

CubeWerx Position Paper for Binary XML Encoding

*Dr. Craig S. Bruce, Sr. Software Developer
CubeWerx Inc., Gatineau, QC, Canada
csbruce@cubeWerx.com*

Introduction

CubeWerx Inc. [CW] has been involved with the OpenGIS® Consortium (OGC) [OGC] in developing various specifications for interoperability among Geographic Information Systems. These specifications have focused on using XML encoding for the interchange format, but the realizations of some of these formats have been bulky and slow to process. One particular format is GML (Geography Markup Language), which is primarily used for interchanging geographic-feature data, including geometries that may have many thousands of coordinate values each. Processing hundreds of megabytes of textual feature information that is dominated by floating-point numbers is very time consuming. Dense numeric data is very poorly suited to be processed efficiently when represented by text.

As a result of these problems, CubeWerx has participated in the development of a specification for efficiently encoding general XML documents and dense-numeric scientific-data formats such as GML which is called “BXML” for “Binary XML”. This specification has the status of a Discussion Paper within OGC and is publicly available for comments [BXML].

Discussion of Issues and Objectives

Binary formats tend to be unduly discounted for interoperability, especially in this context. When one selects “view source” in a web browser, one does not actually look at the raw text of a web page; one usually looks at formatting and syntax coloring applied by the browser. The same thing can be done with a binary representation of XML: viewer/editor tools can hide whether textual or binary XML is being used and the user may not even know or care.

Also, textual XML can easily be regenerated from an equivalent binary encoding if a person wants to read the content as a text document. But most people do not edit XML or even HTML documents by hand. If a binary encoding were adopted by W3C, the major XML-editing tools will be modified to support it. Consider GZIP [GZIP] compression. Does GZIPping a text-XML document make it unusable to web browsers, tools, and people? No, they simply unGZIP the *binary file* and use the recovered content.

Domain-specific compression can be a dangerous idea. Implementations of such schemes are likely dependent on the schema language for the document and when the “flavour of the year” changes for validation languages, existing implementations suffer. These may also make non-validating simple implementations impossible. Generic compression is resilient, orthogonal, and eternal. Some experimentation has shown that the BZIP2 [BZIP2] compression format is very good at compressing XML content. Unless

domain-specific compression formats are enormously better than any generic compression format, they should be avoided for official endorsement.

Also, producing the smallest possible document, while important, should not be an objective held to the exclusion of making the binary format well suited for being processed as a local file. A smaller file is faster to send across a network, but as XML-based formats continue to become more popular, they will start replacing file formats that are normally processed locally on a machine, such as word-processor or spreadsheet documents. Performance in this environment should not be sacrificed for compression, and often compression should not be applied at all to such documents. It may also be beneficial to send uncompressed documents across local-area networks to avoid compression/decompression costs.

Additionally to the issue of changing validation languages, the binary encoding should be as stand-alone from the validation language as textual XML is, in the sense that a non-validating parser can use any textual-XML document without any problems regardless of what validation language is used. The simplicity of the XML format is one of the major reasons for its success. The entry barriers for simple processing systems are very low.

The general notion of “pre parsing,” if we understand the term correctly, may be dangerous as well. Pre-parsing for what language? For what platform? For what higher-level parsing algorithm? A good binary format should be one that stands on its own, and for which compactness of the uncompressed form is not sacrificed. Some pre-parsing forms may be even slower to parse than textual XML because of increased size. No binary encoding should be larger than the text encoding. The structural simplicity of the binary encoding is important also. It need not be any more “complex” than the text-XML encoding to be highly useful.

The APIs for accessing the binary format will need to be extensive enough to allow application programs to directly access binary-encoded numeric values, numeric arrays, and BLOBs (Binary Large Objects) in order to be efficient. Converting numbers or BLOBs from binary to text and back again is a time-consuming process. With end-to-end binary encoding, no conversion is needed at all with modern processors other than occasionally swapping the byte order of values. If an API is used which supports only the text encoding of data, it will significantly reduce the benefits of binary encoding for numeric and BLOB data types.

BXML Encoding

BXML [BXML] is a straightforward encoding for XML that is designed for efficient processing in general with an emphasis on dense numeric data, which is one of the weakest areas for textual XML. It is open, patent-unencumbered, and about as straightforward to implement a parser/generator for as for textual XML. (In fact, in our own implementation, it is even easier, because the structures are more regular.) These properties make it software-developer-friendly, which increases the chances of de-facto adoption.

The structures of the BXML format is defined to be nearly one-to-one to the structures of the textual-XML format and are also very close to the structures that are used by various

libraries to represent tree nodes in memory for efficiency. So, the overall structure will be familiar and documents can easily be translated on an almost token-by-token basis to and from textual XML with no loss of information. This gives BXML the various advantages that textual XML has and makes a BXML file a drop-in replacement for an XML file, as it stands alone and performs exactly the same function. The accessibility of XML is not compromised by BXML.

Because a BXML document serves exactly the same purpose as an XML document, it is not tied to any particular schema language in the same way that an XML document is not. The validity of a BXML file is defined as being the validity of the structures of the BXML file if they were directly lexically translated into textual XML. And that's that. XML validators will normally tokenize the input XML stream into some internal binary representation, so there is usually not even the need to translate BXML into the textual lexical equivalent to perform validation. BXML is a step closer than XML to being easy and efficient to validate.

The primitive data types inside of a BXML file are defined from scratch in a very simple way and are easy to understand and use. Only a handful of numeric primitive types are needed and they are exactly the same raw binary format that all modern computer architectures use, for efficiency and simplicity. A derived numeric type called a "Count" is defined to have a variable length for space efficiency, with count values less than 240 occupying only a single byte, which is important since counts are used very frequently and the values are frequently small.

Strings are defined using a small header, a byte length, and the raw string content. The length is given up front so that string content can be handled using raw-block I/O operations. The byte content of all strings in the file use the character-encoding representation that is identified in the BXML header using the same identifiers that XML uses, so the internationalization of XML is not compromised by BXML since BXML uses the same logical mechanisms.

Numeric arrays are represented similarly to strings as a small header, an element-type identifier, a byte length, and a raw block of numbers. Arrays correspond to the XML-*Schema* concept of a "list" and they can also be handled using raw-block I/O and memory operations for exceptional efficiency. Most importantly, lists of numbers do not need to be converted to and from their textual representation, which is an expensive operation.

All primitive types that are used in user-supplied textual content are type-identified by a leading-byte indicator, which is collapsed for numbers small enough to fit into the `Count` type. Identifying the data types in-line removes any reliance on an external schema to identify the content types and allows for great flexibility. Since the validation is defined in terms of lexical equivalence, primitive types may be arbitrarily substituted at the discretion of the file writer. For example, if a schema identified a content field as consisting of a list of double-precision floating-point numbers but the writer is aware that only integers between 0 and 255 are being written out in one particular instance, it can write the list as an array of byte values. Conversely, if a field contains a value that cannot be represented by the primitive types, such as a 200-digit integer, the writer can simply write out the number as string content instead, and the document will still validate and be

correct. Allowing a string representation of numbers is no significant imposition on an implementation since most conceivable parsers will also implement XML support so they or the applications that use them need to deal with string representations of anything anyway.

Some might argue that a binary-primitive language such as ASN.1 should be used for representing primitive information. We contend that ASN.1 is a “bloated” specification that is not needed here. Some parts of the ASN.1 standard for string handling have been described informally as being “complete insanity”. Given the very limited set of structures that need to be represented in content in XML (only strings, numbers, and lists since most structuring is done using the higher-level markups) and given the lexical equivalence and substitutability of strings for anything that cannot be represented more primitively, a very simple typing system is very appropriate and, in fact, desirable. Ask software developers which typing system they would rather implement and support.

BXML features direct BLOB support. BLOBs are awkward and inefficient to use in textual XML because they must be recoded into some intermediate representation such as BASE-64 because XML is text-only. To support translation back to text format, an identifier is given in the BLOB token of what representation to use when regenerating text, such as using BASE-64 or hexadecimal strings, which is consistent with XML-Schema type-identification and validation. A validating translator could also take XML input and directly materialize a BLOB in BXML output by observing the data type of the content. The same is true of numbers and booleans.

For efficiency and compactness, the BXML format uses a central, global string table to store the names of all elements and attributes. This is an obvious optimization for a binary encoding since it is more compact to represent the index number of a name in-line than the whole string, and it is more efficient to process since a number can be read faster than a string, and auxiliary information can be associated with an entry in the in-memory copy of the string table to make processing faster. Text content can also reference string-table entries, to optimize content values that are frequently repeated, such as enumerated code values.

The complete string table is present in the file to allow the file to stand alone without reliance on external schemas or definitions, and it is organized into fragments that may be spread throughout the file to allow streaming of the format and to not require a writer to know in advance all of the symbols it might write out. Most formats will only use a limited number of symbols in a document instance, so dumping out a complete list in advance of everything that could possibly be used may be wasteful.

However, the fragmented structure of the string table does not preclude random access to the file content. The BXML format contains a trailer token that can optionally include an index to all of the string-table fragments that are spread throughout the file. The writer will be able to build the index at the end of the file because the byte offsets of all of the fragments will be known if the writer recorded them. A random-access reader can regenerate the symbol table as needed.

The trailer token can also include a simple index for looking up element-token locations of literal values resolvable for indicated XPath expressions. For instance an index could

be set up for all literal values that resolve from the XPath expression “//@ID”. This would give direct access to all elements that are identified with an ID attribute value. The writer can build an index for any expressions it wishes, and the index can be arbitrarily augmented later on since it is located at the end of the file; the trailer token can simply be overwritten with a new one. Random access is only possible with BXML files that are uncompressed. This is not necessarily a large imposition since random access will generally only be done on local files and it will be more efficient to access a local file if it is uncompressed. BXML files can be compressed and uncompressed as needed. BXML relies on generic compression methods such as GZIP. Dynamic update is not considered.

Performance

CubeWerx is in the process of developing an open-source reference implementation of BXML called “cwxml”. This library supports both plain-text XML and BXML and features automatic handling of GZIP compression and likely will be extended to include BZIP2 compression. The implementation has been tuned for high performance for both BXML and text-XML formats, where the text performance appears to far exceed other common libraries for XML processing. The testing platform is an AMD Athlon™ XP1800+ server running Red Hat Linux 7.3 with the GCC 2.96 C compiler. The test files were cached in the OS memory so as to measure the parsing speed of the encodings rather than the disk I/O speed.

Performance testing of the library has been carried out for the general processing speed of BXML and the processing speed for numeric data. The general test is a simple utility that converts both ways between XML and BXML encodings and the general scanning speed is tested by running it in a read-only mode on an input file. This is a useful scanning-speed test since each token of the input file is read and the content is passed up through the API to the application program. A custom API is used that preserves numeric and blob data through to the application program and which can operate in three different modes: a raw tokenizer similar to the SAX API but which is “consumer-pull-oriented” rather than “producer-push,” an intermediate mode which reads tokens plus the subtree of the attributes of an element, and a DOM-oriented mode.

Two sample files are tested, one is a “capabilities” statement for an OGC Web Map Server [WMS] and the other is a technical-specification document from the OpenOffice.org word processor [OOO], which uses a native XML format for its files. The sizes (using SI notation, i.e., base-10) of the XML and BXML versions of these files with GZIP and BZIP2 compression applied to them are as follows:

<i>File</i>	<i>XML</i>	<i>BXML</i>	<i>XML +GZIP</i>	<i>BXML +GZIP</i>	<i>XML +BZIP2</i>	<i>BXML +BZIP2</i>
WMS Capabilities	935 KB	521 KB	79.1 KB	76.6 KB	53.6 KB	56.4 KB
OpenOffice.org Doc	702 KB	427 KB	76.6 KB	75.5 KB	52.1 KB	57.6 KB

The uncompressed BXML representation is significantly smaller than the XML representation. This is to be expected, as just about every BXML structure is almost always smaller than the corresponding XML markup. The capabilities XML document is

further filled out by using an indentation of two spaces per nesting level, which is common for XML documents. The OpenOffice.org file has no unnecessary whitespace at all. The compressed results are similar in size. BZIP2 does a good job on XML data for a generic compression method.

The scanning performance for all three modes of the API with all of the files except the BZIP2-compressed ones (since that is not supported yet) is as follows, with elapsed time in seconds and speed in MB/second:

<i>File</i>	<i>Encoding</i>	<i>Raw (~SAX)</i>		<i>Scan</i>		<i>DOM</i>	
		<i>Time</i>	<i>Speed</i>	<i>Time</i>	<i>Speed</i>	<i>Time</i>	<i>Speed</i>
WMS Capabilities	XML	0.044	21.3	0.048	19.5	0.080	11.7
	BXML	0.0050	104	0.0081	64.2	0.026	20.2
	XML+GZIP	0.058	1.36	0.064	1.23	0.101	0.783
	BXML+GZIP	0.012	6.18	0.016	4.94	0.033	2.30
OpenOffice.org Document	XML	0.055	12.8	0.061	11.6	0.101	6.98
	BXML	0.0089	47.9	0.014	30.1	0.047	9.17
	XML+GZIP	0.063	1.21	0.069	1.12	0.109	0.70
	BXML+GZIP	0.015	5.04	0.021	3.73	0.053	1.41

Clearly, the BXML encoding is greatly faster to process than XML. Beware that the scanning speed for the uncompressed XML file is deceptively high, since the input file is larger though it contains the same information. The uncompressed BXML Capabilities file is 9.2 times as fast to read with the SAX-like interface and 4.8 times as fast for compressed. The uncompressed BXML OpenOffice.org file is 6.2 times as fast and 4.2 times as fast for compressed.

This test brings up an implementation issue related to character-set handling. The implementation presently uses ISO-8859-1 string encoding internally and the Capabilities file is also encoded in ISO-8859-1 but the OpenOffice.org file is encoded in UTF-8. BXML encoding works most efficiently when strings do not need to be checked at all, but character-set translation normally requires strings to be translated character-by-character thereby slowing down the process. Since the BXML writer presently writes out only ISO-8859-1 strings, the scanning of UTF-8 has been simulated for fairer comparison by scanning each character of each input string and counting the number of non-ASCII characters. (The input data is predominately ASCII-only text.) The SAX-like scanning speed for the OpenOffice.org file is 13.6 MB/sec for XML and 55.0 MB/sec for BXML if the files are scanned as being ISO-8859-1.

Note that the text-XML implementation in the `cwxml` library is heavily optimized, so the results between BXML and text XML are narrower than they would be when comparing BXML to other text-XML implementations. For comparison purposes, the results of the SAX interface of the popular `libxml2` [LIBXML2] library were measured directly and performance for other popular libraries were extrapolated as best as possible from a

performance comparison on the libxml2 web site [LIBXML2PERF]. The throughputs of these various libraries with the plain XML documents are therefore as follows:

<i>Library</i>	<i>libxml2 scale</i>	<i>WMS Capabilities</i>	<i>OpenOffice.org Doc</i>
cxxml	3.31×	21.3 MB/sec	12.8 MB/sec
expat	1.14×	6.66 MB/sec	4.91 MB/sec
libxml2	1.00×	5.84 MB/sec	4.30 MB/sec
Xerces	0.325×	1.90 MB/sec	1.40 MB/sec
Sun	0.263×	1.54 MB/sec	1.13 MB/sec
Oracle	0.189×	1.10 MB/sec	813 KB/sec

Since we believe that it is much easier to implement a highly efficient scanner for the BXML format than for the XML format, as has been our experience, the performance difference between the BXML and XML encodings as measured with the `cxxml` library should be appropriately increased in general.

The testing for numeric data was carried out by defining an ad-hoc simple encoding format for RGB images and writing programs to convert to and from the PPM-raw image format [PPM]. Representing image data in an XML-based format has been the butt of many jokes about the enormous inefficiency of textual XML encoding for some types of data, but if the temptation to over-structure the pixel representation is resisted and BXML encoding is used, the result is an XML-equivalent format that could be accessed by the various XML tools and which is just as efficient at encoding imagery as PNG [PNG] format, both in terms of file size and processing speed.

The ad-hoc image format is called XDI for “XML Demo Image” and is outlined very abstractly as follows:

```
<XmlDemoImage version="1.1.0">
  <Header>
    <Width>x</Width>
    <Height>y</Height>
    <SampleType>byte</SampleType> <!-- or "double" -->
  </Header>
  <Scanline row="i">
    <RgbSamples>r g b r g b ...</RgbSamples>
  </Scanline>
  ...
</XmlDemoImage>
```

The imagery could be represented even more simply by using one really large list of numbers to represent the entire matrix, in which case it would be almost identical to PPM format (small amount of text, big raw matrix), but the above is probably more representative of a good compromise between using XML mechanisms to structure the higher-level information but using raw arrays to represent the low-level, performance-critical information.

The writing and compression speed and result-file size are tested using a program that reads a PPM image and writes the corresponding XDI file in XML and BXML encodings with optional compression. The source test file is a 2000×1000-pixel 24-bit RGB image of color-coded global land and sea elevation/depth data. Testing is also carried out for the XDI file encoding samples using floating-point doubles (with image-sample values between 0.0 and 1.0), to gauge performance for more-mathematical applications.

File-size and write-speed results for the base-image and PNG formats were acquired for comparison purposes using the common `cat`, `pnmtopng`, `gzip`, and `bzip2` programs. These formats do not support double samples, so only byte samples are tested. The results are:

<i>File</i>	<i>File Size</i>	<i>Write Speed</i>
Source PPM	6.00 MB	158 MB/sec
PPM+GZIP	2.56 MB	2.81 MB/sec
PPM+BZIP2	1.38 MB	0.370 MB/sec
PNG	2.55 MB	0.181 MB/sec

The file-size and write-speed results for the XDI-based formats are as follows:

<i>File</i>	<i>Byte Samples</i>		<i>Double Samples</i>	
	<i>File Size</i>	<i>Write Speed</i>	<i>File Size</i>	<i>Write Speed</i>
XML	21.0 MB	5.19 MB/sec	107 MB	7.32 MB/sec
BXML	6.02 MB	109 MB/sec	48.0 MB	140 MB/sec
XML+GZIP	3.43 MB	0.398 MB/sec	6.80 MB	0.287 MB/sec
BXML+GZIP	2.57 MB	2.27 MB/sec	4.88 MB	0.805 MB/sec
XML+BZIP2	1.58 MB	0.0667 MB/sec	2.10 MB	0.0107 MB/sec
BXML+BZIP2	1.39 MB	0.387 MB/sec	1.87 MB	0.0492 MB/sec

The uncompressed-BXML file is obviously much smaller than the XML file because the BXML encoding uses only one byte per color-channel-sample value whereas the text encoding normally uses four characters. The GZIP-compressed file is significantly smaller as well because apparently the GZIP algorithm is not necessarily all that good at compressing long strings of ASCII digits and also because any compression algorithm will be imperfect at removing *all* redundancy from the source data, and raw BXML samples have significantly less redundancy to begin with.

It is interesting that the BXML/PPM+BZIP2 files are actually much smaller than the PNG file because BZIP2 compression is so much better than the GZIP compression used by PNG format in this test. In other testing, by adding an optional `filter` attribute to each scanline, GZIPped XDI files are usually about the same size as PNG files for various test images, since the two formats are then structurally very similar. It is difficult to tell whether BZIP2 compression is usually better than GZIP for imagery.

The uncompressed-BXML writing speed is obviously enormously faster than XML. This is because the source data is written out using raw block writes of the input PPM sample data, and because the costly conversion from binary integers to their textual lexical representation is not performed. Note that the plain-XML writing-speed figure is deceptively high because the file is so much larger. The adjusted XML writing speed would be 1.49 MB/sec for the 8-bit samples if the produced file were the same size as the BXML file; the BXML writing is 73 times as fast.

The GZIP- and BZIP2-compression speed of BXML is also much faster. This is because the algorithms are relatively slow at compressing data, so the much smaller size of the uncompressed BXML inputted to the algorithms means much less expensive processing. However, this does not explain all of the compression-speed difference, so the algorithms also must be especially slow at compressing long strings of mostly ASCII digits (which have a narrow range of character of values).

The scanning speed is measured by using the read-only mode of a program for converting from the XDI format back to PPM. This mode causes all of the file data to be brought up through the API but not be actually used by the application. Source-data performance is included for comparison. The performance results are as follows:

<i>File</i>	<i>Byte Samples</i>		<i>Double Samples</i>	
	<i>Read Time</i>	<i>Read Speed</i>	<i>Read Time</i>	<i>Read Speed</i>
Source PPM	0.0128 sec	470 MB/sec	-	-
PPM+GZIP	0.144 sec	17.7 MB/sec	-	-
PPM+BZIP2	1.27 sec	1.09 MB/sec	-	-
PNG (GZIP)	0.770 sec	3.31 MB/sec	-	-
XML	1.05 sec	19.9 MB/sec	6.28 sec	17.0 MB/sec
BXML	0.0320 sec	188 MB/sec	0.109 sec	443 MB/sec
XML+GZIP	1.37 sec	2.51 MB/sec	7.62 sec	0.893 MB/sec
BXML+GZIP	0.17 sec	15.1 MB/sec	0.764 sec	6.39 MB/sec

Again, the plain BXML processing is enormously faster, approaching the raw file speed of the OS, because the image samples which dominate the file are read using raw-block operations. The XML processing lags far behind because the input data must be scanned character-by-character and because all of the sample values must be converted back to the binary representation for use by the application. This conversion is well optimized, but the BXML reading is still 33 times as fast for `byte` samples and 58 times as fast for `double` samples. The GZIP-compressed-BXML processing is also eight to ten times as fast since less data must pass through the GZIP decompression algorithm and no text-numeric decoding is needed. The PPM+BZIP2 and PNG reading times are tested using the common `bzcat` and `pngtopnm` programs.

Conclusion

The binary encoding used in BXML is greatly faster than XML encoding for processing general data and is enormously faster for dense numeric data. The uncompressed files are also significantly smaller, especially for numeric data, and are more efficient to compress and uncompress. A BXML file is also a direct stand-alone drop-in replacement for an XML file, which we believe is crucial for the success of standardizing a binary encoding for XML.

References

[BXML] OGC 03-002r8 (May 2003), *Binary-XML Encoding Specification*, Version 0.0.8, Craig Bruce (ed.). <<http://www.opengis.org/techno/discussions/03-002r8.pdf>>.

[BZIP2] BZIP2 compression library, Julian Seward, <<http://sources.redhat.com/bzip2/>>.

[CW] CubeWerx Inc. Independent geo-spatial software vendor. Main web site: <<http://www.cubewerx.com/>>.

[GZIP] IETF RFC 1952 (May 1996), *GZIP File Format Specification Version 4.3*, L. Peter Deutsch, <<http://www.ietf.org/rfc/rfc1952.txt>>.

[LIBXML2] LIBXML2 XML-parser library. Main web site: <<http://www.xmlsoft.org/>>.

[LIBXML2PERF] LIBXML2 performance comparison to other common libraries on site web page: <<http://www.xmlsoft.org/>>.

[OGC] OpenGIS® Consortium. Geo-spatial interoperability organization. Main web site: <<http://www.opengis.org/>>.

[OOO] OpenOffice.org office-productivity suite. Main web site: <<http://openoffice.org/>>.

[PNG] PNG (2003), *PNG: Portable Network Graphics: A Turbo-Study Image Format with Lossless Compression*, Greg Roelofs, et al., <<http://www.libpng.org/pub/png/>>.

[PPM] Portable Pixmap file format. Described at: <<http://www.cis.ohio-state.edu/~parent/classes/681/ppm/ppm-man.html>>.

[WMS] OGC 01-068r3 (January 2001), *Web Map Service Implementation Specification*, Version 1.1.1, Jeff de La Beaujardière (ed.), <<http://www.opengis.org/techno/specs/01-068r3.pdf>>.