Completely Meaningful Binary Encoding

December 5, 2023

Kees-Jan Hermans (kees.jan.hermans@gmail.com)

This paper describes a format of binary encoding of complex data types, that should not be able to break or confuse the decoder because of encoding related inconsistencies ('grammatical mistakes'). The general principle can be summarized as 'every bit is meaningful'.

This paper provides a pseudo code algorithm for the principle's decoder, discusses potential encoding variations, provides reference examples, compares with other encodings and discusses the advantages and disadvantages of the format.

Version 0.0.1 **Unclassified**

Contents

1 Premise

The premise of this treatise is that it must be possible to create a binary encoding format specification such that any random input deterministically and without an error, produces a valid output. The use of this specification must allow for complex data structures in the output of the decoder (for example, as offered by ASN.1 [5] XML [6] or JSON [8]).

In other words, the following should always work:

```
$ cat /dev/random | head -c $ANYNUMBER > /tmp/foo
$ ./decoder < /tmp/foo
$ #... some convoluted, incomprehensible JSON expression on stdout...
```

2 Problem Statement

Binary encodings are often chosen because of their density. However, when applied, they usually contain at least some 'air' (superfluous or useless information). The problem with binary encodings is that this 'air' can also contain encoding mistakes. For example:

- When using TLV [4], they may contain undefined type identifiers.
- When using TLV, they may contain lengths that are illegal. Either because they are zero, which the type does not support, or because they exceed the length of the message size, or the size of the element that contains them.
- For example, in DER [7] encoding it is possible to define lengths that could not possibly exist in our universe.
- For example, in UTF-8, there exist illegal wide character encodings.

Textual formats can convey complex data structures, but are also (very) error prone. For example:

- String escaping may lead to illegal escape sequences or breaking of string enclosing logic.
- Delimiting tokens, such as string braces, brackets or closing tags may be ommitted or over-supplied.
- 8-bit relevant bytes in 7-bits clean messages encoded as 8-bit formats.

etc.

All of these formats are problematic because we have to deal with errors during parsing. Errors in parsing are hardly ever meaningful, since we have no good way to recover from them (but they do introduce a lot of extra code). The most we can usually do, is point out where grammar was broken. Error correcting codes can be provided with any input, but they don't fix 'honest mistakes' - just error(s) in transmission.

Tail end recovery of both 'JSON'- or 'DER'-like encodings should be relatively easy (just wrap up the parsing effort and present the current state of the parser as the result to the caller), but this is not standard. With the content encoding format described in this paper however, tail end recovery is implicit.

3 Design Principles

- Length can never be used in the way that it may be used to under- or overflow the message or message segment size.
- Our atom will be a bit, not a byte, as we cannot have any 'air' in our encoding.
- Complexity of the output must be 'like ASN.1, XML or JSON'. That is to say:
  - Encodes all eight-bit bytes and all lengths of those bytes. And maybe even wide characters.
  - Is complex in that it provides at least for nulls, booleans, strings, integers, floating points, 'hashtable' name-value lists, and arrays. Optionally, the type system is extensible.
- We don't care about being truly efficient. The fact that we're binarily encoding should bring us enough of that. The main goal stands: any decoder input must produce an output.
- We don't care about deterministic reciprocity between encoding and decoding: if the input to the decoder produces a complex data structure, then the complex data structure, when fed to the encoder, doesn't need to produce a copy of the input.
- There exists a 'null' type, which can be given explicitly, or we use it to fill in all sorts of blanks.
- On top, everything is implicitly inside a list. When the implicit list contains one element, it is a scalar. If you want to define a list with one element, you must explicitly define a list.
- At the end of input, however many zero bits as are required can be read. Since all reads are implicitly length delimited ('next byte' in string reading never pops more than nine bits, integers pop 64 bits, etc), this is a one-time affair only. All loops (strings, hashtables, arrays) take the end of input as to mean: jump out of the loop and return.

4 Specification

Explicitly written from the standpoint of the decoder. The encoder will use this specifiction simply in reverse.

- There are the following types (eight, so that they fit exactly into three bits):
  - Implicit NULL
  - Explicit NULL
  - BOOLEAN
  - INTEGER (64 bits signed)
  - FLOAT (of system double size)
  - STRING
  - ARRAY
  - HASHTABLE

- The decoder starts assuming it has to fill a list. If, in the end, this list will contain exactly one element, it will return the element, not the list. In all other cases (empty list, list with more than one element), it will return the list.
- The decoder pops a type triplet and switches according to its value:
  - If it is a null, it will move to pop the next tuple.
  - If it is a boolean, it will pop the next bit to determine its value and move on to the next tuple.
  - If it is an int or a float, it will pop the next 64 bits, cast them to the machine representation, and move on to the next tuple.
  - If it is a string, it will go into a loop determining, per byte, whether or not to read it. It does this by popping a bit and if it is one, it will pop the byte. If the 'continuity bit' is zero, it assumes the string is finished and move on to the next tuple.
  - If it is a list, it does the same 'continuity bit popping' for each list element, and each element recurses into the list-tuple decoding.
  - If it is a hashtable, it pops a string (without type triplet popping) as key, and an element (as in the list), and does this in the same 'continuity bit popping' kind of way as with strings or list elements.
- Should an input end with one to seven zero bits, these should, per the specification above, be rendered as a list of one or more implicit NULL types (one null D-bit and three null type-bits, which can repeat a maximum of two times in seven bits), these will only not be discarded when they are explicit NULL types.

  Note that implicit zeroes, enclosed in other relevant tuples, *are* absorbed as regular nulls.

4.1 Pseudo Code

The main decoding algorithm consumes tuples for as long as there is input, (given by the 'finished' variable which is set to true when the end of input is encountered by the 'read_bits' function) and interprets these as elements of a list. Then it pops any non explicit nulls off of the end of the list and, if the amount of elements in the list is one, gives that element as a result (otherwise, gives the list itself as a result):

```
var finished := false
var result := [ ]
while ! finished :
    var tuple := read_any()
    push result, tuple
while length result > 0 && result [ -1 ] = IMPLICITNULL(0) :
    pop result
if length result = 1 :
    result := result [ 0 ]
result
```

The main tuple consuming function is implemented as follows (default result is null, which stems from the type values of zero and one):

```
fn read_any:
    var type := read_bits(3)
    if type = BOOLEAN(2):
        return read_boolean()
    else if type = INTEGER(3):
        return read_integer()
    else if type = FLOAT(4):
        return read_float))
    else if type = STRING(5):
        return read_string()
    else if type = ARRAY(6):
        return read_array()
    else if type = HASHTABLE(7):
        return read_hashtable()
    else:
        return null
```

Consuming a boolean then simply becomes reading a single bit. Like so:

```
fn read_boolean:
    var value := read_bits(1)
    return value
```

Strings use continuity bits. Every subsequent byte is read only when a single continuity bit is set to true. In pseudo code:

```
fn read_string:
    var string := ""
    while ! finished && read_bits(1):
        string += chr(read_bits(8))
    return string
```

Continuity bits are also used in the consumption of elements of arrays and hashtables. A single bit (which is always returned) determines whether or not the next element is read. Like so:

```
fn read_array:
    var array := [ ]
    while ! finished && read_bits(1):
        var element := read_any()
        push array, element
    return hashtable
```

And:

```
fn read_hashtable:
    var hashtable := { }
    while ! finished && read_bits(1):
        var key := read_string()
        var value := read_any()
        hashtable{key} := value
    return hashtable
```

5 Thoughts on Sub-Encoding Alternatives / Optimizations

5.1 Bigger type space encoding

Currently, by reading three bits, the amount of types possible to encode is limited to eight permutations. This just so happened to satisfy the requirements of encoding JSON. However, it may be that you're having a more extensive need for type encoding. In that case you could extend the type identifyer sequence with one bit, giving you sixteen possible types. Add one more bit and you'll have 32, etc.

If then, however, you're not in need of all sixteen (or 32, or 64, or ...) types, you can either use a modulus of the type space available (however, this will also make the encoder/decoder stages less deterministically linked - since you can have, in some cases, more than one discriminant for a type), or you can appreciate types as binary trees. This will take more processing, but it may also yield a more specific encoding, that won't necessarily require modular arithmetic. For example:

```
                                 /- 0 implicit null
                   /- 0 null ----
                  /              \- 1 explicit null
     /- 0 scalar -
     |            \              /- 0 fixedlength --- - 00 bool (1)
     |             |             |                    - 01 int32 (32)
     |             |             |                    - 10 int64 (64)
     |             |             /                    - 11 float (64)
     |             \- 1 non-null
     |                           \- 1 arbitrarylength - 0  bigint
     |                                                - 10 string (7 bits)
     |                                                - 11 string (wide char)
     /
type               /- 0 array
     \- 1 compound
                   \- 1 hashtable
```

Giving you eleven possible types. Note that this system also has drawbacks: six bits to encode a boolean value, for example - I think that's a lot.

5.2 String Encoding

5.2.1 Overhead

Currently, the implementation pulls a single bit to see if it would need to read another single byte. This creates a string encoding overhead of 12.5% in the limit (long strings), which may be seen as excessive.

It is relatively easy however, to create a string encoding scheme that is more frugal. For example: pull three bits to determine how many of the following 0-7 bytes should be read. This does not exceed the amount of bytes required by the decoder to keep in a buffer (which has to be able to hold an int or double). Overhead then drops to lower percentages in the limit (in this case 3 bits out of 56, or 5.4%) - go to ten bits for a segment length, read up to a kilobyte, and the overhead drops to 0.12%.

In pseudo code:

```
fn read_string:
    var string := ""
    while ! finished:
        var sectionlength := read_bits(3)
        if sectionlength = 0:
            return string
        for i in 1..sectionlength:
            string += chr(read_bits(8))
    return string
```

Note that, in the above code, any positive non null integer can replace the '3', while retaining a valid string decoding algorithm. *Eg* for kilobyte blocks, replace '3' with '10'.

5.2.2 Unicode and UTF-8 Encoding

Because Unicode [1] UTF-8 [2] encoding, which is required for popular outputs such as JSON and XML, has invalid byte encoding sequences (for example, a 110xxxxx initial byte *not* followed by a 10xxxxxx byte), it cannot be used as-is against the decoder (which should be error free against random inputs). The following options are considered:

- Be oblivious to these types of encodings, which means that JSON and XML **must** assume that strings are valid byte-for-byte as presented by the decoder. The accompanying encoder simply adds bytes as it encounters them.

  From a principle standpoint: purely random inputs to the decoder may now fail at the JSON/XML level because they may contain / it is very likely that they will contain invalid UTF-8 sequences. The question is: do you care about this type of invalidity?

- Create a special type in the encoding, a 'wide-character string' type, either as an additional type or as a replacement of the string type, that re-encodes wide characters in a way that cannot have potential interpretation errors. Added bonus: if the 'wide string' type were to be made additionally to the 'regular string' type, the now 'non-wide' characters can all be encoded as seven bits, nixing the string encoding overhead discussion in the previous chapter.

- Wide characters can either be encoded using the full width (20-21 bits in case of Unicode), or they can use their own continuity bits over multiple segments ('this character is wide', read eight bits, 'this character has another segment', read six more bits, etc).

The second option is probably the most desirable. However, it requires:

- An potential extension of the type system with one extra type ('wide character string'), or the assumption that all strings are wide character encoded (which may be costly in terms of overhead).
- A scheme that can encode wide characters without errors.
- An obligation on genuine encoders, to use said encoding when encountering character values > 127.

Whatever the end result, it comes down to the encoder and decoder doing the same thing, and the user being aware of the encoding support required.

5.2.3 UTF-8 'Clone' Encoding

This could work as follows (limited to 2^{20} characters (Unicode has slightly more, but these are currently unused / unlikely to be used soon)); From the POV of the decoder:

- Pop one bit: must I read another character? If no: stop reading the string, if yes:
- Pop one bit: is this character 'Unicode'?
- If it is not Unicode, pop seven pits. These are your ASCII character. So for ASCII characters, this scheme is 'lossless' wrt the original string encoding proposal.
- If it is Unicode, pop eight bits. These will form the least relevant byte of the wide character. Pop a continuity bit. If true, pop another eight bits. Add these to the left of the first byte. Pop a last possible continuity bit. If true, pop the last four bits. These form the most relevant bits of the wide character.

In pseudo code:

```
fn read_string:
    var string := ""
    while ! finished && read_bits(1):
        var unicode := read_bits(1)
        if unicode:

            var highest := 0
            var middle := 0
            var lowest := read_bits(8)
            if read_bits(1):
                middle := read_bits(8)
                if read_bits(1):
                    highest := read_bits(4)
            string += w_chr(lowest | (middle << 8) | (highest << 16))

        else:
            string += chr(read_bits(7))
    return string
```

This will implement (most of) Unicode, yet be as frugal with overhead as ASCII.

5.3 Integer Encoding

Currently, integers are encoded as-is (the system's memory representation of a 64 bit signed integer). It should be possible to create an integer encoding based on continuity bits per 8-bit section. It is assumed that most integers leave the most relevant parts of their encoding zero. In this case, this will make integer compression perform better than DER's, and will be more portable across systems. For example as follows (assuming a 32 bit signed integer):

- Read one bit; this bit is the sign of the number. One means minus.
- Read a continuity bit. If one, read a byte. This byte will be the least relevant byte of the quad.
- Perform this step a maximum of three more times. Insert the bytes from least to most relevant positions in the quad. Note that the last time, there will be no 'non-continuity bit' (because we run out of internal integer representation bits anyway), and the only seven bits will be read (because the sign will be part of the 32 bits being transcoded).

Note that the encoding solution above allows for an interpretation of 'minus zero' by the decoder, which should not be problematic and can be made the subject of any convention ('is zero', or 'is INT_MIN').

6 Comparisons

6.1 DER Comparison - Complex Data Structures

A comparison with DER encoding was made. This requires a bit of introduction: DER and completely meaningful encoding are not comparable. DER formats are only slightly self-descriptive and have to have their structure communicated out-of-band. DER knows no key/value pairs, or hashtable, data structure. On the other hand: completely meaningful encoding, in its naive implementation (which was used), knows only seven data types. Some compromises were made: hashtables, for example, were encoded in DER as arbitrary length sequences of sequences with length two (for key and value - this is common with certificates, for example).

6.1.1 Method

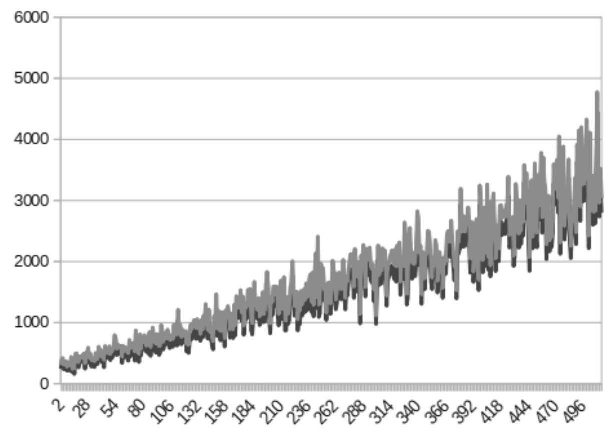
The meaningful encoder tool and a separately written tool, json2der, were each used to create an encoding for random JSON with increasing complexity.

On the X-axis is a measure of complexity of the randomly generated JSON. This complexity pertains to the (bias given to the random) number of traversals of the JSON data structure, and the depth of traversal of the JSON data structure, and the length of the strings of hashtable-keys generated. Source code of the JSON randomizer tool here: [9].

On the Y-axis is the size of the generated binary encoding, averaged out over 16 attempts to generate a structure, per level of complexity.

The light grey represents DER, the dark grey represents the completely meaningful encoding.

6.1.2 Outcome



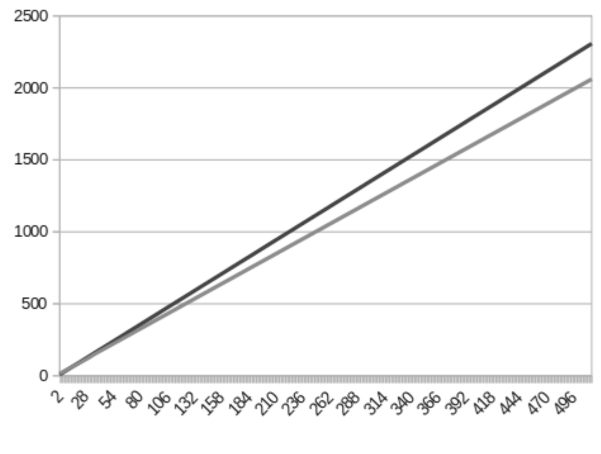
One can see that completely meaningful encoding neatly trails DER, and even is slightly more compressed (average overhead of DER to meaningful encoding in this range is 13%).

6.2 DER Comparison - String Encoding

6.2.1 Method

A JSON file was created, with a single key/value pair, where the value is a string of increasing length (value of the X-axis times four). The JSON was then fed to both the DER encoder and the meaningful encoder. The resulting sizes of the binary were then compared.

6.2.2 Outcome



It can be clearly seen that the DER encoding of strings is more frugal than meaningful encoding (as expected). This should tend to 12.5% (as can be seen from the graph).

6.3 DER Comparison - Conclusion

As expected, DER performs better at leaf (string) encoding while meaningful encoding performs better at structural encoding. The overhead percentages wrt one another seem to cancel each other out statistically. Since this is so, a further expectation is that, with string and integer encoding optimization as proposed in this paper implemented, this balance could made more favourable of meaningful encoding.

7 Examples

Binary data representations can be given in two formats, bit-wise and byte-wise. Bit-wise data is preceeded, per section, by a 'B:', while byte-wise data is preceeded, per section, by a 'H:' (and is denoted in hexadecimal tuplets).

7.1 Example 001

The following input:

```
B: 0 0 0 0  0 0 0 0
```

Produces the following JSON:

```
[ ]
```

Three implicit nulls are read from the input, which are all discarded. Since the implicit top-list now contains zero elements, it is presented as the result of the decoding.

7.2 Example 002

The following input:

```
B: 0 0 1 0  0 0 0 0
```

Produces the following JSON:

```
null
```

An explicit null is read, followed by the read of two implicit nulls. The implicit nulls are discarded (because they are in the tail). Since the implicit top-list only contains one element, it is presented as a single null, not encapsulated in an array.

7.3 Example 003

The following input:

```
B: 0 0 1 0  0 1 0 0
```

Produces the following JSON:

```
[ null, null ]
```

Here, two explicit (un-erasable) nulls are encoded in the implicit top-list, followed by two zero bits. The decoder reads these bits (and one more, which will also be virtually zero), and interprets that as an implicit null. The implicit null is then discarded.

7.4 Example 004

The following input:

```
B: 1 1 0 1  0 0 0 0
```

Produces the following JSON:

```
[ null ]
```

In this example, the type number six (1 1 0) denotes an explicit array. It is followed by a one-byte, which denotes the fact that the array contains at least one more element. The subsequent zeroes denote the filling.

Note that the encoding could have also used an explicit null here (making for 1 1 0 1 0 0 1 0).

7.5 Example 005

The following input:

```
B: 1 1 0 1  0 0 1 1   0 0 1 0  0 0 0 0
```

Read from left to right:

- 'I have a list' (3 bits).
- 'The list has one more element' (1 bit).
- 'I have an explicit null' (3 bits).
- 'The list has one more element' (1 bit).
- 'I have an explicit null' (3 bits).
- 'The list has no more elements' (1 bit).
- .. followed by zero bits which may be interpreted as implicit nulls, but which will be discarded.

Produces the following JSON:

```
[ null, null ]
```

Note that this is the exact same output as given in example 3, but now made explicit.

7.6 Example 006

The following JSON input:

```
[
  "foo", "bar", { "foo" : "bar" }, [], [ [ ] ]
]
```

When encoded, leads to the following binary:

```
00000000  db 66 b7 db db 62 b0 dc  9f b3 5b ed eb 62 b0 dc  |.f...b....[..b..|
00000010  8e 77 00                                          |.w.|
00000013
```

Which, in turn, when fed to the decoder, leads to this JSON:

```
["foo","bar",{"foo":"bar"},[],[[]]]
```

For context: the same structure, more or less (DER has no concept of name/-value pair lists or hashtables), encoded as DER (10 more bytes required):

```
30 1c 04 03 f  o  o  04  03 b  a  r  30 0a 04 03 f
o  o  04 03 b  a  r  30  00 30 02 30 00
```

8 Assessment

For any further information wrt implementation details / test data, please visit the GitHub repository [10].

8.1 Advantages

- The format can be decoded in a streaming manner: the decoder never has to have more than a small amount of bytes in its buffer, no 'looking back' is ever required.
- Therefore, it is 'auto-limiting': a large value such as a large string will never be read for more than one byte, or one block, beyond the input boundary.
- In spite of that, a complex structure a la JSON/XML/ASN.1 can be encoded.
- In fact, using the extensions concept, *any* structure can be encoded (so long as there are no value intrinsic constraints, such as 'this value is a prime number', or 'this value is three times the previous value'). Even Unicode support is possible.
- While the decoder is simple to code, and cannot fail.
- Therefore, cannot fail mid-decoding either (so long as you have the head of the message, you can receive partial messages and still have a valid encoding).

8.2 Disadvantages

- String encoding carries a overhead penalty (in the naive implementation: 12.5% in the limit).

- Encoding and decoding may not be deterministically linked: the same chaos may not be returned when passed through the decoder and encoder subsequently (the reverse order however, from JSON to binary to JSON, does deterministically link the encoder and the decoder - provided JSON hashtable keys are sorted).
- Although correction of inputs - when not using error correcting codes - is neigh on impossible in all popular formats, they can usually provide some feedback. With this format, feedback to the user ('at this line / token your input went wrong') is impossible.
- (When decoding random inputs:) null bytes in strings are possible, and duplicate keys in key/value structures are a possibility (although JSON leaves this behaviour undefined).

References

[1] Unicode https://home.unicode.org/

[2] UTF-8 Encoding https://www.unicode.org/versions/Unicode15.1.0/ch02.pdf#G13708

[3] Character Frequency by Unicode Version http://unicode.org/L2/L2009/09180-char-frequency.pdf

[4] Type-Length-Value Encoding https://en.wikipedia.org/wiki/Type%E2%80%93length%E2%80%93value

[5] ASN.1 https://www.itu.int/en/ITU-T/asn1/Pages/introduction.aspx as well as Rec. ITU-T X.680

[6] XML https://www.w3.org/XML/

[7] A Layman's Guide to a Subset of ASN.1, BER, and DER https://luca.ntop.org/Teaching/Appunti/asn1.html

[8] JSON, JavaScript Object Notation https://www.json.org/

[9] Perl JSON parametrizable randomizer script https://github.com/kjhermans/meaningful/blob/main/randomizer.pl

[10] Meaningful encoding / Sarthaka project GitHub https://github.com/kjhermans/meaningful/