web services, this XML use must not be too resource-consuming. Due to several measurements indicating otherwise, alternate serialization formats for XML data have been proposed. We present here a format for XML data designed from the ground up for the mobile environment. The format is simple, yet gives acceptable document sizes and is efficiently processable. An automaton-based approach gives further improvements when full or partial schema information is available. We provide performance measurements verifying these claims and also consider some issues arising from the use of an alternate XML serialization format.

Key words: XML and Semi-structured Data, Web Services, Mobile Environment, XML Serialization Format, Binary XML

1 Introduction

The use of XML has increased quite rapidly in the last few years, because it is a standard format for representing structured data. However, due to this ubiquity, XML is now even becoming used in areas where its designers never intended it to be. Our research is focused on the use of XML as a syntax for messaging in the mobile environment, typically in the context of SOAP.

Existing measurements of SOAP messaging performance indicate that marshaling and unmarshaling of typed data causes a significant loss in performance [1] and that latencies in wireless networks can be a significant cause of low performance [2]. This paper is mostly concerned with the verbosity and processor strain caused by the XML format, both of which have been found problematic in traditional industries moving towards SOAP use [3].

A natural idea is to replace the XML serialization format with something else that is compatible with XML. This concept is often referred to as “binary XML”, and the recent final report of the World Wide Web Consortium’s XML Binary Characterization (XBC) Working Group [4] details requirements for such a format and lists several existing formats.

This paper presents our format, called Xebu¹, which we use as a part of an XML-based messaging system of a mobile middleware platform. Our analysis of the format requirements [5] mirrors the Web Services for Small Devices use case of the XBC group [6]. Our additions are the requirements for some schema evolvability and special-purpose data encoding.

The main interesting feature of Xebu is that it requires less coupling between communicating peers than many other formats. While our default implementation retains state between XML messages in a stream, the format degrades gracefully in the case where state cannot be retained. In addition, if this state gets corrupted, it will be rebuilt gradually as new messages are processed instead of in one step immediately after corruption. Xebu also allows optimizations when a schema for messages is available, and these optimizations are designed to let the schema evolve in common ways without invalidating existing processors.

We describe the basic tokenization of Xebu in Section 2. Our model for schema-based enhancements is discussed in Section 3 and the actual implementation is described in Section 4. Section 5 presents performance measurements of our format and some others. Finally, Section 6 gives our conclusions and plans for future work.

2 Basic Format

We view an XML document as a sequence of events [7], and serialize it event by event. The serialization of a single event starts with a one-byte discriminator that identifies the event type and contains flag bits indicating how to process the rest of the event. This is followed by the strings applicable to the event type.

Similarly to other formats, such as Fast Infoset [8] and XBIS [9], we define a (one-byte) token for each string when it first appears in an event, and then use the token on later appearances. In contrast to other formats, the token value is given explicitly in the serialized form at definition time, which reduces the coupling between serialization and parsing by making the tokenization policy only concern the serializer. Therefore a token replacement algorithm does not need to be standardized to be interoperable. Furthermore, if the token mapping on the parser falls out of sync, it is sufficient for the serializer to invalidate tokens assigned after the last known good state instead of duplicating the parser's current state.

Since the typical case is that a single namespace has many associated local names and a namespace-name pair many attribute values, the token for a name includes its namespace and the token for a value includes both namespace and name. In this way, e.g., a repeated complete attribute or element name is replaced by a single token instead of having separate tokens for each string. In contrast to Fast Infoset, Xebu tokenizes namespace URIs instead of namespace prefixes.

All strings are serialized in a length-prefixed form. This is the length in bytes according to the encoding defined in the document start event. Use of length prefixes does not sacrifice streamability, since, as with XML, text content can be given in chunks by the application. Unlike in XML, in Xebu these chunks are explicit in the serialized

¹ The implementation is available at <http://www.hiit.fi/fuego/fc/download.html>.

form because of the length prefix. We also encode common data types directly in binary format, as this is more efficient in at least some cases.

3 Schema Optimization Concepts

In many cases when doing XML messaging, a complete or partial schema for the messages is available. In the case of SOAP the high-level structure is always available, and often schema information for some headers or the body is also specified. To be properly considered schema-aware [10], a serialization format needs to exploit the structure present in the schema to achieve its optimizations, and not just to recognize the defined names.

We construct two *omission automata, encoding* (EOA) for the serialization and *decoding* (DOA) for the parsing side, that process the XML event sequence one event at a time. The EOA will omit some events from its input, which the DOA is able to reinsert back into its output. As this technique is based on the event sequence representation, it is not Xebu-specific; it only requires that events be distinguishable in serialized form even if some are omitted.

Both automata are structurally similar. Each has a start state, from which the processing of a document begins. Transitions of each are labeled with an event and a type, and a state transition happens when the event label of a transition matches the event from the sequence being processed. The event may contain wildcards, so there can be several matching transitions. In these cases there is always the most specific transition that is the one selected.

Transitions in the EOA are of two types, `out` and `del`. The former of these outputs the event of the transition while the latter omits it. In the DOA, transitions can be of three types. A `read` transition is the most typical one. A `peek` transition is used to transfer from state to state without consuming the event from the sequence. Finally, a `promise` transition is used when the event is typed content. This will cause the DOA to inform the type of the data, so the parser can interpret it. A `promise` transition is always followed by a `read` transition for the decoded typed content event.

Edges in the DOA also carry two lists of events, the `push` list and the `queue` list. The events that the DOA passes upwards on a transition are first the transition's `push` list, then the event if it is a `read` transition, and finally the `queue` list. Any events of `del` transitions on the EOA side therefore need to appear in `push` or `queue` lists so that they get reinserted back into the sequence.

Each state in both automata also has a default transition to itself that is taken if no other transitions match the current event. These default transitions pass the events unmodified through the automata, which means that unexpected events are passed through as such. Hence it is possible to, e.g., add elements and attributes not present in the schema and still use automata of the old schema without problems.

An example RELAX NG [11] schema fragment of an element containing a typed value and its transformation into an EOA and a DOA are shown in Fig. 1. Each edge is labeled with its corresponding event and type. The EOA will omit the element start, the type attribute, and the element end, which the DOA's `push` and `queue` lists insert back. Fig. 2 shows an XML fragment that matches this schema fragment, and its processing.

Schema fragment: element ns:age { xsd:int }

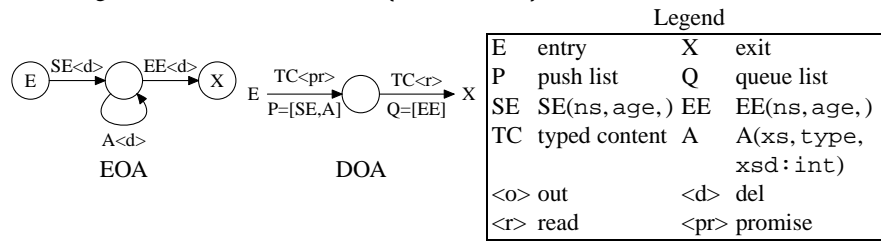


Fig. 1. Example transformation from schema to automata. Default transitions of the automata are not shown.

XML fragment	<code><ns:age xs:type="xsd:int">29</ns:age></code>			
Event sequence	SE(ns, age,)	A(xs, type, xsd:int)	TC(xsd, int, 29)	EE(ns, age,)
EOA processing	del	del	out	del
Produced fragment	%0C%1D			
DOA processing	promise TC	push[SE(ns, age,), A(xs, type, xsd:int)]		
	read TC(xsd, int, 29)	queue[EE(ns, age,)]		
Result sequence	SE(ns, age,)	A(xs, type, xsd:int)	TC(xsd, int, 29)	EE(ns, age,)

Fig. 2. Example processing of an XML fragment.

The serialized fragment consists of two bytes (shown in URL-encoded form): a type tag and the value of the integer.

4 Omission Automata Implementation

The automata are constructed recursively from an abstract syntax tree of a RELAX NG schema [12]. Each piece of supported² RELAX NG syntax produces a *subautomaton* and *entry* and *exit points* for it. The entry and exit points are states for the EOA and edges for the DOA, and are used by higher layers to build their own subautomata.

In general, the construction of a subautomaton for an element cannot determine whether the start and end events can be omitted, since it does not know the context in which the automaton is used. Because of this, each subautomaton has two entry and exit points, the *known* and *unknown* points. The higher-level construction will select the known points when it is certain that the subautomaton will be used, e.g., if it is the second item in a group construct. The unknown points will be selected when this is uncertain, e.g., when the subautomaton is a part of a choice construct.

The full generic handling of an element with a known name is shown in Fig. 3 in similar format as in Fig. 1. The box labeled M refers to the subautomaton built from the content of the element. In the figure the *k* and *u* subscripts refer to the known and unknown entry and exit points, while the M superscripts denote M's entry and exit points. On the DOA side only the modifications made to existing edges are shown, i.e., the additions to the push and queue lists. The DOA side has also inserted two states

² The interleave operator is not processed, and recursive elements are treated as errors.

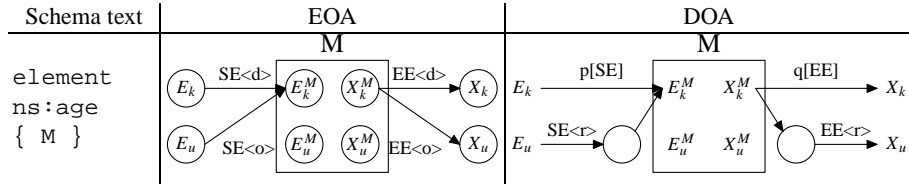


Fig. 3. Generic handling of an element. The p and q indicate a lengthening of the corresponding list (push or queue) from the appropriate end (front or back).

internal to the subautomaton. If these states are not used, they will get removed at the end when we reduce the automata to those states that are mutually reachable from the start state.

The star, i.e., a sequence of indeterminable length, requires additional handling. If what follows the star could be interpreted as included in it, omitting all events in between would make it impossible to determine this. Therefore any subautomaton built from a star is marked an *open* one. An element wrapped around an open subautomaton always has its end event inserted, so that the end of the star can be determined.

Existing approaches for schema optimizations include BiM for MPEG-7 [13] and Fast Web Services [14]. Like our approach, BiM uses automata that are driven by XML events. It achieves a higher level of compactness due to its bit-oriented nature, but at the same time requires a more rigid conformance to the schema than Xebu does. Fast Web Services, on the other hand, leverages existing work in structured data representation by mapping XML Schemas to ASN.1 schemas [15]. Like BiM, this is also tightly coupled to the schema, especially when the recommended Packed Encoding Rules are used.

5 Performance Measurements

There are several considerations on measuring a format for mobile devices. While document size is important, due to constrained memory and the use of battery it is also necessary to have fast and low-memory processing combined with a small implementation footprint.

We conducted measurements on a desktop machine and two mobile phones. The desktop machine had a 3 GHz Intel Pentium 4 processor, 1 GB of main memory, running Debian GNU/Linux 3.1. All of the measured processors are implemented in the Java programming language; we used Sun Microsystems's Java 2 Runtime Environment 1.5.0 as the Java platform and set its maximum heap size to 256 MB. The mobile phones were a Nokia 3660 and a Nokia 7610.

Our test data is a 698-message sequence from an event notification system that uses SOAP as its message format. Full schema information for the notifications was available and was utilized to generate omission automata. For XML we measured the performance of both Apache Xerces and kXML, the latter of which is designed especially for Java MIDP devices. We also measured the Fast Infoset [8] binary XML format. For the mobile phone we measured a 10-message sequence to avoid memory problems; the only processors available for that platform were kXML and Xebu.

Table 1. Comparison of different XML serialization formats

Format	Size		Serialization				Parsing				Foot (KB)
	Size (B)	Size (%)	Time (ms)	Thr (1/s)	DB (ms)	Mem (KB)	Time (ms)	Thr (1/s)	DB (ms)	Mem (KB)	
Xerces	3033	100.0	0.45	2201	0.13	39.77	0.76	1315	0.13	113.23	502.9
XercesZ	675	22.3	0.74	1356	0.14	42.45	0.87	1144	0.13	114.22	–
FI	1689	55.7	0.35	2825	0.13	13.88	0.48	2072	0.13	37.63	191.8
FIZ	687	22.7	0.62	1608	0.13	17.67	0.55	1833	0.13	39.11	–
kXML	3033	100.0	1.06	941	0.15	141.23	0.93	1078	0.14	73.69	29.1
XebuF	1304	43.0	0.53	1874	0.15	55.51	0.83	1198	0.18	58.48	–
XebuFZ	695	22.9	0.93	1079	0.15	56.04	0.91	1094	0.18	60.14	–
Xebu	807	26.6	0.45	2230	0.15	38.63	0.76	1309	0.17	50.87	52.1
XebuFS	493	16.3	0.56	1794	0.14	38.92	0.79	1264	0.18	55.78	–
XebuS	493	16.3	0.42	2357	0.15	34.52	0.77	1299	0.18	47.13	87.7

Table 1 shows the measurement results on the desktop machine. The main implementations are Xerces, FI (for Fast Infoset), kXML, and Xebu. These are suffixed with Z to indicate the use of gzip on the serialized form. Xebu measurements use two other suffixes: F (for forget) indicates that token mappings were not preserved from one message to another and S indicates that omission automata were used. Both total processing time and throughput are reported. Included in the processing time is data binding, which is also reported separately. The footprint, consisting of all Java classes loaded by our experiment, is reported for each distinct system³. Memory consumption did not vary between measurements, and the time deviations were all between 0.01 and 0.02 ms.

Xebu’s main advantages are in message size and footprint. While FI’s processing efficiency is higher, this appears to come at the cost of a complex implementation. Furthermore, the use of prefixes for tokenization in FI contributes to its higher throughput on serialization. The gzip results indicate that the local nature of the tokenization in Xebu and FI has no significant effect on the more global analysis performed by gzip. The similarities in data binding speeds are explained by our test data, which has only integers as typed content. A quick experiment verified that date and floating point values are encoded and decoded by Xebu at 2–3 times the speed of the normal string encoding specified for XML.

Table 2 shows the measurements results on the phones. The footprint measurement is after obfuscation with Proguard, and standard deviations are reported, as they varied more in this case. The kXML implementation performs much better, indicating that the Java virtual machines differ significantly. The data binding of Xebu appears problematic on the 3660, but otherwise the results are consistent with each other. Omission automata appear to provide more benefits in this case, probably due to their more static nature.

We note that if schema information can be assumed, coupling can be reduced by not preserving the token mappings without any loss (and even some gain on the mobile phones) in performance. The footprint increase could be mitigated by having generic automata that would drive the transitions based on a data structure. Furthermore, only

³ Non-Xebu formats do not include marshaling code; this is 7.6 KB on the desktop and 4.8 KB on the phones.

Table 2. XML serialization comparison on a mobile phone

Format	Serialization				Parsing				Foot (KB)
	Time (ms)	Thr (1/s)	DB (ms)	Mem (KB)	Time (ms)	Thr (1/s)	DB (ms)	Mem (KB)	
7610									
kXML	56.0±3.1	17.9	6.3±2.8	23.2±3.3	134.0±5.5	7.5	9.1±3.1	50.5±2.8	16.3
Xebu	60.0±5.1	16.7	8.1±3.6	33.1±2.2	134.1±4.4	7.5	10.8±3.0	50.9±3.5	22.0
XebuS	45.0±4.1	22.2	6.4±5.0	25.5±2.0	123.7±6.7	8.1	10.1±4.9	49.4±5.1	44.3
3660									
kXML	250.0±6.4	4.0	7.5±4.0	8.1±0.3	527.5±2.9	1.9	15.6±3.6	29.3±3.0	–
Xebu	211.1±15.6	4.7	19.2±7.4	22.9±2.5	503.8±43.9	2.0	33.9±8.4	30.7±4.1	–
XebuS	159.5±7.6	6.3	20.9±7.9	18.6±0.3	406.9±11.9	2.5	30.3±9.4	25.7±0.3	–

one such structure per schema would be needed, and the only required per-message information would be the automaton state. We estimate that the generic automata would take approximately 10 kilobytes (5 kilobytes obfuscated), and the automata of our measurements would take an additional 10 kilobytes.

6 Conclusions and Future Directions

In this paper we presented Xebu, a binary serialization format for XML data designed for the mobile environment. Xebu achieves an acceptable level of compactness without sacrificing either processing efficiency or simplicity of implementation. The design explicitly takes into account the loosely coupled nature of Web services.

Our approach of modeling an XML document as a sequence of events and then deriving the format and optimizations through this has proved to be useful. Schema-based automata are easier to describe and implement in terms of abstract sequences instead of byte sequences. Further, the abstract sequence provides a natural view of XML for mobile messaging applications, as we argue elsewhere [7], bridging the serialization and application use cleanly.

The low-level nature of event sequences also enables integration with most existing XML applications. The event sequence can be easily mapped to data models expected by them, and our XML processing system already includes SAX interfaces for the binary format. In the future we intend to explore whether exposing the binary tokens to applications would provide performance gains.

A major consideration in messaging systems will be security features, such as Web Services Security [16]. These rely on mutual understandability of the bytes being transmitted, so the above data model compatibility is not sufficient. Furthermore, using bridges to translate between binary and XML at the edge of the fixed network is not possible, since translation is not possible for encrypted content, and it would also invalidate signatures. Finding an interoperable solution that can be adopted piecewise will be important in the future.

References

1. Juric, M.B., Kezmah, B., Hericko, M., Rozman, I., Vezocnik, I.: Java RMI, RMI tunneling and Web services comparison and performance analysis. *ACM SIGPLAN Notices* **39** (2004) 58–65
2. Laukkanen, M., Helin, H.: Web services in wireless networks — what happened to the performance? In Zhang, L.J., ed.: *Proceedings of the International Conference on Web Services*. (2003) 278–284
3. Kohlhoff, C., Steele, R.: Evaluating SOAP for high performance applications in capital markets. *Journal of Computer Systems, Science, and Engineering* **19** (2004) 241–251
4. World Wide Web Consortium: XML Binary Characterization. (2005) W3C Note.
5. Kangasharju, J., Lindholm, T., Tarkoma, S.: Requirements and design for XML messaging in the mobile environment. In Anerousis, N., Kormentzas, G., eds.: *Second International Workshop on Next Generation Networking Middleware*. (2005) 29–36
6. World Wide Web Consortium: XML Binary Characterization Use Cases. (2005) W3C Note.
7. Kangasharju, J., Lindholm, T.: A sequence-based type-aware interface for XML processing. In Hamza, M.H., ed.: *Ninth IASTED International Conference on Internet and Multimedia Systems and Applications*, ACTA Press (2005) 83–88
8. Sandoz, P., Triglia, A., Pericas-Geertsen, S.: Fast infoset. *On Sun Developer Network* (2004)
9. Sosnoski, D.M.: XBIS XML infoset encoding. In: *W3C Workshop on Binary Interchange of XML Information Item Sets*, World Wide Web Consortium (2003)
10. Pericas-Geertsen, S.: Binary interchange of XML infosets. In: *XML Conference and Exposition*, Philadelphia, USA (2003)
11. Organization for the Advancement of Structured Information Standards: RELAX NG Compact Syntax. (2002)
12. Organization for the Advancement of Structured Information Standards: RELAX NG Specification. (2001)
13. Niedermeier, U., Heuer, J., Hutter, A., Stechele, W., Kaup, A.: An MPEG-7 tool for compression and streaming of XML data. In: *IEEE International Conference on Multimedia and Expo*. (2002) 521–524
14. Sandoz, P., Pericas-Geertsen, S., Kawaguchi, K., Hadley, M., Pelegri-Llopert, E.: Fast Web services. *On Sun Developer Network* (2003)
15. International Telecommunication Union, Telecommunication Standardization Sector: Mapping W3C XML Schema Definitions into ASN.1. (2004) ITU-T Rec. X.694.
16. Organization for the Advancement of Structured Information Standards: Web Services Security: SOAP Message Security 1.0. (2004)