Carnegie Mellon University
Software Engineering Institute

USING XML TO EXCHANGE FLOATING POINT DATA

John Klein

10 February 2022

Motivation

Consider a computation using floating point arithmetic to produce some result values. This computation executes in a single process (or in multiple processes using a mechanism like RPC for interprocess communication). If the computation is decomposed and distributed over a set of processes that use an XML-based mechanism to exchange intermediate computation results as floating point values, then the results of the distributed computation will generally be different from results produced by executing the computation in a single process, unless care is taken to preserve precision in the XML literals exchanged.

Introduction

This short paper explains some issues that arise when XML is used to exchange floating point values, how to address those issues, and the limits of technology to enforce a correct implementation. We begin by specifying the problem to be solved and the correctness conditions of a solution. We then provide brief background on the relevant aspects of XML and floating point data arithmetic. We conclude by describing how to solve the problem using the features of several programming languages.

Problem Statement and Notation

M_s is the representation of a floating point value in the source process memory

L_e is an XML literal representation of M_s (text using decimal digits). This literal exchanged from the source process to the destination process in an XML document.

M_d is the representation of L_e as a floating point value in the destination process memory.

End-to-end error is defined here as the difference between source in-memory value and destination in-memory floating point value: $E = M_s - M_d$

We want to define a series of transformations:

$M_s \rightarrow L_e \rightarrow M_d$ such that $M_d = M_s$, or equivalently $E = 0$

Background

XML Data Types

Each XML data type is defined in terms of a *value space* and a *lexical space* [1, §2.1]. The value space defines the information that can be represented using the data type, while the lexical space defines the literals (i.e., text) that represent those values in XML. For example, see the XML Schema standard definition of the `float` data type [1, §3.2.4].

An XML document contains literals that represent values (L_e in the Problem Statement above). This is in contrast to mechanisms like RPC that exchange data values (M_s above). For most data types, this distinction between values and literals is not a concern. However, for floating point data types, a value is generally represented using base-2 fractions (e.g., the IEEE 754 format discussed below), while the literal is text using base-10 digits. The implementor of an information exchange must decide how many significant base-10 digits to use in the literal representation of the value.

Floating Point Arithmetic

Most computers store floating point values using formats defined by the IEEE Standard for Floating-Point Arithmetic (IEEE 754) [5], which represents a value using base-2 normalized mantissa and a base-2 exponent.

XML Encoding

XML documents can be encoded in various ways. Encoding as text is probably the most common and recognizable, however there are also binary encodings such as Efficient XML Interchange (EXI) [3] and Fast Infoset [2]. The choice of XML encoding method does not affect the problem discussed above or the solution discussed below: All XML encoding methods preserve the value represented by a literal through the encoding and decoding process, regardless of data type.

How to Use XML to Exchange Floating Point Values

We must preserve sufficient precision so that the error in the base-2 $\rightarrow$ base-10 $\rightarrow$ base-2 conversions is less than the smallest value that can be represented using our base-2 format [5, §5.12.2]:

- Error-free exchange of single float requires representing 9 significant decimal digits in the lexical space
- Error-free data exchange of double float requires representing 17 significant decimal digits in the lexical space

Note that significant digits are not the same as fractional digits. For example, 1.23, 12.3, and 123 all have 3 significant digits.

Also note that these requirements specify worst case limits for the number of significant decimal digits. For example, the value $1.0_2 \times 10_2^{-1_2} = 0.5_{10}$ can be exchanged error-free using a literal with just 1 significant decimal digit.

Implementation Considerations

This subsection discusses features and limitations of the technology involved in implementing a solution that satisfies these requirements. There are many ways to develop a correct solution, and this section provides some guidance to help make tradeoffs. It assumes some familiarity with XML processing and with programming languages.

XML Schema's *facet* mechanism [1, §2.4] can constrain the literals that are allowable to represent the value of an element, however the `pattern` facet is the only one that can restrict literals to have a *minimum* number of significant decimal digits. Use of the `pattern` facet to enforce the restriction in this case is possible, but not practical: This restriction would have to be represented by a complicated regular expression that must process a diverse set of inputs: e.g., "-1E4, 1267.43233E12, 12.78e-2, 12 , -0, 0 and INF are all legal literals for float" [1, §3.2.4.1]. Executing this validation at runtime would have a performance impact on any implementation. Finally, XML processor APIs such as JAXB (discussed below) do not execute `pattern` facet validation at runtime.

The application programming interfaces (APIs) for XML processors do not provide direct support to constrain the number of significant decimal digits in the lexical representation of a floating point value. APIs such as SAX [1] treat lexical literals as unconstrained strings when creating XML documents.

The mechanism to transform a base-2 floating point representation to a base-10 literal (text using decimal digits) in the XML lexical space depends on the programming language being used. For example, in the C programming language, the conversion would be performed using a library function like `snprintf`[2]. Many other modern programming languages provide a similar function. The correctness requirements stated above can be satisfied using `snprintf` with a format string using the `E` conversion specifier, which allows specification of significant digits. For example, the format string "`%1.8E`" will preserve 9 significant digits (1 digit to the left of the decimal point and 8 digits in the fraction to the right of the decimal point). A format string using the `f` conversion specifier is limited to specifying only the number of fraction (not significant) digits. The format string "`%1.9f`" will always contain 9 fraction digits (whether needed to satisfy the correctness requirements or not—for example, the literal may have sufficient significant digits to the left of the decimal so that no fraction digits are needed), and so will produce larger XML documents than using an "`E`" format string, on average.

The C++ language provides a library function `std::toString`, however unlike the similarly-named Java function discussed below, this function produces a literal with 6 fraction digits.

The Java programming language has a family of `toString` functions that relieve the implementor from concerns about precision. When operating on floating point values, these methods produce a literal with a variable number of significant decimal digits, however in all cases there are sufficient significant decimal digits in the literal to exactly represent the floating point value [4] and provide an

[1] http://www.saxproject.org
[2] https://www.gnu.org/software/libc/manual/html_mono/libc.html#Formatted-Output-Functions

error-free value exchange. Java also provides an object-to-XML API called `JAXB`. While there is no specification for the conversion behavior of JAXB, it appears to use the `toString` function to transform floating point values to literals and thus provides error-free value exchange. Lacking a specification, implementors should verify this behavior is true in their systems.

NaN Values

IEEE Standard for Floating-Point Arithmetic (IEEE 754) defines a special set of values called `NaN` ("Not a Number") [5, §6.2]. That standard defines two types of NaNs—signaling and quiet. Both types of NaN can carry additional information in the value representation, e.g., a code indicating an uninitialized value.

XML maps all NaN values to the single literal "NaN" [1, §3.2.4.1 and §3.5.2.1]. The type of NaN and any additional information contained in the NaN value is not represented in the literal.

Looking back at our Motivation, any computation that depends on propagating NaN type and value cannot be distributed using XML as the exchange mechanism.

Rounding to reflect measurement uncertainty

The solution prescribed above may not sit well with some readers—many of us were taught that the result of a calculation cannot be more precise than the uncertainty of the inputs. For example, if $y = x \div 13$, then for the input $x = 1.000$, we report that $y = 0.077$ to match the uncertainty in the input value. However, for the input $x = 1.0$, we should round the result to reflect that input and report $y = 0.1$. While it is correct to round a final result, the issue presented in the Motivation above involves exchanging intermediate values, not final results of a computation. If this value is part of a computation using `float` data types, then we should exchange $y = 0.0769230769$ (9 significant decimal digits).

Conclusion

This discussion should not be construed as a recommendation that *all* XML literals representing floating point values should contain 9 (or 17) significant decimal digits.

XML is an information exchange mechanism, not a data exchange mechanism. Information is data in context, and the context determines how a value should be represented by a literal. Some floating point elements in an XML document may have values that are not related to any underlying computations, e.g., the information represented is "A thermometer measures that the temperature is 70.1 degrees Farenheit". In such cases, fewer significant digits are needed. However, if the value is part of a distributed computation as described above, then a full precision literal representation is necessary.

It is incumbent on the designer of an XML schema to provide implementation guidance for representing floating point values, through schema annotations or other means, and then the implementor must follow that guidance.

References

[1] XML Schema part 2: Datatypes. W3C Recommendation Second Edition, W3C, October 2004. URL: https://www.w3.org/TR/2004/REC-xmlschema-2-20041028/datatypes.html [cited 11 Feb 2022].

[2] Information technology – generic applications of ASN.1: Fast infoset. ITU-T Recommendation ITU-T X.891, ITU, May 2005. URL: https://handle.itu.int/11.1002/1000/8491 [cited 11 Feb 2022].

[3] Efficient XML Interchange Working Group. Efficient XML interchange (EXI) primer. W3C working draft, April 2014. URL: https://www.w3.org/TR/exi-primer/ [cited 12 June 2020].

[4] Java. java.lang Class Double [online]. 2022. URL: https://docs.oracle.com/javase/7/docs/api/java/lang/Double.html#toString(double) [cited 17 Feb 2022].

[5] Microprocessor Standards Committee. IEEE standard for floating-point arithmetic. IEEE Std 754-2019 (Revision of IEEE 754-2008), IEEE Computer Society, New York, NY, USA, 2019. doi:10.1109/IEEESTD.2019.8766229.

Contact Us

Software Engineering Institute
4500 Fifth Avenue, Pittsburgh, PA 15213-2612

Phone: 412/268.5800 | 888.201.4479
Web: www.sei.cmu.edu
Email: info@sei.cmu.edu

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The view, opinions, and/or findings contained in this material are those of the author(s) and should not be construed as an official Government position, policy, or decision, unless designated by other documentation.

DM22-0231