Carnegie Mellon University
Software Engineering Institute

TECHNICAL MEMORANDUM

Date: 10 Feb. 2022
To: File
From: Reed Little and John Klein
Subject: Using XML to Exchange Floating Point Data

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 those produced by executing the computation in a single process, unless care is taken to preserve precision in the XML literals exchanged.

Introduction

This memo 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 showing how to solve the problem using the features of several programming languages. An appendix provides examples that illustrate the issues.

Problem Statement and Notation

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

L_s is an XML literal representation of M_s

L_e is an XML literal exchanged from the source process to the destination process. L_e may have fewer significant digits than L_s, i.e., $L_e \approx L_s$

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:

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

Background

Floating Point Arithmetic

Most computers store floating point data using formats defined by the IEEE Standard for Floating-Point Arithmetic (IEEE 754) [5], which represents a value using a base-2 mantissa and a base-2 exponent. The mantissa includes a fraction value. Even integer values will have a floating point representation with a mantissa that has a fractional value. For example, the base-2 integer value 101_2 is represented as $1.01_2 \times 10_2^{10_2}$.

XML Data Types

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

XML is an *information exchange* mechanism, not a *data exchange* mechanism. XML exchanges literals that represent values (L_s and 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 is not relevant. However, for floating point data types, *data* is generally represented using base-2 fractions, while the *information exchange* uses base-10 representations and the implementor of an information exchange must decide how many significant base-10 digits (significant decimals) to use in the literal representation.

In the problem formulation above, we distinguish L_s (the literal representing the source value) from L_e (the literal exchanged from source to destination). There are established practices in science and engineering which exchange a literal with fewer significant decimals than the full precision of the underlying value—this is typically done so that the precision of the literal reflects the uncertainty in the measurement or calculation that produced the value[1].

[1] See, for example, https://en.wikipedia.org/wiki/Significant_figures#Writing_uncertainty_and_implied_uncertainty

Lexical Space-to-Value Space Transformation Errors

As noted above, the values of a floating point data type are represented as a base-2 mantissa with a fractional part and a base-2 exponent, even if the value has no fractional part. Within the precision of a floating point data type, some base-10 fractions cannot be exactly representable in base-2, and *vice versa*. This can introduce an error in the transformation from the lexical representation to the value representation. For example, the XML float literal 123456789 transforms to an in-memory representation of 0x4ceb79a3, and there is an error (= 3) in this transformation (i.e., the in-memory representation 0x4ceb79a3 converts to the literal 123456792).

This error affects the transformation from literal to memory to literal:

$L_1 \rightarrow M \rightarrow L_2.\ L_2 \stackrel{?}{=} L_1$

This is a different problem than the one defined above in the Problem Statement. This problem is well-known among practitioners (e.g., "Don't use floating point data types to represent currency values in financial calculations.") and we mention it here simply to distinguish it from the XML data exchange problem that we are considering.

While this error source may be important for an application, if the XML float literal 123456789 is exchanged, it will produce the same in-memory value in the source and the destination processes (i.e., the value is exchanged with no error).

XML Encoding

XML documents can be encoded in several ways. The text encoding 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 value represented by the exchanged literal is preserved by all of these encoding/decoding technologies, regardless of whether the data type is float, double, or decimal, provided that the value is within the information space of the data type. End-to-end information error is produced when rounding or truncation of the exchanged value makes it different from the value of source information (within the value space of the data type being used), not due to the encoding/decoding process.

How to use XML for data exchange of 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 data 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.

XML Schema's *facet* mechanism [1, §2.4] can constrain the allowable values of an element, however the available facets of XML's floating point data types are unable to restrict lexical representations to have a minimum number of significant decimal digits. XML application programming interfaces (APIs) also 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 [2] treat lexical literals as strings.

The mechanism to convert from a base-2 floating point representation to a base-10 literal 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 printf[3]. Many other modern programming languages provide a similar function. The requirements stated above can be satisfied using `printf` by using a format string with the E conversion specifier, which allows specification of significant digits, e.g., 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 digits.

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

NaN Values

IEEE Standard for Floating-Point Arithmetic (IEEE 754) [5, §6.2] defines a special set of values called NaN (not a number). That standard defines two types of NaNs—signaling and quiet. Both types of NaN can carry additional information encoded 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 and any additional information encoded 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.

[2] http://www.saxproject.org
[3] https://linux.die.net/man/3/printf

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#float [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.

[6] H. Schmidt. IEEE-754 floating point converter [online]. 2022. URL: https://www.h-schmidt.net/FloatConverter/IEEE754.html [cited 3 Feb 2022].

Appendix: Illustrative Examples

These examples used a web-based tool to perform conversions [6]. The tool uses IEEE 754 Single Precision format.

Example 1: The exchanged value is rounded

- Source:
  M_s = 0x3f9e0610 (Value Space)
  L_s = 1.2345600128173828125 (Lexical Space)
- Exchanged value:
  L_e = 1.2346 (round to 5 significant decimals)
- Destination:
  M_d = 0x3f9e075f (= 1.2345999479293823242187 5) (Value Space)

End-to-end information error:

$E = 1.2345600128173828125 - 1.2345999479293823242187 5 \approx 10^{-4}$

Example 2: Preserve precision of exchanged value

Single float requires 9 significant decimals to represent full binary precision [5, §5.12.2].

- Source (same as previous):
  M_s = 0x3f9e0610 (Value Space)
  L_s = 1.2345600128173828125 (Lexical Space)
- Exchanged value:
  L_e = 1.23456001 (round to 9 significant decimals)
- Destination:
  M_d = 0x3f9e0610 (= M_s) (Value Space)

End-to-end information error = 0.

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.

DM22-0176