Network Working Group
Internet-Draft
Intended status: Experimental
Expires: April 15, 2014

JP. Makela
October 12, 2013

Ruoska Encoding
draft-ruoska-encoding-06

Abstract

This document describes hierarchically structured binary encoding format called Ruoska Encoding (later RSK). The main design goals are minimal resource usage, well defined structure with good selection of widely known data types, and still extendable for future usage.

The main benefit when compared to non binary hierarchically structured formats like XML is simplicity and minimal resource demands. Even basic XML parsing is time and memory consuming operation.

When compared to other binary formats like BER encoding of ASN.1 the main benefit is simplicity. ASN.1 with many different encodings is complex and even simple implementation needs a lot of effort. RSK is also more efficient than BER.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on April 15, 2014.

Copyright Notice

Copyright (c) 2013 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Document Structure 3
  1.1. Endianness 4
  1.2. String Encoding 4
2. Frame Definitions 5
  2.1. Leading Byte 5
  2.2. Meta Frames 6
    2.2.1. Null Frame 6
    2.2.2. Begin Frame 6
    2.2.3. End Frame 7
    2.2.4. Array Frame 7
  2.3. Data Frames 8
    2.3.1. Boolean Frame 9
    2.3.2. Integer Frames 9
    2.3.3. Float Frames 10
    2.3.4. String Frame 10
    2.3.5. Binary Frame 11
    2.3.6. DateTime Frames 12
    2.3.7. NTP Short Frame 13
    2.3.8. NTP Timestamp Frame 13
    2.3.9. NTP Date Frame 14
    2.3.10. RSK Date Frame 14
  2.4. Extended Frames 15
3. Identifiers 16
  3.1. Identifier Types in Leading Byte 16
  3.2. Null Identifier 16
  3.3. Integer Identifiers 16
  3.4. String Identifier 17
4. Frame Type Table 18
5. Implementation Notes 20
6. Security Considerations 21
7. IANA Considerations 22
8. Normative References 23
Author's Address 24

1. Document Structure

The principal entity of RSK document is frame. Two main classes of frames exist. Meta Frames to define structure and Data Frames to hold actual payload data.

All frames start with Leading Byte which defines frame type and some type depended instructions. Some meta frames and all data frames can be tagged with an identifier. Identifier type is defined in Leading Byte. If identifier exists it is placed right after the Leading Byte. In Data Frames payload comes after identifier. Meta Frames may also have payload or special fields or both. Data type of the payload is defined by frame type and type depended instructions. All frame types are explained in Section 2.

RSK document is structured as finite tree. The tree is rooted to Begin Frame. After the rooting Begin Frame follows data frames as leafs and Begin - End Frame pairs as branches. Branches may contain data frames as leafs and again Begin - End Frame pairs as sub branches. Null and Array Frames are considered as data frames here.

RSK document always ends with End Frame. Use of End Frame is two fold. It is used to return from branch to parent level and terminate the document. So document must always start with Begin Frame and end with End Frame. Root nesting level 0 must not contain any other than rooting Begin and terminating End Frames. Between root Begin and terminating End Frame is nesting level 1. Nesting level 1 may contain data frames or branches or both.

Example Tree Structure

```
0  1  2  Nesting levels
|  |  |
Begin[id:tractor]                              Begin Frame at level 0
|  String[id:manufacturer, value:Valmet]         Leaf at level 1
|  String[id:model, value:33D]                   Leaf at level 1
|  Begin[id:engine]                              Branch at level 1
|  |  String[id:fuel, value:Diesel]                Leaf at level 2
|  |  UInt8[id:horsepower, value:37]               Leaf at level 2
|  End                                           Ending branch
End                                            Terminating at level 0
```

Figure 1: Tree Structure

1.1. Endianness

Big-endian networking byte order is used. Endianness applies to all integer and floating point numbers. This includes payload of any data frames like Integer, Float, and Timestamp and 16-bits wide integer identifier values and also meta data fields like length of payload. Canonical network byte order is fully described in RFC791, Appendix B.

1.2. String Encoding

All strings are UTF-8 encoded. This applies to string identifiers and payload of String, Date, DateTime and DateTimeMillis Frames.

Implementations using any of frame types above or String Identifier or both must be able to validate UTF-8 encoding. On writing phase UTF-8 encoding violation must be handled as error condition. If UTF-8 encoding fails on reading phase warning must be raised and let user decide to continue reading or not. More information about UTF-8 see RFC 3629 [RFC3629].

2. Frame Definitions

As mentioned earlier the principal entity of RSK is frame. This section explains all frame types and type dependent special instructions in detail.

2.1. Leading Byte

All frames start with Leading Byte. Leading Byte determines frame type and type dependent instructions. The most significant bit is reserved for Extended Frames which may be introduced in later versions. See Section 2.4 for details.

Leading Byte is presented as bit array where left-most bit is the most significant bit. MSB 0 bit numbering scheme is used with two exceptions. Left-most bit is reserved for Extended Frame and thus marked as 'X' for all Leading Byte definitions. Also some bits are marked with 'R' meaning that they are reserved for later use and must not be set in this version.

Leading Byte is followed by frame type dependent fields like Identifier or Payload or both. These fields are presented as labeled byte blocks with possible lengths in bytes, kilobytes like 64k, or gigabytes like 4G.

```
   Leading Byte    Example Field   Example Payload
+-+---------+---+ +-------------+ +---------------+
|X|1|2|3|4|5|6|7| |   4 or 8    | |    0 - 64k    |
+-+---------+---+ +-------------+ +---------------+
 | \       / \ /
 |   \   /    | Frame type dependent instructions bits
 |     |
 |     | Frame type bits
 |
 | Extended frame bit
```

Figure 2: Leading Byte

Example Field: Example of possible frame type dependent byte field. 4 or 8 means that field can be 4 or 8 bytes long. Actual length can be determined by frame type, instruction bits, or some other field.

Example Payload: Example of possible frame type dependent payload field. 0 - 64k means that field can be from 0 to 65535 bytes long. Actual length can be determined by frame type, instruction bits, or some other field.

Frame type dependent instructions bits: These bits determine type dependent instructions. See specific frame type sections for details. For most frame types these are used to define identifier type.

Frame type bits: This bit field determines frame type. Values are defined in Section 4.

Extended frame bit: Extended frame bit. See Section 2.4 for details.

2.2. Meta Frames

Meta Frames define document structure.

2.2.1. Null Frame

Null Frame can be tagged with an identifier but does not contain any payload data.

```
   Leading Byte   Identifier
+-+---------+---+ +---------+
|X|1|2|3|4|5|6|7| | 0 - 256 |
+-+---------+---+ +---------+
   \       / \ /
     \   /    | Id bits
       |
       | Type bits
```

Figure 3: Null Frame

Identifier & Id bits: See Section 3 for details.

Type bits: See Section 4.

2.2.2. Begin Frame

Document and branches start with Begin Frame. Begin Frame may have an identifier. More details about tree structure see Section 1.

```
  Leading Byte     Identifier
+-+---------+---+ +----------+
|X|1|2|3|4|5|6|7| | 0 - 256  |
+-+---------+---+ +----------+
   \       / \ /
     \   /    | Id bits
       |
       | Type bits
```

Figure 4: Begin Frame

Identifier & Id bits: See Section 3 for details.

Type bits: See Section 4.

2.2.3. End Frame

End Frame is used to return from branch to parent level in tree structure and also used to terminate a document. More details about tree structure see Section 1.

```
  Leading Byte
+-+---------+---+
|X|1|2|3|4|5|R|R|
+-+---------+---+
   \       /
     \   /
       |
       | Type bits
```

Figure 5: End Frame

Type bits: See Section 4.

2.2.4. Array Frame

Array column in Frame Type Table in Section 4 defines frame types which can be enclosed into a array.

Array itself and each item may have identifiers. Array identifier is defined in Leading Byte. All items have identifier of same type and all items are same frame type. Frame and identifier type for all items are defined by CLB (Common Leading Byte).

Array capacity is defined by selecting corresponding Array Frame type. See Section 4 for details.

```
  Leading Byte    Identifier   CLB    Count          Items
+-+--------+---+ +---------+ +---+ +-------+ +-------+ +-------+
|X|1|2|3|4|5|6|7| | 0 - 256 | | 1 | | 1/2/4 | | Item1 | | Item2 | ...
+-+--------+---+ +---------+ +---+ +-------+ +-------+ +-------+
   \       / \ /
     \   /    | Id bits
       |
       | Type bits
```

Figure 6: Array Frame

Identifier & Id bits: Array identifier. See Section 3 for details.

CLB: Common Leading Byte determines type of items and type of item identifiers.

Count: 8, 16, or 32-bits wide unsigned integer telling item count. Width of Count field depends on array type, see Section 4 for details.

Items: Array of items.

Type bits: See Section 4.

Array items are stored right after Count field. Items may have Identifier.

```
  Identifier     Item Payload
+------------+ +--------------+
|   0 - 256  | |    1 - 4G    |
+------------+ +--------------+
```

Figure 7: Array Item

2.3. Data Frames

Data Frames are collection of widely used data types. There are frames for boolean, integer and floating point numbers, UTF-8 encoded strings, dates, and timestamps. There is also frame for raw binary data. All data frames can be tagged with an identifier.

2.3.1. Boolean Frame

Boolean value (False/True) is defined by choosing corresponding Boolean Frame type. See Section 4.

```
   Leading Byte    Identifier
+-+---------+---+ +----------+
|X|1|2|3|4|5|6|7| | 0 - 256  |
+-+---------+---+ +----------+
   \       / \ /
     \   /    | Id bits
       |
       | Type bits
```

Figure 8: Boolean Frame

Identifier & Id bits: See Section 3 for details.

Type bits: See Section 4.

2.3.2. Integer Frames

Wide range of integer types are supported. Integer width and signedness are defined by choosing corresponding Integer Frame type. Signed integers are presented in two’s complement notation.

Integer payload is always stored in big-endian format. See Section 1.1 for details.

```
   Leading Byte    Identifier       Payload
+-+---------+---+ +----------+ +---------------+
|X|1|2|3|4|5|6|7| | 0 - 256  | |  1,2,4 or 8  |
+-+---------+---+ +----------+ +---------------+
   \       / \ /
     \   /    | Id bits
       |
       | Type bits
```

Figure 9: Integer Frames

Identifier & Id bits: See Section 3 for details.

Payload: Payload integer value.

Type bits: See Section 4.

2.3.3. Float Frames

Floating point number precision is defined by choosing corresponding Float Frame type. See Section 4 for frame types. Floats are presented in IEEE754 standard format and endianness is big-endian. See Section 1.1 for details.

```
   Leading Byte    Identifier      Payload
+-+---------+---+ +----------+ +------------+
|X|1|2|3|4|5|6|7| | 0 - 256  | |  2,4 or 8  |
+-+---------+---+ +----------+ +------------+
   \       / \ /
     \   /    | Id bits
       |
       | Type bits
```

Figure 10: Float Frames

Identifier & Id bits: See Section 3 for details.

Payload: Payload float value.

Type bits: See Section 4.

2.3.4. String Frame

String Frame can hold UTF-8 encoded string. If implementation supports String Frame it must be able to validate UTF-8 encoding. See Section 1.2 for details.

Frame capacity is defined by selecting corresponding String Frame type. See Section 4 for details.

```
      Leading Byte    Identifier   Length      Payload
   +-+--------+---+ +----------+ +-------+ +-----------+
   |X|1|2|3|4|5|6|7| | 0 - 256  | | 1/2/4 | |  0 - 4G   |
   +-+--------+---+ +----------+ +-------+ +-----------+
      \       / \ /
        \   /    | Id bits
          |
          | Type bits
```

Figure 11: String Frame

Identifier & Id bits: See Section 3 for details.

Length: 8, 16, or 32-bits wide unsigned integer telling length of string in bytes. Depends on String Frame type, see Section 4 for details.

Payload: UTF-8 encoded string.

Type bits: See Section 4.

2.3.5. Binary Frame

Binary Frame holds arbitrary binary data.

Frame capacity is defined by selecting corresponding Binary Frame type. See Section 4 for details.

```
      Leading Byte    Identifier   Length      Payload
   +-+--------+---+ +----------+ +-------+ +-----------+
   |X|1|2|3|4|5|6|7| | 0 - 256  | | 1/2/4 | |  0 - 4G   |
   +-+--------+---+ +----------+ +-------+ +-----------+
      \       / \ /
        \   /    | Id bits
          |
          | Type bits
```

Figure 12: Binary Frame

Identifier & Id bits: See Section 3 for details.

Length: 8, 16, or 32-bits wide unsigned integer telling length of payload in bytes. Depends on Binary Frame type, see Section 4 for details.

Payload: Arbitrary binary data.

Type bits: See Section 4.

2.3.6. DateTime Frames

DateTime Frames hold date or date and time in UTC timescale as UTF-8 encoded string. String formats are compatible with RFC 3339 [RFC3339].

If implementation supports any of DateTime Frames it must be able to validate UTF-8 encoding. See Section 1.2 for details. Besides string formats must be validated but date data validation is not part of RSK. On writing phase illegal string format must be handled as error. On reading phase string format violation can be handled by rising warning and let user decide continue reading or not.

Date frame types and corresponding date string formats:

Date: YYYY-MM-DD

DateTime: YYYY-MM-DDTHH:MM:SSZ

DateTimeMillis: YYYY-MM-DDTHH:MM:SS.SSSZ

```
   Leading Byte    Identifier   Date/DateTime/DateTimeMillis
+-+-------+-+---+ +----------+ +----------------------------+
|X|1|2|3|4|5|6|7| | 0 - 256  | |         10/20/24           |
+-+-------+-+---+ +----------+ +----------------------------+
   \       / \ /
     \   /    | Id bits
       |
       | Type bits
```

Figure 13: DateTime Frame

Identifier & Id bits: See Section 3 for details.

Date/DateTime/DateTimeMillis: Date, DateTime, or DateTimeMillis string depends of date frame type.

Type bits: See Section 4.

2.3.7. NTP Short Frame

NTP Short Frame holds NTP Short Format compatible timestamp. See RFC 5905 [RFC5905] for details.

```
   Leading Byte    Identifier   Seconds   Fraction
+-+-------+-+---+ +----------+ +-------+ +--------+
|X|1|2|3|4|5|6|7| | 0 - 256  | |   2   | |   2    |
+-+-------+-+---+ +----------+ +-------+ +--------+
   \       / \ /
     \   /    | Id bits
       |
       | Type bits
```

Figure 14: NTP Short Format Frame

Identifier & Id bits: See Section 3 for details.

Seconds: 16-bits unsigned integer telling seconds.

Fraction: 16-bits unsigned integer holding fractions of second.

Type bits: See Section 4.

2.3.8. NTP Timestamp Frame

NTP Timestamp Frame holds NTP Timestamp Format compatible timestamp. See RFC 5905 [RFC5905] for details.

```
   Leading Byte    Identifier   Seconds   Fraction
+-+-------+-+---+ +----------+ +-------+ +--------+
|X|1|2|3|4|5|6|7| | 0 - 256  | |   4   | |   4    |
+-+-------+-+---+ +----------+ +-------+ +--------+
   \       / \ /
     \   /    | Id bits
       |
       | Type bits
```

Figure 15: NTP Timestamp Format Frame

Identifier & Id bits: See Section 3 for details.

Seconds: 32-bits unsigned integer telling seconds.

Fraction: 32-bits unsigned integer holding fractions of second.

Type bits: See Section 4.

2.3.9. NTP Date Frame

NTP Date Frame holds NTP Date Format compatible date. See RFC 5905 [RFC5905] for details.

```
   Leading Byte    Identifier   Era   Era Offset   Fraction
+-+-------+-+---+ +----------+ +---+ +----------+ +--------+
|X|1|2|3|4|5|6|7| | 0 - 256  | | 4 | |    4     | |   8    |
+-+-------+-+---+ +----------+ +---+ +----------+ +--------+
   \       / \ /
     \   /    | Id bits
       |
       | Type bits
```

Figure 16: NTP Date Format Frame

Identifier & Id bits: See Section 3 for details.

Era: 32-bits signed integer telling the era of timestamp. See RFC 5905 [RFC5905] for era definitions.

Offset: 32-bits unsigned integer holding number of seconds since beginning of the Era.

Fraction: 64-bits unsigned integer holding fractions of second.

Type bits: See Section 4.

2.3.10. RSK Date Frame

RSK Date is optimized version of NTP Date Format defined in RFC 5905 [RFC5905].

Differences with NTP Date Format

Width of Era field: NTP Date Format has 32-bits wide Era field. Here Era is only 8-bits wide. Interpretation is same but with narrowed range. Epoch is same so Era 0 starts at 1900-01-01 00:00:00 UTC like in NTP Date Format.

Width of Fraction Field: NTP Date Format uses 64-bits wide Fraction field. Here fraction is only 16-bits wide and thus capable to 16 microsecond resolution.

```
   Leading Byte    Identifier   Era   Era Offset   Fraction
+-+-------+-+---+ +----------+ +---+ +----------+ +---------+
|X|1|2|3|4|5|6|7| | 0 - 256  | | 1 | |    4     | |   2     |
+-+-------+-+---+ +----------+ +---+ +----------+ +---------+
   \       / \ /
     \   /    | Id bits
       |
       | Type bits
```

Figure 17: RSK Date Frame

Identifier & Id bits: See Section 3 for details.

Era: 8-bit signed integer telling the era of timestamp. See RFC 5905 [RFC5905] for era definitions.

Offset: 32-bit unsigned integer holding number of seconds since beginning of the Era.

Fraction: 16-bit unsigned integer holding fractions of second.

Type bits: See Section 4.

2.4. Extended Frames

Extended Frame is concept to introduce new structures and data types in future versions of RSK. The most significant bit in Leading Byte is reserved for Extended Frames. In this version Extended bit must not be set when writing a RSK document. If Extended Frame is discovered on reading phase it must be handles as error in this version.

3. Identifiers

All data frames and some of the meta frames can be tagged with an identifier. Identifier can be defined as 8 or 16-bit wide unsigned integer or as length-prefixed UTF-8 encoded string. If identifier is not needed it can be set to Null.

Frame’s Leading Byte tells type of identifier. Identifier bytes are placed immediately after the Leading Byte. In case of integer identifier there is one or two bytes depending on selected integer identifier type. String identifier can take up to 256 bytes. See following sections for details.

3.1. Identifier Types in Leading Byte

Two least significant bits of Leading Byte are reserved for Id bits in all frame types which can be tagged with an identifier.

```
  Leading Byte     Identifier
+-+---------+---+ +----------+
|X|1|2|3|4|5|6|7| | 0 - 256  |
+-+---------+---+ +----------+
             \ /
              | Id bits
```

Figure 18: Identifier Types

Id bits values and identifier types:

00 Null Identifier. See Section 3.2.

01 8-bits wide Integer Identifier. See Section 3.3.

10 16-bits wide Integer Identifier. See Section 3.3.

11 String Identifier. See Section 3.4

3.2. Null Identifier

Some frames in a document may not need identifier so it can be left empty by setting it Null in Leading Byte.

3.3. Integer Identifiers

Integer identifier types are 8 or 16-bits wide unsigned integers. Integer identifiers are presented in big-endian format. See

Section 1.1 for details.

```
  Leading Byte      Uint8           Uint16
+---------------+ +-------+      +----------+
|X|1|2|3|4|5|6|7| |   1   |  OR  |    2     |
+---------------+ +-------+      +----------+
                  |                         |
                  \---Integer Identifier ---/
```

Figure 19: Integer Identifier

3.4. String Identifier

String identifier is length-prefixed and UTF-8 encoded. Length is presented by one byte as 8-bits wide unsigned integer at the beginning of identifier field. String identifier itself can be 0 - 255 bytes long.

If implementation supports string identifiers it must be able to validate UTF-8 encoding. See Section 1.2 for details.

```
  Leading Byte      Length     UTF-8 Encoded String
+---------------+ +--------+ +----------------------+
|X|1|2|3|4|5|6|7| |   1    | |       0 - 255        |
+---------------+ +--------+ +----------------------+
                  |                                 |
                  \------ String Identifier --------/
```

Figure 20: String Identifier

4. Frame Type Table

No	Frame	Type	Id	Array	Payload
1.	Null	0x00	[x]	[ ]	0
2.	Begin	0x04	[x]	[ ]	0
3.	End	0x08	[ ]	[ ]	0
4.	Boolean False	0x0C	[x]	[ ]	0
5.	Boolean True	0x10	[x]	[ ]	0
6.	TinyArray	0x14	[x]	[ ]	0 - 255 (items)
7.	Array	0x18	[x]	[ ]	0 - 64k (items)
8.	LongArray	0x1C	[x]	[ ]	0 - 4G (items)
9.	TinyString	0x20	[x]	[x]	0 - 255
10.	String	0x24	[x]	[x]	0 - 64k
11.	LongString	0x28	[x]	[x]	0 - 4G
12.	TinyBinary	0x2C	[x]	[x]	0 - 255
13.	Binary	0x30	[x]	[x]	0 - 64k
14.	LongBinary	0x34	[x]	[x]	0 - 4G
15.	Signed int 8-bits	0x38	[x]	[x]	1
16.	Signed int 16-bits	0x3C	[x]	[x]	2
17.	Signed int 32-bits	0x40	[x]	[x]	4
18.	Signed int 64-bits	0x44	[x]	[x]	8
19.	Unsigned int 8-bits	0x48	[x]	[x]	1
20.	Unsigned int 16-bits	0x4C	[x]	[x]	2
21.	Unsigned int 32-bits	0x50	[x]	[x]	4
22.	Unsigned int 64-bits	0x54	[x]	[x]	8
23.	Floating 16-bits	0x58	[x]	[x]	2
24.	Floating 32-bits	0x5C	[x]	[x]	4
25.	Floating 64-bits	0x60	[x]	[x]	8
26.	Date	0x64	[x]	[x]	10
27.	DateTime	0x68	[x]	[x]	20
28.	DateTimeMillis	0x6C	[x]	[x]	24
29.	NTP Short Format	0x70	[x]	[x]	4
30.	NTP Timestamp Format	0x74	[x]	[x]	8
31.	NTP Date Format	0x78	[x]	[x]	16
32.	RSK Date	0x7C	[x]	[x]	7

Frame Type Table columns:

Frame: Name of frame type. See Section 2 for detailed frame definitions.

Type: Hexadecimal value of Leading Byte with mask 0xFC. See Section 2.1 for detailed description of Leading Byte.

Id: All marked with X has identifier field. See Section 3 for details.

Array: All marked with X can be enclosed into a array. See Section 2.2.4 for details.

Payload: Payload length in bytes for data frames and item count for arrays.

5. Implementation Notes

RSK is designed so that implementations could have very small memory and other resource demands. Pay attention to memory usage and try to perform IO operations efficiently.

Implementations must make sure that well formed documents are written. On reading phase any deformation in document or frame structure must be detected and handled as error condition.

Implementations can vary depending on environment and usage. All implementations must support at least Begin and End Frames to be able to handle document structure. Other frame types may not be supported. Implementation may also support most or all frame types but not all identifier types. Some frame types can be also partially supported so that they can be detected and skipped on reading phase although their payload data is not interpreted.

6. Security Considerations

RSK is data encoding format and does not include any executable commands. Implementations must make sure that any parts of encoded documents are not leaked into execution memory. Even malformed documents must be handled so that memory leaks are avoided.

RSK does not include any means to validate payload data integrity. Protocols based on RSK or underlying mechanisms which are utilized by those protocols must take care of this. If data integrity is not checked can data get corrupted by malfunctioning devices, software, or malicious attackers.

7. IANA Considerations

The MIME media type for RSK documents is application/ruoska.

Type name: application

Subtype name: ruoska

Required parameters: n/a

Optional parameters: n/a

8. Normative References

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

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

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

Author's Address

Jukka-Pekka Makela
Janakkala, Tavastia Proper
Finland