APP area
Internet-Draft
Intended status: Informational
Expires: August 29, 2013

C. Bormann
Universitaet Bremen TZI

February 25, 2013

The BinaryPack1pre2 JSON-like representation format
draft-bormann-apparea-bpack-01

Abstract

JSON (RFC 4627) is an extremely successful format for the representation of structured information, supporting Boolean values, numbers, strings, arrays, and tables. Recently, a number of applications have started to look for binary representation formats that solve a similar problem. In particular, constrained node networks can benefit from such a binary representation format.

A very successful binary representation that is otherwise comparable to JSON is MessagePack. Recently, a number of implementations have modified or extended MessagePack such that it allows for distinguishing UTF-8 strings from binary data. Further discussion on the MessagePack repository has resulted in proposals how to integrate such an addition back into the MessagePack community.

This draft, as an independent effort, documents one such format, tentatively calling it BinaryPack1pre2 while the MessagePack extension proposals make their way through the MessagePack community.

The current version -01 of this document is a snapshot that demonstrates a general direction. The details may change in future versions based on the development of the MessagePack specification.

This Internet-Draft will expire on August 29, 2013.

Table of Contents

1. Introduction

(To be written - for now please see the Abstract.)

A description of the MessagePack binary representation format can be found in [msgpack]. A recent proposal for an update, still under discussion, is in [msgpack-update].

One of the early proposals implementing separate types for byte strings and UTF-8 strings was called BinaryPack. An implementation of BinaryPack is available in [binarypack]. (An extension similar in spirit, but different in details, was made for the [msgpack-js] and [msgpack-js-browser] projects.)

1.1. Objectives

(TBD, but this is a rough first approach:)

The objectives of the present specification, roughly in decreasing order of importance, are:

- Representing a reasonable set of basic data types and structures using binary encoding. "Reasonable" here is largely influenced by the capabilities of JSON, with the single addition of adding raw byte strings. The structures supported are limited to trees; no loops or lattice-style graphs.

- Being implementable in a very small amount of code, thus being applicable to constrained nodes [I-D.ietf-lwig-terminology], even of class 1. (Complexity goal.) As a corollary: Being close to contemporary machine representations of data (e.g., not requiring binary-to-decimal conversion).

- Being applicable to schema-less use. For schema-informed binary encoding, a number of approaches are already available in the IETF, including XDR [RFC4506]. (However, schema-informed use of the present specification, such as for a marshaling scheme for an RPC IDL, is not at all excluded. Any IDL for this is out of scope for this specification.)

- Being reasonably compact. "Reasonable" here is bounded by JSON as an upper bound in size, and by implementation complexity maintaining a lower bound. The use of general compression schemes violates both of the complexity goals.

- Being reasonably frugal in CPU usage. (The other complexity goal.) This is relevant both for constrained nodes and for potential usage in high-volume applications.

- Supporting a reasonable level of round-tripping with JSON, as long as the data represented are within the capabilities of JSON. Defining a unidirectional mapping towards JSON for all types of data.

1.2. Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

The term "byte" is used in its now customary sense as a synonym for "octet".

All multi-byte integers in this protocol are interpreted in network byte order.

Where arithmetic is used, this specification uses the notation familiar from the programming language C, except that the operator "**" stands for exponentiation.

1.3. Notation

This specification uses a trivial notation for code bytes and the bitfields in them the meaning of which should be mostly obvious. More formally speaking, the meaning of the notation is:

Potential values for the code bytes themselves are expressed by templates that represent 8-bit most-significant-bit-first binary numbers (without any special prefix), where 0 stands for 0, 1 for 1, and variable segments in these code byte templates are indicated by sequences of the same letter such as kkkkkkk or ssss, the length of which indicates the length of the variable segment in bits.

In the notation of values derived from the code bytes, 0b is used as a prefix for expressing binary numbers in most-significant-bit first notation (akin to the use of 0x for most-significant-digit-first hexadecimal numbers in the C programming language). Where the above-mentioned sequences of letters are then referenced in such a binary number in the text, the intention is that the value from these bitfields in the actual code byte be inserted.

Example: The code byte template

```
101nssss
```

stands for a byte that starts (most-significant-bit-first) with the bits 1, 0, and 1, and continues with five variable bits, the first of

which is referenced as "n" and the next four are referenced as "ssss". Based on this code byte template, a reference to

0b0ssss000

means a binary number composed from a zero bit, the four bits that are in the "ssss" field (for 101nssss, the four least significant bits) in the actual byte encountered, kept in the same order, and three more zero bits.

Also, 0xhh stands for the hexadecimal value hh, and 1B, 2B, 4B, 8B, nB stand for 1, 2, 4, 8, or n bytes of data following; (1B) etc. stand for the numerical value of these bytes as an integer interpreted in network byte order; nD stands for n data objects, each in turn in BinaryPack1pre2 representation format.

2. The BinaryPack1pre2 Representation Format

2.1. Data Types

The BinaryPack1pre2 representation format is able to represent the following data types:

- Integers (represented in signed and unsigned forms)
- Floating point values (in IEEE 754 32-bit and 64-bit forms)
- special values nil, false, true
- opaque ("raw") byte strings, or "binary strings"
- UTF-8 strings
- arrays, which can contain any combination of data types
- tables (often called maps, hashes, dictionaries; objects in JSON), which contain pairs, key and value, which may in turn be of any data type

This list is mostly faithful to JSON [RFC4627], which however does not distinguish integer from floating point number types. Based on recent discussions on the use of binary representation formats, the present specification distinguishes UTF-8 strings from opaque binary strings. (Interestingly, such a separation was already done in the binaryjs implementation of a "95 % MessagePack" format [binarypack], so the author of the present specification started out by just lazily copying that; more recent input taken from the msgpack developers [msgpack-update] is the technical basis for the current proposal.)

2.2. Integers

BinaryPack1pre2 provides a number of representations for integer values, assuming that these occur often. The encoder is free to choose any of these representations that is able to represent the desired value.

Bits	Value	Description
0nnnnnnn	0bnnnnnnn	Positive Integer (0..127)
111nnnnn	0bnnnnn - 32	Negative Integer (-32..-1)
0xcc 1B	1B as uint	Unsigned Integer
0xcd 2B	2B as uint	Unsigned Integer
0xce 4B	4B as uint	Unsigned Integer
0xcf 8B	8B as uint	Unsigned Integer
0xd0 1B	1B as sint	Signed Integer
0xd1 2B	2B as sint	Signed Integer
0xd2 4B	4B as sint	Signed Integer
0xd3 8B	8B as sint	Signed Integer

2.3. Floating Point Values

BinaryPack1pre2 provides 32-bit and 64-bit IEEE 754 values. (See also Appendix B.2.)

Bits	Value	Description
0xca 4B	4B as 32-bit IEEE 754	Float
0xcb 8B	8B as 64-bit IEEE 754	Double

2.4. Special Values

Similar to the special literals "false null true" in JSON, BinaryPack1pre2 provides three special values:

Bits	Value	Description
0xc0	nil	null, nothing
0xc2	false	Boolean false
0xc3	true	Boolean true

2.5. Binary: Opaque Byte Strings

(Note that the specific codepoint allocations in this section are very much up for discussion. It can also be argued that we should be spending some of the remaining reserved codepoints for short byte strings.)

Bits	Value	Description
0xd5 1B nB	n = (1B)	byte string (0..(2**8-1) bytes)
0xd6 2B nB	n = (2B)	byte string (0..(2**16-1) bytes)
0xd7 4B nB	n = (4B)	byte string (0..(2**32-1) bytes)

2.6. UTF-8 Strings

Bits	Value	Description
101nnnnn nB	n = 0bnnnnn	Short UTF-8 string (0..31 bytes)
0xd9 1B nB	n = (1B)	UTF-8 string (0..(2**8-1) bytes)
0xda 2B nB	n = (2B)	UTF-8 string (0..(2**16-1) bytes)
0xdb 4B nB	n = (4B)	UTF-8 string (0..(2**32-1) bytes)

The strings transported MUST be UTF-8 strings [RFC3629]. (The general assumption is that these UTF-8 strings are in Network Unicode form [RFC5198], see Appendix A for some more discussion.)

2.7. Arrays

Bits	Value	Description
1001nnnn nD	n = 0bnnnn	Short array (0..15 data elements)
0xdc 2B nD	n = (2B)	array (0..(2**16-1) data elements)
0xdd 4B nD	n = (4B)	array (0..(2**32-1) data elements)

2.8. Tables

Bits	Value	Description
1000nnnn nD	n = 2 * 0bnnnn	Short table (0..15 data pairs)
0xde 2B nD	n = 2 * (2B)	table (0..(2**16-1) data pairs)
0xdf 4B nD	n = 2 * (4B)	table (0..(2**32-1) data pairs)

The sequence of n elements is a sequence of pairs of data objects, each pair represented as one data object representing the key followed by the data object representing its associated value.

3. Discussion

This draft tries to be faithful to the successful MessagePack [msgpack] format, including an recent extension proposal that enables the distinction between opaque binary byte strings and UTF-8 byte strings [msgpack-update].

Little analysis has been made whether a slightly different bit allocation (e.g., using up fewer of the code combination for single-byte integers) would be advantageous. However, the gains from a different allocation are likely to be limited except for pathological cases. (The main benefit achievable may be to have more codepoints reserved for future expansion.)

A short floating point (e.g., based on the 16-bit IEEE 754 floating point value) might be a useful additional representation format. Adding decimal floating point values probably is not so useful, except where high fidelity to JSON is desired.

Some additional data types might be useful for some protocols, e.g. UUIDs [RFC4122], date/time. See also Appendix B. This would further increase the distance from JSON that BinaryPack1pre2 creates by distinguishing opaque and UTF-8 strings.

3.1. JSON roundtripping

BinaryPack1pre2 enables mostly lossless translation to JSON. JSON [RFC4627]. JSON roundtripping, however, is not necessarily the primary design goal of BinaryPack1pre2, but it is a consideration.

In the translation of BinaryPack1pre2 to JSON, opaque byte strings SHOULD be converted to equivalent base64url [RFC4648] UTF-8 strings. Without a schema, it is hard to do the inverse consistently, as base64url encoded byte strings are not specially marked up in JSON.

When translating BinaryPack1pre2 floating point values to JSON, the usual problem of converting binary fractions to decimal representation arises. In the other direction, the choice of a floating point format may be hard to do properly. Clearly, any number that can be transformed from a 64-bit IEEE 754 number to a 32-bit IEEE 754 number without loss of information can be represented as the latter. Without schema information, it may be hard to find other cases where the precision maybe is not that important.

4. IANA Considerations

Once this has received some discussion, we will understand how exactly to register Internet media types for this.

The potential extension mechanisms discussed in Appendix B may need an IANA registry.

5. Security Considerations

(Nothing but generic warnings about correctly implementing protocol encoders/decoders so far; this section will certainly grow as additional security considerations become known.)

6. Acknowledgements

MessagePack was developed and promoted by Sadayuki Furuhashi ("frsyuki").

BinaryPack is a minor derivation of MessagePack that was developed by Eric Zhang for the binaryjs project. A similar, but different extension was made by Tim Caswell for his [msgpack-js] and [msgpack-js-browser] projects.

The author of the present specification deserves absolutely no credits whatsoever for any of this.

7. References

7.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006.

[RFC5198] Klensin, J. and M. Padlipsky, "Unicode Format for Network Interchange", RFC 5198, March 2008.

[RFC5905] Mills, D., Martin, J., Burbank, J., and W. Kasch, "Network Time Protocol Version 4: Protocol and Algorithms Specification", RFC 5905, June 2010.

7.2. Informative References

[I-D.ietf-lwig-terminology] Bormann, C. and M. Ersue, "Terminology for Constrained Node Networks", draft-ietf-lwig-terminology-00 (work in progress), February 2013.

[N4246R2] Lunde, K., "Stabilizing CJK Compatibility Ideographs through the use of Standardized Variants", ISO/IEC JTC1/SC2/WG2 N4246R2, March 2012, <ftp://std.dkuug.dk/JTC1/sc2/wg2/docs/n4246.pdf>.

[RFC3339] Klyne, G., Ed. and C. Newman, "Date and Time on the Internet: Timestamps", RFC 3339, July 2002.

[RFC4122] Leach, P., Mealling, M., and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace", RFC 4122, July 2005.

[RFC4506] Eisler, M., "XDR: External Data Representation Standard", STD 67, RFC 4506, May 2006.

[RFC4627] Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)", RFC 4627, July 2006.

[binarypack] Zhang, E., "BinaryPack for Javascript browsers", 2012, <https://github.com/binaryjs/js-binarypack>.

[msgpack-js-browser] Caswell, T., "msgpack for the browser", 2012, <https://github.com/creationix/msgpack-js-browser>.

[msgpack-js] Caswell, T., "msgpack for node", 2012, <https://github.com/creationix/msgpack-js>.

[msgpack-update] Furuhashi, S., "msgpack-update-proposal1.md", February 2012, <https://gist.github.com/frsyuki/5022569>.

[msgpack] Ohta, K. and S. Colebourne, "MessagePack format specification", 2011, <http://wiki.msgpack.org/display/MSGPACK/Format+specification>.

Appendix A. Unicode Considerations

(TBD. Some initial guidelines at [msgpack-update]. This section should make clear that:)

- At the BinaryPack1pre2 encoding/decoding layer, implementations are never concerned about Unicode normalization.

- Internet usage of Unicode is governed by [RFC5198]. The present specification will not try to second-guess the evolution of this standards-track document.

- [RFC5198] states that >>Before transmission, all character sequences SHOULD be normalized according to Unicode normalization form "NFC"<<. There may be some need to interpret this "SHOULD" in the context of the present specification, as follows.

- There is a very strong expectation that applications making use of BinaryPack1pre2 will lean towards using Unicode in NFC form, as opposed to NFD. In other words, receivers may expect data in the maximally composed form, as opposed to decomposed form.

- The Normalization component of NFC may create problems in some applications (e.g., see [N4246R2]). Before this is repaired in some future version of Unicode, there is no expectation that all applications generating BinaryPack1pre2 always perform the canonical normalization where information loss would result.

- There is a strong expectation that BinaryPack1pre2 receivers be resilient to the small variations in Unicode usage discussed here.

Appendix B. Potential future work

Two data types have been discussed for addition to BinaryPack1pre2.

B.1. Reserved Code Points

As of today, the following code points are reserved and could be used for further extension, if required:

0xc1, 0xc4..0xc9, 0xd4, 0xd8

B.2. 16-bit floating point

16-bit floating points have become popular recently. BinaryPack1pre2 could enable the efficient transport of small floating point numbers by adding a Half-precision floating point representation:

Bits	Value	Description
0xc9 2B	2B as 16-bit IEEE 754	Half
0xca 4B	4B as 32-bit IEEE 754	Float
0xcb 8B	8B as 64-bit IEEE 754	Double

B.3. DateTime

Many applications need the transport of Date/Time information. Some need micro- or nanosecond resolution, some are more concerned about significant range.

In the IETF, both NTP timestamps [RFC5905] and ISO8601 dates [RFC3339] are popular. The former probably require short and long versions to accommodate the different requirements in precision and range. As a start, a 32.32 and a 64.64 NTP timestamp could be defined. ISO8601 dates would need a length indicator and could

therefore look close to the string8 form in BinaryPack1pre2. It is worth limiting the set of choices based on some more input on what is actually required.

B.4. Prefixing extensions

As the small number of remaining code points could be used up quickly, some additions might preferably be expressed by a prefixing scheme. E.g., if 0xc1 is picked for prefixing, the format

```
0xc1 0xnn 0xd5 0x08 ...
```

could be used for designating an 8-byte binary string (0xd5 0x08 ...) as e.g. a date/time in 32.32 NTP timestamp format; the same value for 0xnn could also be followed by a 16-byte binary string for a full 64.64 NTP timestamp and maybe even followed by an UTF-8 string for GeneralizedTime _or_ an ISO8601 time, depending on which of these formats are desirable. Implementations unaware of the semantics for a specific value of 0xnn could still process the information as a binary or UTF-8 string.

The number of extensions defined this way should be kept very small, not only to preserve coding efficiency by making do with the single-byte discriminator. The values for 0xnn would then be maintained in an IANA registry, with a suitably careful allocation policy. This needs further discussion.

B.5. Extension Points

More generally, evolution of a format always raises considerations about compatibility. There are two directions of compatibility: - Old data/old senders to new receivers (forward compatibility) and - new data/new senders to old receivers (backward compatibility).

Further extension of the msgpack format currently always loses backward compatibility, as there is no way for an older implementation to find out the length that is consumed by a construct that uses a new codepoint. In addition to a prefixing mechanism, the BinaryPack1pre2 format could include deliberate extension points that would at least allow an old receiver to decode future versions of the BinaryPack1pre2 format without losing synchronization in the byte stream, while possibly having to treat some of the information as opaque.

Author's Address

Carsten Bormann
Universitaet Bremen TZI
Postfach 330440
Bremen D-28359
Germany

Phone: +49-421-218-63921
Email: cabo@tzi.org