Byte-efficient Representation of XML Messages

Jaakko Kangasharju and Kimmo Raatikainen
University of Helsinki, Department of Computer Science
jaakko.kangasharju@hiit.fi, kimmo.raatikainen@cs.helsinki.fi

XML is emerging as a standard for representing hierarchically structured data. It is also gaining ground as a messaging format in the form of SOAP. Unfortunately the XML format is quite verbose, which makes it less than optimal for the increasing number of wireless links in use. In addition, while XML is designed to be parsed efficiently, it is still a text format, and small mobile devices having wireless connections do not have much processing power. Below we present a binary representation of XML that is intended to alleviate both these problems. We also specify an extension framework for our format to better take into account the needs of special-purpose applications, such as the case where XML Schemas are available for the message syntaxes. A prototype implementation will soon be publicly available.

1. Position

1) SOAP/HTPP/TCP performance in wide-area wireless networks (e.g. GPRS in GSM) has been far from satisfactory. The problems are due to both communication and processing at high-end mobile phones. Therefore, binary encoding for SOAP is really needed. In addition, HTTP between SOAP and transport is unnecessary overhead. However, WAP binary XML is not sufficient. It only tokenizes predefined TAGs. We need an extensible tokenized binary representation.
2) Web services is important but not the ONLY anticipated usage of SOAP. For example, Liberty Alliance has also specified SOAP-based messaging. The same is also true for UDDI. Therefore, Web Service specific binary encoding will fork the code base. We would prefer pure SOAP binary encoding.
3) The decision that SOAP messages are XML Infosets may need to be revisited for wireless world. In 2.5G networks SOAP performance needs to be improved by factor 10-20. Binary encodings based on XML Schema and/or RDF representation is worth of careful analysis. Caching widely used and well-known URIs is useful. The same is also true for often used URIs in a session.
4) Interworking with "standard" SOAP can be arranged by a proxy server/gateway.
5) In addition to SOAP, there will be other usages of XML in mobile handsets. Therefore, the same binary encoder/decoder for any XML document would be benefitical. This is not, however, a mandatory requirement. In particular, XHTML binary encoding is not necessary. However, CC/PP information (and other context information expressed in XML) should have an efficient binary representation.

2. Assumptions

In many contexts the use of normal XML is quite sufficient. However, in the wireless world, XML is considered too verbose and too processor-intensive. We will make the following assumptions on our XML usage to help us design a more suitable encoding:

– XML is used as a message format to communicate in a distributed system
– Message syntax is described using XML Infoset (this is the case of SOAP), permitting alternate encodings
– The size of the messages is quite small, on the order of a few kilobytes at most
– In the system there exist point-to-point connections between applications on different hosts, through which messages are sent intermittently
– The connections persist independently of the sent messages; both sides in a connection are capable of holding state related to the connection

– At least one side of each connection may be a mobile node with a wireless connection and severely limited processing power

The assumptions here are derived from the demands of real distributed systems in a mobile environment. The use of XML as a message syntax is grounded in the need to exchange messages between diverse applications all using e.g. some message-oriented middleware system. The persistence of connections derives from the inefficiency of wireless connections in establishing transport connectivity, e.g. with TCP.

3. Basic Solution

From the assumptions we have decided on using a binary encoding, which we will call Xebu, for our XML messages. There are already some binary XML formats in existence, which are intended for general-purpose XML usage. While our solution is also intended to be fully general, we have decided to keep the basic format simple to generate, and to enhance it with application-specific extensions. Our goal is also to keep any extensions simple to decode. This simplicity may occasionally adds tokens that would not be strictly necessary; experiments could be conducted to determine whether elimination of these would offset the added complexity sufficiently.

The key concept in our binary format is the use of caches to store the associations between tokens and their textual representations. We keep several different caches for different kinds of XML items, and keep each cache size small to avoid the code pages of WBXML (we believe that the separation of different cache types as well as the various specializations alleviate the need for these).

Fundamentally, the representation of a document is the sequence of SAX events comprising the document. There is a binary token corresponding to each SAX event type. Here we describe only the representation for the basic SAX events: documents, namespaces, elements, attributes, and content. It is quite straightforward to extend this to other, rarer events.

A SAX event typically contains text data, such as the element name in a start element event. Such text data is encoded as a data block: the starting token followed by the data and a DATA END token. Data blocks are named by their starting token. Any DATA END bytes appearing in the data are byte-stuffed. The starting token may also include either a CACHE ENTRY or CACHE FETCH ag, in which cases the byte immediately following the starting token is the cache index to use (so each cache size is 256). Each different starting token gets its own cache. In case of CACHE ENTRY, the data follows as before and is entered in the cache at the specified index. In case of CACHE FETCH, the data is given by the cache index, so the event ends after the cache index.

Every document starts with a START DOCUMENT token. This token can use tags to specify extensions. Currently, no extension nor a facility for specifying them are given.

Namespace mappings begin, like in SAX, prior to the element and are not given as the element’s attributes. A namespace start consists of the URI as a NAMESPACE START data block followed by the prefix as a DATA START data block. Namespace mapping endings are implicit in the structure of the document and are not represented in Xebu.

An element start consists of the element’s qualified name as an ELEMENT START data block followed by the attributes. Each attribute consists of the name as an ATTRIBUTE NAME data block followed by the value as an ATTRIBUTE VALUE data block. The attribute list may always be terminated with an ATTRIBUTE END token; this must be done if the first byte of the element content has the value ATTRIBUTE NAME or ATTRIBUTE END, or if there would otherwise be two DATA END tokens in sequence. Elements end simply with an ELEMENT END token; the name of the ending element is always uniquely determined in XML, so it need not be repeated.

The content of an element is normally given as a DATA START data block. An encoder implementing the element caching described below should rarely encounter the need to cache element content, since element caching provides for most of these benefits. It is useful to note that a Xebu document can not contain two consecutive DATA END tokens unless they are a byte-stuffed byte with the same value as the DATA END token.

With SOAP messages it is common that the element content is of some datatype specified by XML Schema. Therefore XML Schema datatypes are encoded in binary. The specific binary encoding is not specified here, nor is a full list of all datatypes so encoded. With binary encoding the length of the data is known already, so the content is not a data block, just the plain binary encoding.

4. Optimizations

There are three basic optimizations to the basic format that are included as standard in Xebu. These are called pre-caching, element caching, and item omission. Of these, pre-caching and item omission are specific to document syntax descriptions (such as DTDs); this document describes these optimizations for SOAP.

Pre-caching is simply filling the caches with values prior to encoding any documents. These pre-cached values are expected to exist in most documents, and so should not be evicted from the caches at all. The values should be extracted in some standard way (which is not yet specified) from the syntax description.

In all cases, namespaces for XML Schema (with prefix xsd) and XML Schema instance (with prefix xsi) are pre-cached. Similarly, the attribute name type from the instance namespace, and any names of XML Schema datatypes are pre-cached.

For SOAP, pre-cached values also include namespaces for envelope and encoding. The assumed prefixes for these are soapenv, and SOAP-ENC or soapenc (both should exist). Pre-cached element names are Envelope, Header, and Body with their pre-cached namespace prefix. Also, the attribute name encodingStyle in the soapenv namespace and the attribute value equal to the encoding namespace are cached in their respective caches.

Element caching adds a new tag for the ELEMENT START and ELEMENT END tokens. This tag signifies that the full element content is either fetched from or entered into the element cache. Unlike the normal caches, this cache has 32768 slots. The tagged token is followed by the index encoded as a compressed short int (this is why it is one bit short of the full short int range). In the case of ELEMENT END, it means that the element just ended is to be cached, and in the case of ELEMENT START, it means to fetch the element.

Element caching should not be used at the top levels of an XML document by a generic encoder, since it is unlikely that the full document would be repeated. With SOAP, it is typically wise to start element caching at the third level, i.e. leave Envelope and Body alone. Of course, if the encoder has more knowledge of the messages sent, going further up may pay off.

This is specified as element caching, since when used only on elements, the start and end points are clearly marked without the need for external markers. It would be straightforward to add new tokens to delimit the to-be-cached material and thereby be able to cache any sequence of SAX events in the document. Future versions of Xebu may specify this more general caching, if it does not impact decoder simplicity.

Item omission is the final optimization described here. It is also the least specified, since there is little experience of its effects. The idea behind it is that from the syntax description it is possible to deduce the existence of e.g. some element start tags without them appearing explicitly.

A useful item omission facility in all cases would be to assume that any namespace prefixes defined on the whole document also apply for all further documents in the stream. It is expected that the documents sent are at least similar, and the namespace prefixes could be chosen to not collide. This would save some space.

For SOAP it is assumed that all SOAP messages carry the four namespaces envelope, encoding, XML Schema, and XML Schema instance (in this order). If the prefixes are the pre-cached ones from above, they are assumed for all SOAP messages. If the prefixes are not the cached ones, they will need to be specified.

Also, with SOAP it is not necessary to include the Envelope or Body elements, since they are always present. The attribute encodingStyle is also expected for any element where it is permitted, with the value being the encoding namespace URI. If an alternate encoding is wished, it is specified normally. Due to the loose nature of SOAP syntax, this is all that can be specified.