Binary Application Record Encoding (BARE)
draft-devault-bare-07

Abstract

The Binary Application Record Encoding (BARE) is a data format used to represent application records for storage or transmission between programs. BARE messages are concise and have a well-defined schema, and implementations may be simple and broadly compatible. A schema language is also provided to express message schemas out-of-band.

Comments

Comments are solicited and should be addressed to the mailing list at ~sircmpwn/public-inbox@lists.sr.ht and/or the author(s).

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 https://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 12 November 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.
1. Introduction

The purpose of the BARE message encoding, like hundreds of others, is to encode application messages. The goals of such encodings vary (leading to their proliferation); BARE’s goals are the following:

* Concise messages
* A well-defined message schema
* Broad compatibility with programming environments
Simplicity of implementation

This document specifies the BARE message encoding, as well as a schema language that may be used to describe the layout of a BARE message. The schema of a message must be agreed upon in advance by each party exchanging a BARE message; message structure is not encoded into the representation. The schema language is useful for this purpose but not required.

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

1.2. Use-cases

The goals of a concise, binary, strongly-typed, and broadly-compatible structured message encoding format support a broad number of use-cases. Examples include:

* Self-describing authentication tokens for web services

* Opaque messages for transmitting arbitrary state between unrelated internet services

* A representation for packets in an internet protocol

* A structured data format for encrypted or signed application messages

* A structured data format for storing data in persistent storage

The conciseness of a BARE-encoded message enables representing structured data under strict limitations on message length in a large variety of contexts. The simple binary format may also be easily paired with additional tools, such as plain-text encodings, compression, or cryptography algorithms, as demanded by the application’s needs, without increasing the complexity of the message encoding. A BARE message has a comparable size and entropy to the underlying state it represents.

The BARE schema language also provides a means of describing the format of BARE messages without implementation-specific details. This encourages applications that utilize BARE to describe their state in a manner that other programmers can easily utilize for
application interoperation. The conservative set of primitives offered by BARE aids in making such new implementations easy to write.

2. Specification of the BARE Message Encoding

A BARE message is a single value of a pre-defined type, which may be of an aggregate type enclosing multiple values. Unless otherwise specified, there is no additional container or structure around the value; it is encoded plainly.

A BARE message does not necessarily have a fixed length, but the schema author may make a deliberate choice to constrain themselves to types of well-defined lengths if this is desired.

The names for each type are provided to establish a vocabulary for describing a BARE message schema out-of-band, by parties who plan to exchange BARE messages. The type names used here are provided for this informative purpose, but are more rigourously specified by the schema language specification in Section 3.

2.1. Primitive Types

Primitive types represent exactly one value.

**uint**

A variable-length unsigned integer encoded using the Unsigned Little Endian Base 128 (ULEB128). Every octet of the encoded value has the most-significant bit set, except for the last octet. The remaining bits are the zero-extended integer value in 7-bit groups, the least-significant group first.

The encoder MUST encode uint using the minimum necessary number of octets, and the decoder SHOULD raise an error if it encounters the opposite.

The maximum precision of such a number is 64-bits. The maximum length of an encoded uint is therefore 10 octets.

Numbers that require all ten octets will have 6 bits in the final octet that do not have meaning, between the least- and most-significant bits. The implementation MUST set these to zero.

**int**

A signed integer with variable-length encoding. Signed integers are represented as uint using a "zig-zag" encoding: positive values \(x\) are written as \(2x + 0\), negative values as...
-2x - 1. Another way of looking at it is that negative numbers are complemented, and whether to complement is encoded in bit 0.

The encoder MUST encode int using the minimum necessary number of octets, and the decoder SHOULD raise an error if it encounters the opposite.

The maximum precision of such a number is 64-bits. The maximum length of an encoded int is therefore 10 octets.

Numbers that require all ten octets will have 6 bits in the final octet that do not have meaning, between the least- and most-significant bits. The implementation MUST set these to zero.

u8, u16, u32, u64
Unsigned integers of a fixed precision, respectively 8, 16, 32, and 64 bits. They are encoded in little-endian (least significant octet first).

i8, i16, i32, i64
Signed integers of a fixed precision, respectively 8, 16, 32, and 64 bits. They are encoded in little-endian (least significant octet first), with two’s complement notation.

f32, f64
Floating-point numbers represented with the IEEE 754 [IEEE.754.1985] binary32 and binary64 floating point number formats.

bool
A boolean value, either true or false, encoded as a u8 type with a value of one or zero, respectively representing true or false.

If a value other than one or zero is found in the u8 representation of the bool, the message is considered invalid, and the decoder SHOULD raise an error if it encounters such a value.

str
A string of text. The length of the text in octets is encoded first as a uint, followed by the text data represented with the UTF-8 encoding [RFC3629].
If the data is found to contain invalid UTF-8 sequences, it is considered invalid, and the decoder SHOULD raise an error if it encounters such a value.

**data**

Arbitrary data of a variable length. The length (in octets) is encoded first as a uint, followed by the data itself encoded literally.

**data[length]**

Arbitrary data of a fixed "length", e.g. data[16]. The length (in octets) is not encoded into the message. The data is encoded literally in the message.

**void**

A type with zero length. It is not encoded into BARE messages.

**enum**

An unsigned integer value from a set of named values agreed upon in advance, encoded with the uint type.

An enum whose uint value is not a member of the values agreed upon in advance is considered invalid, and the decoder SHOULD raise an error if it encounters such a value.

Note that this makes the enum type unsuitable for representing several enum values that have been combined with a bitwise OR operation.

Using uint for enum value makes it possible to encode named values with different number of octets. Constant-length enum can be achieved when all the enum values are encoded by uints with the same number of octets.

### 2.2. Aggregate Types

Aggregate types may store zero or more primitive or aggregate values.

**optional<type>**

A value of "type" that may or may not be present, e.g. optional<u32>. Represented as either a u8 with a value of zero, indicating that the optional value is unset; or a u8 with a value of one, followed by the encoded data of the optional type.
An optional value whose initial u8 is set to a number other than zero or one is considered invalid, and the decoder SHOULD raise an error if it encounters such a value.

`list<type>`
A variable-length list of "type" values, e.g. list<str>. The length of the list (number of values) is encoded as a uint, followed by the encoded values of each member of the list concatenated.

`list<type>[length]`
A list of "length" values of "type", e.g. list<uint>[10]. The length is not encoded into the message. The encoded values of each member of the list are concatenated to form the encoded list.

`map<type A><type B>`
A mapping of "type B" values keyed by "type A" values, e.g. map<u32><str>. The encoded representation of a map begins with the number of key/value pairs encoded as a uint, followed by the encoded key/value pairs concatenated. Each key/value pair is encoded as the encoded key concatenated with the encoded value.

A message with repeated keys is considered invalid, and the decoder SHOULD raise an error if it encounters such a value.

union
A tagged union whose value may be one of any type from a set of types agreed upon in advance. Every type in the set is assigned a numeric identifier. The value is encoded as the selected type’s identifier represented with the uint encoding, followed by the encoded value of that type.

A union with a tag value that does not have a corresponding type assigned is considered invalid, and the decoder SHOULD raise an error if it encounters such a value.

struct
A set of values of arbitrary types concatenated in an order agreed upon in advance. Each value is called "field", and the field has a name and type.
2.3. User-Defined Types

A user-defined type gives a name to another type. This creates a distinct type whose representation is equivalent to the named type. An arbitrary number of user-defined types may be used for the same underlying type; each is distinct from the other.

2.4. Invariants

The following invariants are specified:

* Any type that is ultimately a void type (either directly or via a user-defined type) MUST NOT be used as an optional type, list value, map key, map value, or struct field type. Void types may only be used as members of the set of types in a tagged union.

* Enums MUST have at least one named value, and each named value of an enum MUST be unique.

* The lengths of fixed-length data and fixed-length list types MUST be at least one and MUST NOT be longer than 18,446,744,073,709,551,615 octets (the maximum value of a u64).

* Any map key type (directly or via a user-defined type) MUST be of a primitive type that is not f32, f64, data, data[length], or void.

* Unions MUST have at least one type, and each type of a union MUST be unique.

* Structs MUST have at least one field, and each field of a struct MUST have a unique name.

* Any user-defined type MUST be defined before used. Any user-defined type MUST NOT be defined recursively (directly or indirectly).

3. BARE Schema Language Specification

The use of the schema language is optional. Implementations SHOULD support decoding arbitrary BARE messages without a schema document, by defining the schema in a manner that utilizes more native tools available from the programming environment.

However, it may be useful to have a schema document for use with code generation, documentation, or interoperability. A domain-specific language is provided for this purpose.
3.1. Lexical Analysis

During lexical analysis, "#" is used for comments; if encountered, the "#" character and any subsequent characters are discarded until a line feed (%x0A) is found.

3.2. ABNF Grammar

The syntax of the schema language is provided here in Augmented Backus-Naur Form [RFC5234]. However, this grammar differs from [RFC5234] in that literal text strings are case-sensitive (e.g. "type" does not match "TypE").
schema = [WS] user-types [WS]

user-types = user-type [WS user-types]
user-type = "type" WS user-type-name WS any-type
user-type-name = UPPER *(ALPHA / DIGIT) ; first letter is uppercase

any-type = "uint" / "u8" / "u16" / "u32" / "u64"
any-type = / "int" / "i8" / "i16" / "i32" / "i64"
any-type = / "f32" / "f64"
any-type = / "bool"
any-type = / "str"
any-type = / "data" [length]
any-type = / "void"
any-type = / "optional" type
any-type = / "list" type [length]
any-type = / "map" type type


integer = 1*DIGIT

enum-values = enum-value [WS enum-values]
enum-value = enum-value-name [WS] "=" [WS] integer
enum-value-name = UPPER *(UPPER / DIGIT / "_")

type = [WS] "<" [WS] any-type [WS] ">

union-members = union-member [WS] "|" [WS] union-members
union-member = any-type [WS] "=" [WS] integer

struct-fields = struct-field [WS struct-fields]
struct-field = 1*ALPHA [WS] ":" [WS] any-type

UPPER = %x41-5A ; uppercase ASCII letters, i.e. A-Z
ALPHA = %x41-5A / %x61-7A ; A-Z / a-z
DIGIT = %x30-39 ; 0-9

WS = 1*(%x0A / %x09 / " "); whitespace

See Appendix B.1 for an example schema written in this language.
3.3. Semantic Elements

The names of fields and user-defined types are informational: they are not represented in BARE messages. They may be used by code generation tools to inform the generation of field and type names in the native programming environment.

Enum values are also informational. Values without an integer token are assigned automatically in the order that they appear, starting from zero and incrementing by one for each subsequent unassigned value. If a value is explicitly specified, automatic assignment continues from that value plus one for subsequent enum values.

Union type members are assigned a tag in the order that they appear, starting from zero and incrementing by one for each subsequent type. If a tag value is explicitly specified, automatic assignment continues from that value plus one for subsequent values.

4. Application Considerations

Message authors who wish to design a schema that is backwards- and forwards-compatible with future messages are encouraged to use union types for this purpose. New types may be appended to the members of a union type while retaining backwards compatibility with older message types. The choice to do this must be made from the first message version -- moving a struct into a union _does not_ produce a backwards-compatible message.

The following schema provides an example:

type Message union {MessageV1 | MessageV2 | MessageV3}

type MessageV1 ...

type MessageV2 ...

type MessageV3 ...

An updated schema that adds a MessageV4 type would still be able to decode versions 1, 2, and 3.

If a message version is later deprecated, it may be removed in a manner compatible with future versions 2 and 3 if the initial tag is specified explicitly.

type Message union {MessageV2 = 1 | MessageV3}
5. Future Considerations

To ensure message compatibility between implementations and backwards- and forwards-compatibility of messages, constraints on vendor extensions are required. This specification is final, and new types or extensions will not be added in the future. Implementors MUST NOT define extensions to this specification.

To support the encoding of novel data structures, the implementor SHOULD make use of user-defined types in combination with the data or data[length] types.

6. IANA Considerations

This memo includes no request to IANA.

7. Security Considerations

Message decoders are common vectors for security vulnerabilities. BARE addresses this by making the message format as simple as possible. However, the decoder MUST be prepared to handle a number of error cases when decoding untrusted messages, such as a union type with an invalid tag, or an enum with an invalid value. Such errors may also arise by mistake, for example when attempting to decode a message with the wrong schema.

Support for data types of an arbitrary, message-defined length (lists, maps, strings, etc) is commonly exploited to cause the implementation to exhaust its resources while decoding a message. However, legitimate use-cases for extremely large data types (possibly larger than the system has the resources to store all at once) do exist. The decoder MUST manage its resources accordingly, and SHOULD provide the application a means of providing their own decoder implementation for values that are expected to be large.

There is only one valid interpretation of a BARE message for a given schema, and different decoders and encoders should be expected to provide that interpretation. If an implementation has limitations imposed from the programming environment (such as limits on numeric precision), the implementor MUST document these limitations, and prevent conflicting interpretations from causing undesired behavior.

8. Normative References

Appendix A. Example Values

This section lists example values in decimal, as string, or as named value (left or top), and their encoded representation in hexadecimal (right or bottom).

### uint

<table>
<thead>
<tr>
<th>Value</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>01</td>
</tr>
<tr>
<td>126</td>
<td>7e</td>
</tr>
<tr>
<td>127</td>
<td>7f</td>
</tr>
<tr>
<td>128</td>
<td>80 01</td>
</tr>
<tr>
<td>129</td>
<td>81 01</td>
</tr>
<tr>
<td>255</td>
<td>FE 01</td>
</tr>
</tbody>
</table>

### int

<table>
<thead>
<tr>
<th>Value</th>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>00</td>
</tr>
<tr>
<td>1</td>
<td>02</td>
</tr>
<tr>
<td>-1</td>
<td>01</td>
</tr>
<tr>
<td>63</td>
<td>7e</td>
</tr>
<tr>
<td>-63</td>
<td>7d</td>
</tr>
<tr>
<td>64</td>
<td>80 01</td>
</tr>
<tr>
<td>-64</td>
<td>7f</td>
</tr>
<tr>
<td>65</td>
<td>82 01</td>
</tr>
<tr>
<td>-65</td>
<td>81 01</td>
</tr>
<tr>
<td>255</td>
<td>FE 03</td>
</tr>
<tr>
<td>-255</td>
<td>FD 03</td>
</tr>
</tbody>
</table>
u32
0 00 00 00 00
1 01 00 00 00
255 FF 00 00 00

i16
0 00 00
1 01 00
-1 FF FF
255 FF 00
-255 01 FF

f64
0.0 00 00 00 00 00 00 00
1.0 00 00 00 00 00 00 f0 3f
2.55 66 66 66 66 66 66 04 40
-25.5 00 00 00 00 00 80 39 c0

bool
true 01
false 00

str
"BARE" 04 42 41 52 45

data
Example value is in hexadecimal.
aa ee ff ee dd cc bb aa ee dd cc bb ee dd cc bb
10 aa ee ff ee dd cc bb aa ee dd cc bb ee dd cc bb

data[16]
Example value is in hexadecimal.
aa ee ff ee dd cc bb aa ee dd cc bb ee dd cc bb
aa ee ff ee dd cc bb aa ee dd cc bb ee dd cc bb

void
Not encoded.

enum {FOO BAR = 255 BUZZ}
FOO 00
BAR FF 01
BUZZ 80 02
optional<u32>
  (unset) 00
  0 01 00 00 00
  1 01 01 00 00
  255 01 FF 00 00

list<str>
  "foo" "bar" "buzz"
  03 03 66 6f 03 62 61 72 61 72 04 62 75 7A 7A

list<uint>[10]
  01 254 255 256 257 126 127 128 129
  01 FF 01 80 02 81 02 7E 7F 80 01 81 01

map<u32><str>
  0 => "zero"
  1 => "one"
  255 => "two hundreds and fifty five"
  03 00 00 00 00 00 04 7A 65 72 6F 01 00 00 00 03 6F
  6E 65 6E FF 03 00 00 00 1B 74 77 6F 20 68 75 6E 64 72
  65 64 72 20 66 69 76 65

union {int | uint = 255 | str}
  0 00 00
  1 00 02
  1 00 01
  -1 00 01
  255 00 FE 03
  255 FF 01 FF 01
  -255 00 FD 03
  "BARE" 80 02 04 42 41 52 45

struct {foo: uint bar: int buzz: str}
  foo => 255
  bar => -255
  buzz => "BARE"
  FF 01 FD 03 04 42 41 52 45

Appendix B. Example Company

An example company that uses BARE to encode data about customers and employees.
B.1. Message Schema

The following is an example of a schema written in the BARE schema language.

type PublicKey data[128]
type Time str # ISO 8601

type Department enum {
  ACCOUNTING
  ADMINISTRATION
  CUSTOMER_SERVICE
  DEVELOPMENT
  # Reserved for the CEO
  JSMITH = 99
}

type Address list<str>[4] # street, city, state, country

type Customer struct {
  name: str
  email: str
  address: Address
  orders: list<struct {
    orderId: i64
    quantity: i32
  }>
  metadata: map<str><data>
}

type Employee struct {
  name: str
  email: str
  address: Address
  department: Department
  hireDate: Time
  publicKey: optional<PublicKey>
  metadata: map<str><data>
}

type TerminatedEmployee void

type Person union {Customer | Employee | TerminatedEmployee}
B.2. Encoded Messages

Some basic example messages in hexadecimal are provided for the schema specified in Appendix B.1.

A "Person" value of type "Customer" with the following values:

name            James Smith
email           jsmith@example.org
address         123 Main St; Philadelphia; PA; United States
orders (1)      orderId: 4242424242; quantity: 5
metadata        (unset)

Encoded BARE message:

00 0b 4a 61 6d 65 73 20 53 6d 69 74 68 12 6a 73
   6d 69 74 68 40 65 78 61 6d 70 6c 65 2e 6f 72 67
   0b 31 32 33 20 4d 61 69 6e 20 53 74 0c 50 68 69
   6c 61 64 65 6c 70 68 69 61 02 50 41 0d 55 6e 69
   74 65 64 20 53 74 61 74 65 73 01 b2 41 de fc 00
00 00 00 05 00 00 00 00

Encoded BARE message, but characters of strings are decoded:

00 0b  J  a  m  e  s     S  m  i  t  h 12  j  s
   m  i  t  h  @  e  x  a  m  p  l  e  .  o  r  g
   0b 1 2 3     M  a  i  n     S  t 0c  P  h  i
   l  a  d  e  l  p  h  i  a 02  P  A 0d  U  n  i
t  e  d     S  t  a  t  e  s 01 b2 41 de fc 00
00 00 00 05 00 00 00 00

A "Person" value of type "Employee" with the following values:

name            Tiffany Doe
email           tiffanyd@acme.corp
address         123 Main St; Philadelphia; PA; United States
department      ADMINISTRATION
hireDate        2020-06-21T21:18:05Z
publicKey       (unset)
metadata (unset)

Encoded BARE message:

01 0b 54 69 66 66 61 6e 79 20 44 6f 65 12 74 69
ff f a n y d @ a c m e . c o r p
0b 31 32 33 20 4d 61 69 6e 20 53 74 0c 50 68
6c 64 65 65 6c 70 68 69 61 02 50 41 0d 55
6e 69 74 65 64 20 53 74 61 74 65 73 01 14
32 30 32 30 - 0 6 - 2 1 T 2 1 : 1 8 : 0 5 Z
00 00

A "Person" value of type "TerminatedEmployee".

Encoded BARE message:

02

Appendix C. Complex Data

BARE schema examples for complex data structures.

C.1. Simple Hierarchical Data

Recursive data types are forbidden in BARE. The following examples show how linked list and binary tree, widely used recursive data types, can be encoded in BARE messages.

As BARE supports variable-length lists, encoding of linked list is straightforward.

type Element struct {
    str: what
}

type LinkedList list<Element>
A binary tree can be encoded to BARE’s variable-length list with 2x + 1 and 2x + 2 indexing.

```go
type Node optional<struct {
    str: what
}>

type BinaryTree list<Node>
```

C.2. JSON Schema

Sometimes it is needed to deal with generic format of data. When the use-case for recursive types is encountered, each element to encode needs to be identified.

```go
type ElementId uint

type False void

type True void

type Null void

type Object map<str><ElementId>

type Array list<ElementId>

type Element union {
    False
    True
    Null
    f64
    str
    Object
    Array
}

type JSONDocument list<Element>
```

C.3. Graph

It is not possible to encode pointers in BARE. However, an arbitrary graph can be encoded in the lists of nodes and connections.
type NodeId uint

type Node struct {
    what: str
}

type Connection struct {
    from: NodeId
    to: NodeId
    why: str
}

type Graph struct {
    nodes: map<NodeId><Node>
    edges: list<Connection>
}

Appendix D. Design Decisions

This section documents the reasoning behind the decisions made during BARE specification process.

*f32 and f64 are fully compliant with IEEE 754 [IEEE.754.1985]*

The use-case is a sensor sending NaN values or encoding of infinity in scientific applications.

The consequences are that encoded values of f32 and f64 types are not canonical, and therefore forbidden as map keys.

*Types of a union needs to be unique*

However, user-defined types are distinct types, so it is not a problem overall.

*Recursive types are forbidden*

Recursive types bring the possibility to encode arbitrary tree data structures for the price of:

1. runtime errors for cyclic references,
2. possible stack overflows during encoding/decoding when recursive encoders/decoders are used,
3. confusion because they do not come with pointers, although data to encode usually uses pointers.

It is not worth it.
The consequence is that recursive types need to be mapped to non-recursive types when used.

*Namespaces or imports are not used*
   It would increase complexity. BARE schema language is simple.

*There is no bitmap type*
   Use data[length] instead.

*There is no date/time type*
   Use u64 for timestamp or str.

*There is no ordered map type*
   Ordered maps are not widely supported in programming environments. Users that want to use ordered maps can use a list of pairs:

```
list<struct {
    key: KeyType
    val: ValType
}>
```

Author’s Address

Drew DeVault
SourceHut
454 E. Girard Ave #2R
Philadelphia, PA 19125
United States of America
Phone: +1 719 213 5473
Email: sir@cmpwn.com
URI: https://sourcehut.org/
Discarding Priority of RTP Video Packets
draft-dong-priority-rtp-packet-02

Abstract

This document illustrates that significance difference or discarding priority might exist among RTP packets which encapsulate video streaming data with the existing modern video codecs, i.e., H.264/AVC, SVC, H.265/HEVC and H.266/VVC.

The document overviews the RTP NALU header format for the existing modern video codecs. Each contains at least one field that indicates the RTP packet’s relative significance within the video stream. With the dominance of video traffic in the Internet, selectively dropping RTP packets from competing video streams according to their significances or discarding priorities could be a complementary mechanism when dealing with network congestion. The document proposes the Differentiated Services Code Point (DSCP) value mapping to the RTP packet discarding priority carried in the RTP NALU header. The document also proposes a new Hop-by-Hop Extension Header (HbH-EH) with a value that is copied from the discarding priority of the RTP packet, if the 6-bit DSCP value is not long enough for the mapping.

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 https://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."
1. Introduction

The modern video codecs, e.g., H.264/AVC [H.264], SVC [H.264], H.265/HEVC [H.265], and H.266/VVC [ISO23090-3] [VVC] use the NAL-unit-based syntax structure. The NAL unit structure provides convenient packetization/framing of video data to be transmitted in packet-based systems using transport protocols such as RTP [RFC3550]. The transport layer can identify the boundaries among adjacent NAL units without use of start code. Therefore, the overhead for these start codes can be eliminated. Depending on the characteristics of the NAL unit(s) encapsulated in a RTP packet, the priority/importance of RTP packets from the same video streaming flow could differ from each other. In the following, we firstly overview how the priority information is carried in RTP packets for H.264/AVC, SVC, H.265/HEVC, and H.266/VVC by referring to [RFC6184] [RFC6190] [RFC7798] [RTP.VVC]
respectively. Next we discuss how to make the network layer aware of and utilize such priority information for selective packet dropping when network congestion happens and outgoing buffer overflows.

2. Terms and Abbreviations

The terms and abbreviations used in this document are listed below.

* AF: Assured Forwarding
* AP: Aggregation Packet
* AVC: Advanced Video Coding
* DF: Default Forwarding
* DSCP: Differentiated Services Code Point
* EF: Expedited Forwarding
* HDTV: High Definition Television
* HEVC: High Efficiency Video Coding
* HbH-EH: Hop-by-Hop Extension Header
* IDR: Instantaneous Decoding Refresh
* FU: Fragmentation Unit
* MANE: Media Aware Network Element
* MTAP: Multi-Time Aggregation Packet
* NAL: Network Abstract Layer
* PACI: PAyload Content Information
* PHB: Per Hop Behavior
* QoE: Quality of Experience
* QoS: Quality of Service
* RTP: Real Time Protocol
* STAP: Signal-Time Aggregation Packet
The above terminology is defined in greater details in the remainder of this document.

3. Packet Level Priority

For different versions of video encoding schemes, the RTP packet payload format has been and is being standardized. Within a video flow, the importance or discarding priority can differ among different RTP packets, depending on the NAL unit(s) encapsulated in the RTP packets. In the following, we give a brief overview of such property, which is shown in different versions of video encoders.

3.1. Packet Level Priority Difference in H.264 RTP Packets

The H.264 video codec [H.264] has a very broad application range that covers all forms of digital compressed video, from low bitrate Internet streaming applications to HDTV broadcast and digital cinema applications with nearly lossless coding. The coded video data is organized into NAL units, each of which contains an integer number of bytes. The H.264/AVC specification adopts a byte stream format. Each NAL unit has a prefix of a specific pattern of three bytes, which is called a start code prefix. The boundaries of the NAL unit can then easily be detected by searching the coded data for this unique start code prefix pattern. A set of NAL units in a specified form comprises as an access unit. The decoding of each access unit results in one decoded picture.

The syntax and semantics of the NAL unit type octet are specified in [H.264], includes the essential properties of the NAL unit type octet in the NAL unit header. The RTP packet for H.264 video [RFC6184] inherits the same NAL unit header. As shown in Figure 1, the 2 bits NRI field (i.e., nal_ref_idc) indicates the relative importance/transport priority of the NRI unit determined by the encoder. A value of 00 indicates that the content of the NAL unit is not used to reconstruct reference pictures for inter picture prediction. Such NAL units can be discarded without risking the integrity of the reference pictures. Values greater than 00 indicate that the decoding of the NAL unit is required to maintain the integrity of the reference pictures. The H.264 specification requires that the value of NRI SHALL be equal to 0 for all NAL units having nal_unit_type equal to 6, 9, 10, 11, or 12. For NAL units having nal_unit_type equal to 7 or 8 (indicating a sequence parameter set or a picture
parameter set, respectively), an H.264 encoder should set the value of NRI to '11'. For coded slice NAL units of a primary coded picture having nal_unit_type equal to 5 (indicating a coded slice belonging to an IDR picture), an H.264 encoder sets the value of NRI to '11'. Non-IDR coded slice is specified with '10' NRI value, coded slice data partition A has '10' NRI value, while partition B and C have '01' NRI value.

```
+---------------+
|0|1|2|3|4|5|6|7|
+---------------+
|F|NRI| Type   |
+---------------+

The Structure of the H.264 NAL Unit Header.

Figure 1
```

The 'Type' field indicates the payload format with three different basic payload structures:

* Single NAL Unit Packet: Contains only a single NAL unit in the payload. The NRI field is associated with this single NAL unit.

* Aggregation Packet (AP): Packet type used to aggregate multiple NAL units into a single RTP payload. This packet exists in four versions, the Single-Time Aggregation Packet type A (STAP-A), the Single-Time Aggregation Packet type B (STAP-B), Multi-Time Aggregation Packet (MTAP) with 16-bit offset (MTAP16), and Multi-Time Aggregation Packet (MTAP) with 24-bit offset (MTAP24). A NAL unit header is followed by one or more NAL units in aggregation packets. The value of NRI is the maximum of all the NAL units carried in the aggregation packet.

* Fragmentation Unit (FU): Used to fragment a single NAL unit over multiple RTP packets. It exists with two versions, FU-A and FU-B respectively. Each FU packet has a FU indicator which has the same format as above. The value of the NRI field is set according to the value of the NRI field in the fragmented NAL unit, which means all the FU packets belong to the same NAL unit have the same NRI value.
3.2. Packet Level Priority Difference in SVC RTP Packets

Scalable Video Coding (SVC) extension of the H.264/AVC video coding standard is specified in Amendment 3 to ISO/IEC 14496 Part 10 [ISO_IEC14496-10] and equivalently in Annex G of ITU-T Rec. H.264 [H.264]. SVC defines a coded video representation in which a given bitstream offers representations of the source material at different levels of scalability: spatial (picture size), quality (or Signal-to-Noise Ratio (SNR)), and temporal (pictures per second). Bitstream components associated with a given level of spatial, quality, and temporal fidelity are identified using corresponding parameters in the bitstream: dependency_id, quality_id, and temporal_id. There are three additional octets in the NAL unit header of SVC RTP packets [RFC6190], which are shown in Figure 2.

```
+---------------+---------------+---------------+
|0|1|2|3|4|5|6|7|0|1|2|3|4|5|6|7|
+---------------+---------------+---------------+
|R|I| PRID   |N| Did | QID  | TID |U|D|O| RR|
+---------------+---------------+---------------+
```

Additional Octets in the SVC NAL Unit Header.

Figure 2

The priority of a NAL unit in SVC video stream can be further specified by the priority_id field (PRID), which has 6 bits. A lower value of PRID indicates a higher priority.

3.3. Packet Level Priority Difference in H.265 RTP Packets

The H.265/HEVC [H.265] significantly improves coding efficiency over H.264. Similarly, H.265 also includes a Video Coding Layer (VCL), which is often used to refer to the coding-tool features, and a Network Abstraction Layer (NAL), which is often used to refer to the systems and transport interface aspects of the codecs. HEVC includes an improved support of temporal scalability over H.264, by inclusion of the signaling of TemporalId in the NAL unit header. HEVC maintains the NAL unit concept of H.264 with modifications. The RTP packet for H.265/HEVC video [RFC7798] uses a two-byte NAL unit header as shown in Figure 3.
The 3 bits field TID specifies the temporal identifier of the NAL unit plus 1. The value of TemporalId is equal to TID minus 1. The TID value indicates (among other things) the relative importance of an RTP packet. For example, because NAL units belonging to higher temporal sub-layers are not used for the decoding of lower temporal sub-layers. A lower value of TID indicates a higher importance. More-important NAL units might need to be better protected against transmission loss or packet dropping than less-important NAL units.

```
+---------------+---------------+
|0|1|2|3|4|5|6|7|0|1|2|3|4|5|6|7|
+-------------------------------+
|F|   Type    |  LayerId  | TID |
+-------------------------------+
```

The Structure of the HEVC NAL Unit Header.

Figure 3

The type field indicates the different types of RTP packet payload structures.

* Single NAL Unit Packet: Contains only a single NAL unit in the payload. The TID field is associated with this single NAL unit.

* Aggregation Packet (AP): Packet type used to aggregate multiple NAL units into a single RTP payload. A payload header is followed by one or more NAL units in aggregation packets. The value of TID is set as the lowest value of TID of all the aggregated NAL units.

* Fragmentation Unit (FU): Used to fragment a single NAL unit over multiple RTP packets. Each FU packet has a FU payload header which has the same format as above. The value of the TID field is set according to the value of the TID field in the fragmented NAL unit, which means all the FU packets belong to the same NAL unit have the same TID value.

* PAYload Content Information (PACI): Used to increase the payload header efficiency. The value of TID is a copy of the TID field of the PACI payload NAL unit or NAL-unit-like structure.
3.4. Packet Level Priority Difference in H.266 RTP Packets

Versatile Video Coding (VVC) is formally published as both ITU-T Recommendation H.266 [VVC] and ISO/IEC International Standard 23090-3 [ISO23090-3]. VVC is reported to provide significant coding efficiency gains over H.265/HEVC, and other earlier video codecs. The RTP payload format for H.266/VVC [RTP.VVC] allows for packetization of one or more Network Abstraction Layer (NAL) units in each RTP packet payload as well as fragmentation of a NAL unit into multiple RTP packets.

VVC maintains the NAL unit concept of HEVC with modifications. VVC uses a two-byte NAL unit header, as shown in Figure 4. The payload of a NAL unit refers to the NAL unit excluding the NAL unit header.

```
+---------------+---------------+
|0|1|2|3|4|5|6|7|0|1|2|3|4|5|6|7|
+---------------+---------------+
|F|Z| LayerID | Type | TID |
+---------------+---------------+
```

The Structure of the VVC NAL Unit Header.

Similar to H.265, the TID value indicates (among other things) the relative importance of an RTP packet, for example, because NAL units belonging to higher temporal sublayers are not used for the decoding of lower temporal sublayers. A lower value of TID indicates a higher importance. More-important NAL units might need to be better protected against transmission loss or packet dropping than less-important NAL units.

The LayerID field is used to identify the layer a NAL unit belongs to, wherein a layer may be, e.g., a spatial scalable layer, a quality scalable layer, a layer containing a different view, etc. The LayerID has integer values, where higher values designate components that are higher in the hierarchy. Decoding of a particular component requires the availability of all the components it depends upon, either directly, or indirectly. So the NAL unit with lower LayerID would be likely be used to predict the NAL units with higher LayerID, therefore likely to be more important.

The type field indicates the different types of RTP packet payload structures.

* Single NAL Unit Packet: Contains only a single NAL unit in the payload. The TID field is associated with this single NAL unit.
* Aggregation Packet (AP): Packet type used to aggregate multiple NAL units into a single RTP payload. A payload header is followed by one or more NAL units in aggregation packets. The value of TID is set as the lowest value of TID of all the aggregated NAL units.

* Fragmentation Unit (FU): Used to fragment a single NAL unit over multiple RTP packets. Each FU packet has a FU payload header which has the same format as above. The value of the TID field is set according to the value of the TID field in the fragmented NAL unit, which means all the FU packets belong to the same NAL unit have the same TID value.

4. Implementation of Priority-Based Discarding of RTP Video Packets

Due to the explicit layering in the protocol stack, the upper layer data or headers are transparent to the network layer. The priority or importance associated with the NAL units encapsulated in RTP packets is invisible to intermediate routers. The concept of media-aware network element (MANE) was introduced in [RFC6184], which is a network element, such as a middlebox or application layer gateway that is capable of parsing certain aspects of the RTP payload headers or the RTP payload and reacting to the contents. The concept of a MANE goes beyond normal routers or gateways in that a MANE has to be aware of the signaling (e.g., to learn about the payload type mappings of the media streams) and that it has to be trusted when working with Secure Real-time Transport Protocol (SRTP) [RFC3711]. The advantage of using MANEs is that they allow packets to be dropped according to the needs of the media coding. For example, if a MANE has to drop packets due to congestion on a certain link, it can identify and remove those packets whose elimination produces the least adverse effect on the user experience.

MANEs can access the field that indicates the importance of the NAL unit, which was overviewed in the previous section. In summary:

* The two bits NRI field in H.264 and SVC NAL unit header.

* The 3 bits TID filed in H.265 and H.266 NAL unit header.

* The 6 bits PRID field in SVC NAL unit extension header, which provides even finer granularity of priority differentiation for NAL units in SVC.

* The 6 bits LayerID field in H.266 NAL unit payload header, which provides even finer granularity of priority differentiation for NAL units in VVC.
MANE is an overlay network element that might be co-located with a few routers, e.g., at network edge. So when network congestion happens in other routers that is not deployed with MANE, the packet dropping is subject to DiffServ classification [RFC2475]. DiffServ uses a 6-bit differentiated services code point (DSCP) in the 8-bit differentiated services field (DS field) in the IP header for packet classification purposes. In theory, a network could have up to 64 different traffic classes by using the 64 available DSCP values. However, the commonly defined per-hop behaviors only include 4 categories:

* Default Forwarding (DF) PHB, which is typically best-effort traffic.

* Expedited Forwarding (EF) PHB, which is dedicated to low-loss, low-latency traffic.

* Assured Forwarding (AF) PHB, which gives assurance of delivery under prescribed conditions

* Class Selector PHBs, which maintain backward compatibility with the IP precedence field.

We consider the two video types: interactive video and non-interactive video. The video stream from both types could be encoded according to H.264, SVC, H.265, H.266. For H.264 and SVC, the NAL units have the NRI field to indicate the discarding priority of the RTP packets. For H.265 and H.266, the NAL units have the TID field to indicate the discarding priority of the RTP packets. The NRI field is of 2 bits, and the TID field is of 3 bits, thus the DSCP value can be mapped according to either the NRI value or the TID value, as well as the video types. In general, the NAL units with the same NRI value or the TID value in interactive video has higher priority than in non-interactive video. The recommended DSCP values for RTP packets according to NRI value and video type are shown in Table 1. The recommended DSCP values for RTP packets according to TID value and video type are shown in Table 2. These values are based on the framework and recommended values in [RFC4594].
Either the video host or the MANE at the DiffServ domain edge can do the mapping and set up the DSCP value for each RTP packet. The discarding precedence of the RTP packets can be determined when link congestion happens.

Compared to H.265, SVC and H.266 employ additional scalability other than the temporal scalability, namely spatial scalability and quality scalability. Thus in the NAL extension header for SVC, there is an additional field (i.e., PRID) used to indicate the importance of the RTP packet at finer granularity. The PRID field occupies 6 bits.
additionally. In the NAL unit header for h.266, the LayerID is used to identify the layer a NAL unit belongs to, wherein a layer may be, e.g., a spatial scalable layer, a quality scalable layer, a layer containing a different view, etc. The LayerID field provides the importance information of the RTP packet at finer granularity as well. The LayerID field occupies 6 bits additionally.

It is not feasible to use the DSCP mapping to indicate the additional discarding precedence provided by the 6 bits PRID, and the 6 bits LayerID. Thus, other solutions need to explored in the future if discarding precedence at finer granularity is considered to be supported.

5. IANA Considerations

This document requires no actions from IANA.

6. Security Considerations

This document introduces no new security issues.

7. Acknowledgements

8. Informative References


Authors’ Addresses

Lijun Dong
Futurewei Technologies Inc.
United States of America
Email: lijun.dong@futurewei.com
Privacy Considerations for Web Feed Readers
draft-nottingham-feed-privacy-00

Abstract

This specification collects privacy-enhancing guidelines for Web feed readers.

About This Document

This note is to be removed before publishing as an RFC.

Status information for this document may be found at https://datatracker.ietf.org/doc/draft-nottingham-feed-privacy/.

Information can be found at https://mnot.github.io/I-D/.

Source for this draft and an issue tracker can be found at https://github.com/mnot/I-D/labels/feed-privacy.

Note to Readers

This draft is a quick straw-man; it is intended to assess implementer and community interest in the topic, not to state concrete requirements (yet). Feedback much appreciated.

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 https://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 23 December 2022.
1. Introduction

Many web sites offer a feed of updates to their content, using [ATOM] or [RSS]. While they are consumed in a variety of ways and for a variety of purposes, web feeds are often presented to users by dedicated software, colloquially known as a "feed reader."
Feed readers use HTML and HTTP, and can be considered as part of the web, but one that is distinct from web browsers. Unlike browsers, feed readers do not easily facilitate cross-site tracking or behavioural advertising, because their capabilities are more limited, thereby establishing an alternative, more privacy-respecting web platform.

At the same time, browsers are protecting privacy in increasingly sophisticated ways; for example, by taking steps to prevent active fingerprinting [FINGERPRINTING].

This specification seeks to codify these privacy-enhancing distinctions while incorporating browser’s privacy advances by offering a definition for "feed reader" in Section 2, providing guidelines for how they make requests in Section 3, and providing guidelines for their handling of content in Section 4.

1.1. Notational Conventions

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2. Feed Readers

A feed reader acts as a user agent (per [HTTP]) that consumes and presents information from documents in [ATOM], [RSS], and/or similar formats to users.

A feed reader might be local software program on a host that the user controls, or a remote service that they access over the Internet, such as through a web browser. Typically, a feed reader will allow the user to subscribe to URIs that identify feeds, and regularly poll those URIs for new content. When a feed entry has already been seen, a reader might keep this state.

Feed readers make HTTP requests and parse, render and display HTML content (including some embedded content). Users can also follow links from content in a feed reader.
3. Making Feed Requests

When a feed reader makes a request for a feed document, privacy can be impacted in several ways. This section contains guidelines for such requests; note that they do not apply to requests for embedded content and user-initiated navigation to links in content (see Section 4).

3.1. Encryption

In HTTP, encryption protects communication from observation and modification, and is used to establish the identity of the server. Feed readers, therefore, are expected to follow best current practice for encryption, as captured in the relevant RFCs and industry practice.

This includes implementation of the most recent version of TLS (as of this writing, [TLS13]), the Strict-Transport-Security mechanism [HSTS], and Certificate Transparency checking [TRANS].

3.2. Cookies

The HTTP Cookie mechanism has aspects that are problematic for privacy; see, eg., Part xx of [COOKIES]. Therefore, when making feed requests feed readers MUST NOT send the Cookie header field, and when receiving feed responses, they MUST NOT process the Set-Cookie header field.

3.3. ETags

HTTP ETags (see Part x.x of [HTTP]) are especially useful to feed readers, as they enable more efficient transfers when there have been no changes to a feed. However, they can also be used to track user activity. Therefore, feed readers SHOULD periodically send requests without If-None-Match header fields, to assure that ETags are changed.

3.4. User-Agent

Feed readers SHOULD NOT include more significant detail than an identifier for the software being used and its version. In particular, detail about libraries used and other aspects of the environment can contribute to the formation of an identifier for the user.
3.5. Client IP Address

Feed readers SHOULD take steps to prevent servers hosting feeds from using the client's IP address to identify them or track their activity. For example, [MASQUE] might be used to this end.

4. Handling Feed Content

When a feed reader displays a feed content (including an individual feed entry) to its user, interaction with the feed’s server is limited in several ways to reduce privacy impact. This section outlines those limits.

4.1. Requesting Remote Resources

Feed readers MAY make requests for remote resources that are explicitly part of the feed or feed entry's metadata. For example, a feed reader might fetch the URL in the atom:logo element (defined in Section 4.2.7 of [ATOM]) in order to present it to the user.

Feed readers MAY make requests for remote resources that are embedded in feed content. However, the user MUST be able to control this behaviour.

4.2. Executing Scripts

When handling feed content, feed readers MUST NOT execute embedded or linked scripts.

4.3. Reporting

Feed readers MUST NOT trigger reporting mechanisms designed for Web browsers when handing feed content. For example, [NEL], [CSP].

4.4. Following Links

When a user explicitly follows a link in a feed reader, their expectation will be that it either opens in their preferred Web browser, or that the resulting functionality is equivalent (e.g., a browser embedded in the feed reader). Once a link is followed, the feed reader is no longer handling feed content; the user’s activity is now either in a separate Web browser, or in an embedded web browser that is considered a distinct context.

Therefore, the context used to follow a link MUST be separate from that used to make requests for feed documents. In particular, separate underlying connections are to be used, and no state such as cookies is to be shared.
5. IANA Considerations

This document has no actions for IANA.

6. Security Considerations

_TBD_

7. References

7.1. Normative References


7.2. Informative References


Author’s Address

Mark Nottingham
Prahran
Australia
Email: mnot@mnot.net
URI: https://www.mnot.net/
New UUID Formats
draft-peabody-dispatch-new-uuid-format-04

Abstract

This document presents new Universally Unique Identifier (UUID) formats for use in modern applications and databases.

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 https://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 25 December 2022.

Copyright Notice

Copyright (c) 2022 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 (https://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 Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.
Table of Contents

1. Introduction ........................................... 3
2. Terminology ............................................. 4
   2.1. Requirements Language ................................. 4
   2.2. Abbreviations ....................................... 5
3. Summary of Changes ....................................... 5
   3.1. changelog ........................................... 5
4. Variant and Version Fields ............................... 7
5. New Formats ............................................. 8
   5.1. UUID Version 6 ....................................... 8
   5.2. UUID Version 7 ..................................... 10
   5.3. UUID Version 8 ..................................... 11
   5.4. Max UUID .......................................... 12
6. UUID Best Practices ..................................... 12
   6.1. Timestamp Granularity ............................... 12
   6.2. Monotonicity and Counters ........................... 14
   6.3. Distributed UUID Generation ......................... 17
   6.4. Collision Resistance ................................ 18
   6.5. Global and Local Uniqueness ......................... 18
   6.6. Unguessability ..................................... 19
   6.7. Sorting ............................................ 19
   6.8. Opacity ............................................ 19
   6.9. DBMS and Database Considerations ................... 19
7. IANA Considerations ...................................... 20
8. Security Considerations .................................. 20
9. Acknowledgements ........................................ 20
10. Normative References .................................... 20
11. Informative References .................................. 21
Appendix A. Example Code ...................................... 23
   A.1. Creating a UUIDv6 Value .............................. 23
   A.2. Creating a UUIDv7 Value .............................. 23
   A.3. Creating a UUIDv8 Value .............................. 25
Appendix B. Test Vectors ..................................... 26
   B.1. Example of a UUIDv6 Value ............................. 27
   B.2. Example of a UUIDv7 Value ............................. 28
   B.3. Example of a UUIDv8 Value ............................. 28
Appendix C. Version and Variant Tables ..................... 29
   C.1. Variant 10xx Versions ............................... 29
Authors’ Addresses .......................................... 30
1. Introduction

Many things have changed in the time since UUIDs were originally created. Modern applications have a need to create and utilize UUIDs as the primary identifier for a variety of different items in complex computational systems, including but not limited to database keys, file names, machine or system names, and identifiers for event-driven transactions.

One area UUIDs have gained popularity is as database keys. This stems from the increasingly distributed nature of modern applications. In such cases, "auto increment" schemes often used by databases do not work well, as the effort required to coordinate unique numeric identifiers across a network can easily become a burden. The fact that UUIDs can be used to create unique, reasonably short values in distributed systems without requiring synchronization makes them a good alternative, but UUID versions 1-5 lack certain other desirable characteristics:

1. Non-time-ordered UUID versions such as UUIDv4 have poor database index locality. Meaning new values created in succession are not close to each other in the index and thus require inserts to be performed at random locations. The negative performance effects of which on common structures used for this (B-tree and its variants) can be dramatic.

2. The 100-nanosecond, Gregorian epoch used in UUIDv1 timestamps is uncommon and difficult to represent accurately using a standard number format such as [IEEE754].

3. Introspection/parsing is required to order by time sequence; as opposed to being able to perform a simple byte-by-byte comparison.

4. Privacy and network security issues arise from using a MAC address in the node field of Version 1 UUIDs. Exposed MAC addresses can be used as an attack surface to locate machines and reveal various other information about such machines (minimally manufacturer, potentially other details). Additionally, with the advent of virtual machines and containers, MAC address uniqueness is no longer guaranteed.

5. Many of the implementation details specified in [RFC4122] involve trade offs that are neither possible to specify for all applications nor necessary to produce interoperable implementations.
6. [RFC4122] does not distinguish between the requirements for
generation of a UUID versus an application which simply stores
one, which are often different.

Due to the aforementioned issue, many widely distributed database
applications and large application vendors have sought to solve the
problem of creating a better time-based, sortable unique identifier
for use as a database key. This has lead to numerous implementations
over the past 10+ years solving the same problem in slightly
different ways.

While preparing this specification the following 16 different
implementations were analyzed for trends in total ID length, bit
Layout, lexical formatting/encoding, timestamp type, timestamp
format, timestamp accuracy, node format/components, collision
handling and multi-timestamp tick generation sequencing.

1. [ULID] by A. Feerasta
2. [LexicalUUID] by Twitter
3. [Snowflake] by Twitter
4. [Flake] by Boundary
5. [ShardingID] by Instagram
6. [KSUID] by Segment
7. [Elasticflake] by P. Pearcy
8. [FlakeID] by T. Pawlak
9. [Sonyflake] by Sony
10. [orderedUuid] by IT. Cabrera
11. [COMBGUID] by R. Tallent
12. [SID] by A. Chilton
13. [pushID] by Google
14. [XID] by O. Poitrey
15. [ObjectID] by MongoDB
16. [CUID] by E. Elliott

An inspection of these implementations and the issues described above
has led to this document which attempts to adapt UUIDs to address
these issues.

2. Terminology

2.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",
"SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and
"OPTIONAL" in this document are to be interpreted as described in BCP
14 [RFC2119] [RFC8174] when, and only when, they appear in all
capitals, as shown here.
2.2. Abbreviations

The following abbreviations are used in this document:

- **UUID**  Universally Unique Identifier [RFC4122]
- **CSPRNG**  Cryptographically Secure Pseudo-Random Number Generator
- **MAC**  Media Access Control
- **MSB**  Most Significant Bit
- **DBMS**  Database Management System

3. Summary of Changes

The following UUIDs are hereby introduced:

**UUID version 6 (UUIDv6)**
A re-ordering of UUID version 1 so it is sortable as an opaque sequence of bytes. Easy to implement given an existing UUIDv1 implementation. See Section 5.1

**UUID version 7 (UUIDv7)**
An entirely new time-based UUID bit layout sourced from the widely implemented and well known Unix Epoch timestamp source. See Section 5.2

**UUID version 8 (UUIDv8)**
A free-form UUID format which has no explicit requirements except maintaining backward compatibility. See Section 5.3

**Max UUID**
A specialized UUID which is the inverse of [RFC4122], Section 4.1.7 See Section 5.4

3.1. changelog

RFC EDITOR PLEASE DELETE THIS SECTION.

draft-04

- Fixed bad title in IEEE754 Normative Reference
- Fixed bad GMT offset in Test Vector Appendix
- Removed MAY in Counters section
- Condensed Counter Type into Counter Methods to reduce text
- Removed option for random increment along with fixed-length counter

Peabody & Davis Expires 25 December 2022 [Page 5]
- Described how to handle scenario where New UUID less than Old UUID
- Allow timestamp increment if counter overflows
- Replaced UUIDv8 C code snippet with full generation example
- Fixed RFC4086 Reference link
- Describe reseeding best practice for CSPRNG
- Changed MUST to SHOULD removing requirement for absolute monotonicity

draft-03

- Reworked the draft body to make the content more concise
- UUIDv6 section reworked to just the reorder of the timestamp
- UUIDv7 changed to simplify timestamp mechanism to just millisecond Unix timestamp
- UUIDv8 relaxed to be custom in all elements except version and variant
- Introduced Max UUID.
- Added C code samples in Appendix.
- Added test vectors in Appendix.
- Version and Variant section combined into one section.
- Changed from pseudo-random number generators to cryptographically secure pseudo-random number generator (CSPRNG).
- Combined redundant topics from all UUIDs into sections such as Timestamp granularity, Monotonicity and Counters, Collision Resistance, Sorting, and Unguessability, etc.
- Split Encoding and Storage into Opacity and DBMS and Database Considerations
- Reworked Global Uniqueness under new section Global and Local Uniqueness
- Node verbiage only used in UUIDv6 all others reference random/rand instead
- Clock sequence verbiage changed simply to counter in any section other than UUIDv6
- Added Abbreviations section
- Updated IETF Draft XML Layout
- Added information about little-endian UUIDs

draft-02

- Added Changelog
- Fixed misc. grammatical errors
- Fixed section numbering issue
- Fixed some UUIDvX reference issues
- Changed all instances of "motonic" to "monotonic"
- Changed all instances of "#-bit" to 
- Changed "proceeding" verbiage to "after" in section 7
- Added details on how to pad 32 bit Unix timestamp to 36 bits in UUIDv7
- Added details on how to truncate 64 bit Unix timestamp to 36 bits in UUIDv7
- Added forward reference and bullet to UUIDv8 if truncating 64 bit Unix Epoch is not an option.
- Fixed bad reference to non-existent "time_or_node" in section 4.5.4

Complete rewrite of entire document.
- The format, flow and verbiage used in the specification has been reworked to mirror the original RFC 4122 and current IETF standards.
- Removed the topics of UUID length modification, alternate UUID text formats, and alternate UUID encoding techniques.
- Research into 16 different historical and current implementations of time-based universal identifiers was completed at the end of 2020 in attempt to identify trends which have directly influenced design decisions in this draft document (https://github.com/uuid6/uuid6-ietf-draft/tree/master/research)
- Prototype implementation have been completed for UUIDv6, UUIDv7, and UUIDv8 in various languages by many GitHub community members. (https://github.com/uuid6/prototypes)

4. Variant and Version Fields

The variant bits utilized by UUIDs in this specification remain in the same octet as originally defined by [RFC4122], Section 4.1.1.

The next table details Variant 10xx (8/9/A/B) and the new versions defined by this specification. A complete guide to all versions within this variant has been includes in Appendix C.1.
Table 1: New UUID variant 10xx (8/9/A/B) versions defined by this specification

For UUID version 6, 7 and 8 the variant field placement from [RFC4122] are unchanged. An example version/variant layout for UUIDv6 follows the table where M is the version and N is the variant.

00000000-0000-6000-8000-000000000000
00000000-0000-6000-9000-000000000000
00000000-0000-6000-A000-000000000000
00000000-0000-6000-B000-000000000000
xxxxxxxx-xxxx-Mxxx-Nxxx-xxxxxxxxxxxx

Figure 1: UUIDv6 Variant Examples

5. New Formats

The UUID format is 16 octets; the variant bits in conjunction with the version bits described in the next section in determine finer structure.

5.1. UUID Version 6

UUID version 6 is a field-compatible version of UUIDv1, reordered for improved DB locality. It is expected that UUIDv6 will primarily be used in contexts where there are existing v1 UUIDs. Systems that do not involve legacy UUIDv1 SHOULD consider using UUIDv7 instead.

Instead of splitting the timestamp into the low, mid and high sections from UUIDv1, UUIDv6 changes this sequence so timestamp bytes are stored from most to least significant. That is, given a 60 bit timestamp value as specified for UUIDv1 in [RFC4122], Section 4.1.4,
for UUIDv6, the first 48 most significant bits are stored first, followed by the 4 bit version (same position), followed by the remaining 12 bits of the original 60 bit timestamp.

The clock sequence bits remain unchanged from their usage and position in [RFC4122], Section 4.1.5.

The 48 bit node SHOULD be set to a pseudo-random value however implementations MAY choose to retain the old MAC address behavior from [RFC4122], Section 4.1.6 and [RFC4122], Section 4.5. For more information on MAC address usage within UUIDs see the Section 8

The format for the 16-byte, 128 bit UUIDv6 is shown in Figure 1

0                   1                   2                   3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           time_high                           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|           time_mid            |      time_low_and_version     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|clk_seq_hi_res |  clk_seq_low |         node (0-1)            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                         node (2-5)                            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Figure 2: UUIDv6 Field and Bit Layout

time_high:
The most significant 32 bits of the 60 bit starting timestamp. Occupies bits 0 through 31 (octets 0-3)

time_mid:
The middle 16 bits of the 60 bit starting timestamp. Occupies bits 32 through 47 (octets 4-5)

time_low_and_version:
The first four most significant bits MUST contain the UUIDv6 version (0110) while the remaining 12 bits will contain the least significant 12 bits from the 60 bit starting timestamp. Occupies bits 48 through 63 (octets 6-7)

clk_seq_hi_res:
The first two bits MUST be set to the UUID variant (10) The remaining 6 bits contain the high portion of the clock sequence. Occupies bits 64 through 71 (octet 8)
clock_seq_low:
The 8 bit low portion of the clock sequence. Occupies bits 72 through 79 (octet 9)

node:
48 bit spatially unique identifier Occupies bits 80 through 127 (octets 10-15)

With UUIDv6 the steps for splitting the timestamp into time_high and time_mid are OPTIONAL since the 48 bits of time_high and time_mid will remain in the same order. An extra step of splitting the first 48 bits of the timestamp into the most significant 32 bits and least significant 16 bits proves useful when reusing an existing UUIDv1 implementation.

5.2. UUID Version 7

UUID version 7 features a time-ordered value field derived from the widely implemented and well known Unix Epoch timestamp source, the number of milliseconds seconds since midnight 1 Jan 1970 UTC, leap seconds excluded. As well as improved entropy characteristics over versions 1 or 6.

Implementations SHOULD utilize UUID version 7 over UUID version 1 and 6 if possible.

```
+---------------------------------------------+---------------------------------------------+---------------------------------------------+---------------------------------------------+
| unix_ts_ms                                                                                           | ver                                                                 | rand_a                                                                 |
| unix_ts_ms                                                                                           |                                                                       | rand_b                                                                 |
| var                                                                                                 |                                                                       |                                                                       |
+---------------------------------------------+---------------------------------------------+---------------------------------------------+---------------------------------------------+
```

Figure 3: UUIDv7 Field and Bit Layout

unix_ts_ms:
48 bit big-endian unsigned number of Unix epoch timestamp as per Section 6.1.

ver:
4 bit UUIDv7 version set as per Section 4
rand_a:
12 bits pseudo-random data to provide uniqueness as per
Section 6.2 and Section 6.6.

var:
The 2 bit variant defined by Section 4.

rand_b:
The final 62 bits of pseudo-random data to provide uniqueness as
per Section 6.2 and Section 6.6.

5.3. UUID Version 8

UUID version 8 provides an RFC-compatible format for experimental or
vendor-specific use cases. The only requirement is that the variant
and version bits MUST be set as defined in Section 4. UUIDv8’s
uniqueness will be implementation-specific and SHOULD NOT be assumed.

The only explicitly defined bits are the Version and Variant leaving
122 bits for implementation specific time-based UUIDs. To be clear:
UUIDv8 is not a replacement for UUIDv4 where all 122 extra bits are
filled with random data.

Some example situations in which UUIDv8 usage could occur:

* An implementation would like to embed extra information within the
UUID other than what is defined in this document.

* An implementation has other application/language restrictions
which inhibit the use of one of the current UUIDs.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

+------------------------------------------------------------------------|
<p>| custom_a | |
|------------------------------------------------------------------------|
| custom_a | ver | custom_b |
|------------------------------------------------------------------------|
| var| custom_c |
|------------------------------------------------------------------------|</p>
<table>
<thead>
<tr>
<th>custom_c</th>
</tr>
</thead>
</table>

Figure 4: UUIDv8 Field and Bit Layout

custom_a:
The first 48 bits of the layout that can be filled as an
implementation sees fit.
The 4 bit version field as defined by Section 4

12 more bits of the layout that can be filled as an implementation sees fit.

The 2 bit variant field as defined by Section 4.

The final 62 bits of the layout immediately following the var field to be filled as an implementation sees fit.

The Max UUID is special form of UUID that is specified to have all 128 bits set to 1. This UUID can be thought of as the inverse of Nil UUID defined in [RFC4122], Section 4.1.7

FFFFFFFF-FFFF-FFFF-FFFF-FFFFFFFFFFFF

Figure 5: Max UUID Format

The minimum requirements for generating UUIDs are described in this document for each version. Everything else is an implementation detail and up to the implementer to decide what is appropriate for a given implementation. That being said, various relevant factors are covered below to help guide an implementer through the different trade-offs among differing UUID implementations.

6.1. Timestamp Granularity

UUID timestamp source, precision and length was the topic of great debate while creating this specification. As such choosing the right timestamp for your application is a very important topic. This section will detail some of the most common points on this topic.
Reliability:
Implementations SHOULD use the current timestamp from a reliable source to provide values that are time-ordered and continually increasing. Care SHOULD be taken to ensure that timestamp changes from the environment or operating system are handled in a way that is consistent with implementation requirements. For example, if it is possible for the system clock to move backward due to either manual adjustment or corrections from a time synchronization protocol, implementations must decide how to handle such cases. (See Altering, Fuzzing, or Smearing bullet below.)

Source:
UUID version 1 and 6 both utilize a Gregorian epoch timestamp while UUIDv7 utilizes a Unix Epoch timestamp. If other timestamp sources or a custom timestamp epoch are required UUIDv8 SHOULD be leveraged.

Sub-second Precision and Accuracy:
Many levels of precision exist for timestamps: milliseconds, microseconds, nanoseconds, and beyond. Additionally fractional representations of sub-second precision may be desired to mix various levels of precision in a time-ordered manner. Furthermore, system clocks themselves have an underlying granularity and it is frequently less than the precision offered by the operating system. With UUID version 1 and 6, 100-nanoseconds of precision are present while UUIDv7 features fixed millisecond level of precision within the Unix epoch that does not exceed the granularity capable in most modern systems. For other levels of precision UUIDv8 SHOULD be utilized.

Length:
The length of a given timestamp directly impacts how long a given UUID will be valid. That is, how many timestamp ticks can be contained in a UUID before the maximum value for the timestamp field is reached. Care should be given to ensure that the proper length is selected for a given timestamp. UUID version 1 and 6 utilize a 60 bit timestamp and UUIDv7 features a 48 bit timestamp.

Alterating, Fuzzing, or Smearing:
Implementations MAY alter the actual timestamp. Some examples included security considerations around providing a real clock value within a UUID, to correct inaccurate clocks or to handle leap seconds. This specification makes no requirement or guarantee about how close the clock value needs to be to actual time.
Padding:
When timestamp padding is required, implementations MUST pad the most significant bits (left-most) bits with zeros. An example is padding the most significant, left-most bits of a 32 bit Unix timestamp with zero’s to fill out the 48 bit timestamp in UUIDv7.

Truncating:
Similarly, when timestamps need to be truncated: the lower, least significant bits MUST be used. An example would be truncating a 64 bit Unix timestamp to the least significant, right-most 48 bits for UUIDv7.

6.2. Monotonicity and Counters

Monotonicity is the backbone of time-based sortable UUIDs. Naturally time-based UUIDs from this document will be monotonic due to an embedded timestamp however implementations can guarantee additional monotonicity via the concepts covered in this section.

Additionally, care SHOULD be taken to ensure UUIDs generated in batches are also monotonic. That is, if one-thousand UUIDs are generated for the same timestamp; there is sufficient logic for organizing the creation order of those one-thousand UUIDs. For batch UUID creation implementations MAY utilize a monotonic counter which SHOULD increment for each UUID created during a given timestamp.

For single-node UUID implementations that do not need to create batches of UUIDs, the embedded timestamp within UUID version 1, 6, and 7 can provide sufficient monotonicity guarantees by simply ensuring that timestamp increments before creating a new UUID. For the topic of Distributed Nodes please refer to Section 6.3

Implementations SHOULD choose one method for single-node UUID implementations that require batch UUID creation.
Fixed-Length Dedicated Counter Bits (Method 1):
This references the practice of allocating a specific number of bits in the UUID layout to the sole purpose of tallying the total number of UUIDs created during a given UUID timestamp tick. Positioning of a fixed bit-length counter SHOULD be immediately after the embedded timestamp. This promotes sortability and allows random data generation for each counter increment. With this method rand_a section of UUIDv7 SHOULD be utilized as fixed-length dedicated counter bits that are incremented by one for every UUID generation. The trailing random bits generated for each new UUID in rand_b can help produce unguessable UUIDs. In the event more counter bits are required the most significant, left-most, bits of rand_b MAY be leveraged as additional counter bits.

Monotonic Random (Method 2):
With this method the random data is extended to also double as a counter. This monotonic random can be thought of as a "randomly seeded counter" which MUST be incremented in the least significant position for each UUID created on a given timestamp tick. UUIDv7’s rand_b section SHOULD be utilized with this method to handle batch UUID generation during a single timestamp tick. The increment value for every UUID generation SHOULD be a random integer of any desired length larger than zero. It ensures the UUIDs retain the required level of unguessability characters provided by the underlying entropy. The increment value MAY be one when the amount of UUIDs generated in a particular period of time is important and guessability is not an issue. However, it SHOULD NOT be used by implementations that favor unguessibility, as the resulting values are easily guessable.

The following sub-topics cover topics related solely with creating reliable fixed-length dedicated counters:

Fixed-Length Dedicated Counter Seeding:
Implementations utilizing fixed-length counter method SHOULD randomly initialize the counter with each new timestamp tick. However, when the timestamp has not incremented; the counter SHOULD be frozen and incremented via the desired increment logic. When utilizing a randomly seeded counter alongside Method 1; the random MAY be regenerated with each counter increment without impacting sortability. The downside is that Method 1 is prone to overflows if a counter of adequate length is not selected or the random data generated leaves little room for the required number of increments. Implementations utilizing fixed-length counter method MAY also choose to randomly initialize a portion counter rather than the entire counter. For example, a 24 bit counter could have the 23 bits in least-significant, right-most, position
randomly initialized. The remaining most significant, left-most counter bits are initialized as zero for the sole purpose of guarding against counter rollovers.

Fixed-Length Dedicated Counter Length:
Care MUST be taken to select a counter bit-length that can properly handle the level of timestamp precision in use. For example, millisecond precision SHOULD require a larger counter than a timestamp with nanosecond precision. General guidance is that the counter SHOULD be at least 12 bits but no longer than 42 bits. Care SHOULD also be given to ensure that the counter length selected leaves room for sufficient entropy in the random portion of the UUID after the counter. This entropy helps improve the unguessability characteristics of UUIDs created within the batch.

The following sub-topics cover rollover handling with either type of counter method:

Counter Rollover Guards:
The technique from Fixed-Length Dedicated Counter Seeding which describes allocating a segment of the fixed-length counter as a rollover guard is also helpful to mitigate counter rollover issues. This same technique can be leveraged with Monotonic random counter methods by ensuring the total length of a possible increment in the least significant, right most position is less than the total length of the random being incremented. As such the most significant, left-most, bits can be incremented as rollover guarding.

Counter Rollover Handling:
Counter rollovers SHOULD be handled by the application to avoid sorting issues. The general guidance is that applications that care about absolute monotonicity and sortability SHOULD freeze the counter and wait for the timestamp to advance which ensures monotonicity is not broken. Alternatively, implementations MAY increment the timestamp ahead of the actual time and reinitialize the counter.

Implementations MAY use the following logic to ensure UUIDs featuring embedded counters are monotonic in nature:

1. Compare the current timestamp against the previously stored timestamp.
2. If the current timestamp is equal to the previous timestamp; increment the counter according to the desired method.
3. If the current timestamp is greater than the previous timestamp; re-initialize the desired counter method to the new timestamp and generate new random bytes (if the bytes were frozen or being used as the seed for a monotonic counter).

Implementations SHOULD check if the currently generated UUID is greater than the previously generated UUID. If this is not the case then any number of things could have occurred. Such as, but not limited to, clock rollbacks, leap second handling or counter rollovers. Applications SHOULD embed sufficient logic to catch these scenarios and correct the problem ensuring the next UUID generated is greater than the previous. To handle this scenario, the general guidance is that application MAY reuse the previous timestamp and increment the previous counter method.

6.3. Distributed UUID Generation

Some implementations MAY desire to utilize multi-node, clustered, applications which involve two or more nodes independently generating UUIDs that will be stored in a common location. While UUIDs already feature sufficient entropy to ensure that the chances of collision are low as the total number of nodes increase; so does the likelihood of a collision. This section will detail the approaches that MAY be utilized by multi-node UUID implementations in distributed environments.

Centralized Registry:

With this method all nodes tasked with creating UUIDs consult a central registry and confirm the generated value is unique. As applications scale the communication with the central registry could become a bottleneck and impact UUID generation in a negative way. Utilization of shared knowledge schemes with central/global registries is outside the scope of this specification.

Node IDs:

With this method, a pseudo-random Node ID value is placed within the UUID layout. This identifier helps ensure the bit-space for a given node is unique, resulting in UUIDs that do not conflict with any other UUID created by another node with a different node id. Implementations that choose to leverage an embedded node id SHOULD utilize UUIDv8. The node id SHOULD NOT be an IEEE 802 MAC address as per Section 8. The location and bit length are left to implementations and are outside the scope of this specification. Furthermore, the creation and negotiation of unique node ids among nodes is also out of scope for this specification.
Utilization of either a Centralized Registry or Node ID are not required for implementing UUIDs in this specification. However implementations SHOULD utilize one of the two aforementioned methods if distributed UUID generation is a requirement.

6.4. Collision Resistance

Implementations SHOULD weigh the consequences of UUID collisions within their application and when deciding between UUID versions that use entropy (random) versus the other components such as Section 6.1 and Section 6.2. This is especially true for distributed node collision resistance as defined by Section 6.3.

There are two example scenarios below which help illustrate the varying seriousness of a collision within an application.

**Low Impact**

A UUID collision generated a duplicate log entry which results in incorrect statistics derived from the data. Implementations that are not negatively affected by collisions may continue with the entropy and uniqueness provided by the traditional UUID format.

**High Impact:**

A duplicate key causes an airplane to receive the wrong course which puts people’s lives at risk. In this scenario there is no margin for error. Collisions MUST be avoided and failure is unacceptable. Applications dealing with this type of scenario MUST employ as much collision resistance as possible within the given application context.

6.5. Global and Local Uniqueness

UUIDs created by this specification MAY be used to provide local uniqueness guarantees. For example, ensuring UUIDs created within a local application context are unique within a database MAY be sufficient for some implementations where global uniqueness outside of the application context, in other applications, or around the world is not required.

Although true global uniqueness is impossible to guarantee without a shared knowledge scheme; a shared knowledge scheme is not required by UUID to provide uniqueness guarantees. Implementations MAY implement a shared knowledge scheme introduced in Section 6.3 as they see fit to extend the uniqueness guaranteed this specification and [RFC4122].
6.6. Unguessability

Implementations SHOULD utilize a cryptographically secure pseudo-random number generator (CSPRNG) to provide values that are both difficult to predict ("unguessable") and have a low likelihood of collision ("unique"). Care SHOULD be taken to ensure the CSPRNG state is properly reseeded upon state changes, such as process forks, to ensure proper CSPRNG operation. CSPRNG ensures the best of Section 6.4 and Section 8 are present in modern UUIDs.

Advice on generating cryptographic-quality random numbers can be found in [RFC4086]

6.7. Sorting

UUIDv6 and UUIDv7 are designed so that implementations that require sorting (e.g. database indexes) SHOULD sort as opaque raw bytes, without need for parsing or introspection.

Time ordered monotonic UUIDs benefit from greater database index locality because the new values are near each other in the index. As a result objects are more easily clustered together for better performance. The real-world differences in this approach of index locality vs random data inserts can be quite large.

UUIDs formats created by this specification SHOULD be Lexicographically sortable while in the textual representation.

UUIDs created by this specification are crafted with big-ending byte order (network byte order) in mind. If Little-endian style is required a custom UUID format SHOULD be created using UUIDv8.

6.8. Opacity

UUIDs SHOULD be treated as opaque values and implementations SHOULD NOT examine the bits in a UUID to whatever extent is possible. However, where necessary, inspectors should refer to Section 4 for more information on determining UUID version and variant.

6.9. DBMS and Database Considerations

For many applications, such as databases, storing UUIDs as text is unnecessarily verbose, requiring 288 bits to represent 128 bit UUID values. Thus, where feasible, UUIDs SHOULD be stored within database applications as the underlying 128 bit binary value.

For other systems, UUIDs MAY be stored in binary form or as text, as appropriate. The trade-offs to both approaches are as such:
* Storing as binary requires less space and may result in faster data access.
* Storing as text requires more space but may require less translation if the resulting text form is to be used after retrieval and thus maybe simpler to implement.

DBMS vendors are encouraged to provide functionality to generate and store UUID formats defined by this specification for use as identifiers or left parts of identifiers such as, but not limited to, primary keys, surrogate keys for temporal databases, foreign keys included in polymorphic relationships, and keys for key-value pairs in JSON columns and key-value databases. Applications using a monolithic database may find using database-generated UUIDs (as opposed to client-generate UUIDs) provides the best UUID monotonicity. In addition to UUIDs, additional identifiers MAY be used to ensure integrity and feedback.

7. IANA Considerations

This document has no IANA actions.

8. Security Considerations

MAC addresses pose inherent security risks and SHOULD not be used within a UUID. Instead CSPRNG data SHOULD be selected from a source with sufficient entropy to ensure guaranteed uniqueness among UUID generation. See Section 6.6 for more information.

Timestamps embedded in the UUID do pose a very small attack surface. The timestamp in conjunction with an embedded counter does signal the order of creation for a given UUID and it’s corresponding data but does not define anything about the data itself or the application as a whole. If UUIDs are required for use with any security operation within an application context in any shape or form then [RFC4122] UUIDv4 SHOULD be utilized.

9. Acknowledgements

The authors gratefully acknowledge the contributions of Ben Campbell, Ben Ramsey, Fabio Lima, Gonzalo Salgueiro, Martin Thomson, Murray S. Kucherawy, Rick van Rein, Rob Wilton, Sean Leonard, Theodore Y. Ts’o., Robert Kieffer, sergeyprokhorenko, LiosK As well as all of those in the IETF community and on GitHub to who contributed to the discussions which resulted in this document.

10. Normative References
11. Informative References


[Snowflake] Twitter, "Snowflake is a network service for generating unique ID numbers at high scale with some simple guarantees.", Commit b3f6a3c, May 2014, <https://github.com/twitter-archive/snowflake/releases/tag/snowflake-2010>.


[Elasticflake]


Appendix A. Example Code

A.1. Creating a UUIDv6 Value

This section details a function in C which converts from a UUID version 1 to version 6:

```c
#include <stdio.h>
#include <stdint.h>
#include <inttypes.h>
#include <arpa/inet.h>
#include <uuid/uuid.h>

/* Converts UUID version 1 to version 6 in place. */
void uuidv1tov6(uuid_t u) {
    uint64_t ut;
    unsigned char *up = (unsigned char *)u;

    // load ut with the first 64 bits of the UUID
    ut = ((uint64_t)ntohl(*((uint32_t*)up))) << 32;
    ut |= ((uint64_t)ntohl(*((uint32_t*)&up[4])));

    // dance the bit-shift...
    ut =
        ((ut >> 32) & 0x0FFF) // 12 least significant bits
        (0x6000) // version number
        ((ut >> 28) & 0x00000000FFFF0000) // next 20 bits
        ((ut << 20) & 0x000FFFF000000000) // next 16 bits
        (ut << 52); // 12 most significant bits

    // store back in UUID
    *((uint32_t*)up) = htonl((uint32_t)(ut >> 32));
    *((uint32_t*)&up[4]) = htonl((uint32_t)(ut));
}
```

Figure 6: UUIDv6 Function in C

A.2. Creating a UUIDv7 Value
```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <time.h>

// ... 

// csprng data source
FILE *rndf;
rndf = fopen("/dev/urandom", "r");
if (rndf == 0) {
    printf("fopen /dev/urandom error\n");
    return 1;
}

// ... 

// generate one UUIDv7E
uint8_t u[16];
struct timespec ts;
int ret;

ret = clock_gettime(CLOCK_REALTIME, &ts);
if (ret != 0) {
    printf("clock_gettime error: %d\n", ret);
    return 1;
}

uint64_t tms;
tms = ((uint64_t)ts.tv_sec) * 1000;
tms += ((uint64_t)ts.tv_nsec) / 1000000;
memset(u, 0, 16);

fread(&u[6], 10, 1, rndf); // fill everything after the timestamp with random bytes

*((uint64_t*)(u)) |= htonl(tms << 16); // shift time into first 48 bits and OR into place
u[8] = 0x80 | (u[8] & 0x3F); // set variant field, top two bits are 1, 0
u[6] = 0x70 | (u[6] & 0x0F); // set version field, top four bits are 0, 1, 1, 1

Figure 7: UUIDv7 Function in C
```
A.3. Creating a UUIDv8 Value

UUIDv8 will vary greatly from implementation to implementation.

The following example utilizes:

- 32 bit custom-epoch timestamp (seconds elapsed since 2020-01-01 00:00:00 UTC)
- 16 bit exotic resolution (~15 microsecond) subsecond timestamp encoded using the fractional representation
- 58 bit random number
- 8 bit application-specific unique node ID
- 8 bit rolling sequence number
#include <stdint.h>
#include <time.h>

int get_random_bytes(uint8_t *buffer, int count) {
    // ...
}

int generate_uuidv8(uint8_t *uuid, uint8_t node_id) {
    struct timespec tp;
    if (clock_gettime(CLOCK_REALTIME, &tp) != 0)
        return -1; // real-time clock error
    // 32 bit biased timestamp (seconds elapsed since 2020-01-01 00:00:00 UTC)
    uint32_t timestamp_sec = tp.tv_sec - 1577836800;
    uuid[0] = timestamp_sec >> 24;
    uuid[1] = timestamp_sec >> 16;
    uuid[2] = timestamp_sec >> 8;
    uuid[3] = timestamp_sec;
    // 16 bit subsecond fraction (~15 microsecond resolution)
    uint16_t timestamp_subsec = ((uint64_t)tp.tv_nsec << 16) / 1000000000;
    uuid[4] = timestamp_subsec >> 8;
    uuid[5] = timestamp_subsec;
    // 58 bit random number and required ver and var fields
    if (get_random_bytes(&uuid[6], 8) != 0)
        return -1; // random number generator error
    uuid[6] = 0x80 | (uuid[6] & 0x0f);
    uuid[8] = 0x80 | (uuid[8] & 0x3f);
    // 8 bit application-specific node ID to guarantee application-wide uniqueness
    uuid[14] = node_id;
    // 8 bit rolling sequence number to help ensure process-wide uniqueness
    static uint8_t sequence = 0;
    uuid[15] = sequence++; // NOTE: unprotected from race conditions
    return 0;
}

Figure 8: UUIDv8 Function in C

Appendix B. Test Vectors

Both UUIDv1 and UUIDv6 test vectors utilize the same 60 bit
timestamp: 0x1EC9414C232AB00 (138648505420000000) Tuesday, February
22, 2022 2:22:22.000000 PM GMT-05:00
Both UUIDv1 and UUIDv6 utilize the same values in clk_seq_hi_res, clock_seq_low, and node. All of which have been generated with random data.

```plaintext
# Unix Nanosecond precision to Gregorian 100-nanosecond intervals
gregorian_100_ns = (Unix_64_bit_nanoseconds / 100) + gregorian_Unix_offset

# Gregorian to Unix Offset:
# The number of 100-ns intervals between the
# UUID epoch 1582-10-15 00:00:00 and the Unix epoch 1970-01-01 00:00:00.
# gregorian_Unix_offset = 0x01b21dd213814000 or 122192928000000000

# Unix 64 bit Nanosecond Timestamp:
# Unix NS: Tuesday, February 22, 2022 2:22:22 PM GMT-05:00
# Unix_64_bit_nanoseconds = 0x16D6320C3D4DCC00 or 1645557742000000000

# Work:
# gregorian_100_ns = (1645557742000000000 / 100) + 122192928000000000
# (138648505420000000 - 122192928000000000) * 100 = Unix_64_bit_nanoseconds

# Final:
# gregorian_100_ns = 0x1EC9414C232AB00 or 1386485054200000000

# Original: 00111110110010010100000101001100001000110010101100000000
# UUIDv1:  11000010001100101010101100000000|1001010000010100|0001|00011110111
# UUIDv6:  00111110110010010100000101001100|0010001100101010|0110|10110000000

Figure 9: Test Vector Timestamp Pseudo-code

B.1. Example of a UUIDv6 Value

```

---
<table>
<thead>
<tr>
<th>field</th>
<th>bits</th>
<th>value_hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>time_low</td>
<td>32</td>
<td>0xC232AB00</td>
</tr>
<tr>
<td>time_mid</td>
<td>16</td>
<td>0x9414</td>
</tr>
<tr>
<td>time_hi_and_version</td>
<td>16</td>
<td>0x11EC</td>
</tr>
<tr>
<td>clk_seq_hi_res</td>
<td>8</td>
<td>0xB3</td>
</tr>
<tr>
<td>clock_seq_low</td>
<td>8</td>
<td>0xC8</td>
</tr>
<tr>
<td>node</td>
<td>48</td>
<td>0x9E6BDECD846</td>
</tr>
</tbody>
</table>
---
```

```
| total | 128  |
---
```

final_hex: C232AB00-9414-11EC-B3C8-9E6BDECD846

Figure 10: UUIDv1 Example Test Vector

Peabody & Davis Expires 25 December 2022
B.2. Example of a UUIDv7 Value

This example UUIDv7 test vector utilizes a well-known 32 bit Unix epoch with additional millisecond precision to fill the first 48 bits rand_a and rand_b are filled with random data.

The timestamp is Tuesday, February 22, 2022 2:22:22.00 PM GMT-05:00 represented as 0x17F22E279B0 or 1645557742000

```
<table>
<thead>
<tr>
<th>field</th>
<th>bits</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>unix_ts_ms</td>
<td>48</td>
<td>0x17F22E279B0</td>
</tr>
<tr>
<td>var</td>
<td>4</td>
<td>0x7</td>
</tr>
<tr>
<td>rand_a</td>
<td>12</td>
<td>0xCC3</td>
</tr>
<tr>
<td>var</td>
<td>2</td>
<td>b10</td>
</tr>
<tr>
<td>rand_b</td>
<td>62</td>
<td>0x18C4DC0C0C07398F</td>
</tr>
</tbody>
</table>
```

```
total: 128
```

```
final: 017F22E2-79B0-7CC3-98C4-DC0C0C07398F
```

Figure 12: UUIDv7 Example Test Vector

B.3. Example of a UUIDv8 Value

This example UUIDv8 test vector utilizes a well-known 64 bit Unix epoch with nanosecond precision, truncated to the least-significant, right-most, bits to fill the first 48 bits through version.

```
<table>
<thead>
<tr>
<th>field</th>
<th>bits</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>unix_ts_ms</td>
<td>48</td>
<td>0x17F22E279B0</td>
</tr>
<tr>
<td>var</td>
<td>4</td>
<td>0x7</td>
</tr>
<tr>
<td>rand_a</td>
<td>12</td>
<td>0xCC3</td>
</tr>
<tr>
<td>var</td>
<td>2</td>
<td>b10</td>
</tr>
<tr>
<td>rand_b</td>
<td>62</td>
<td>0x18C4DC0C0C07398F</td>
</tr>
</tbody>
</table>
```

```
total: 128
```

```
final: 017F22E2-79B0-7CC3-98C4-DC0C0C07398F
```

Figure 13: UUIDv8 Example Test Vector
The next two segments of custom_b and custom_c are filled with random data.

Timestamp is Tuesday, February 22, 2022 2:22:22.000000 PM GMT-05:00 represented as 0x16D6320C3D4DCC00 or 1645557742000000000

It should be noted that this example is just to illustrate one scenario for UUIDv8. Test vectors will likely be implementation specific and vary greatly from this simple example.

---

<table>
<thead>
<tr>
<th>field</th>
<th>bits</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>custom_a</td>
<td>48</td>
<td>0x320C3D4DCC00</td>
</tr>
<tr>
<td>ver</td>
<td>4</td>
<td>0x8</td>
</tr>
<tr>
<td>custom_b</td>
<td>12</td>
<td>0x75B</td>
</tr>
<tr>
<td>var</td>
<td>2</td>
<td>b10</td>
</tr>
<tr>
<td>custom_c</td>
<td>62</td>
<td>0xEC932D5F69181C0</td>
</tr>
</tbody>
</table>

---

final: 320C3D4D-CC00-875B-8EC9-32D5F69181C0

Figure 13: UUIDv8 Example Test Vector

Appendix C. Version and Variant Tables

C.1. Variant 10xx Versions

<table>
<thead>
<tr>
<th>Msb0</th>
<th>Msb1</th>
<th>Msb2</th>
<th>Msb3</th>
<th>Version</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Unused</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>The Gregorian time-based UUID from in [RFC4122], Section 4.1.3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>DCE Security version, with embedded POSIX UIDs from [RFC4122], Section 4.1.3</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>The name-based version specified in [RFC4122], Section 4.1.3 that uses MD5 hashing.</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>The randomly or pseudo-</td>
</tr>
</tbody>
</table>
### Table 2: All UUID variant 10xx (8/9/A/B) version definitions.

<table>
<thead>
<tr>
<th>Variant</th>
<th>Bit 0</th>
<th>Bit 1</th>
<th>Bit 2</th>
<th>Bit 3</th>
<th>Bit 4</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 1 0 1 5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>randomly generated version specified in [RFC4122], Section 4.1.3.</td>
</tr>
<tr>
<td>0 1 1 0 6</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>6</td>
<td>The name-based version specified in [RFC4122], Section 4.1.3 that uses SHA-1 hashing.</td>
</tr>
<tr>
<td>0 1 1 1 7</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>7</td>
<td>Reordered Gregorian time-based UUID specified in this document.</td>
</tr>
<tr>
<td>1 0 0 0 8</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>8</td>
<td>Unix Epoch time-based UUID specified in this document.</td>
</tr>
<tr>
<td>1 0 0 1 9</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>9</td>
<td>Reserved for custom UUID formats specified in this document.</td>
</tr>
<tr>
<td>1 0 1 0 10</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>10</td>
<td>Reserved for future definition.</td>
</tr>
<tr>
<td>1 0 1 1 11</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>11</td>
<td>Reserved for future definition.</td>
</tr>
<tr>
<td>1 1 0 0 12</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>12</td>
<td>Reserved for future definition.</td>
</tr>
<tr>
<td>1 1 0 1 13</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>13</td>
<td>Reserved for future definition.</td>
</tr>
<tr>
<td>1 1 1 0 14</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>14</td>
<td>Reserved for future definition.</td>
</tr>
<tr>
<td>1 1 1 1 15</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>15</td>
<td>Reserved for future definition.</td>
</tr>
</tbody>
</table>

Authors’ Addresses

Brad G. Peabody