Workgroup: dispatch
Internet-Draft:
draft-peabody-dispatch-new-uuid-format-01
Updates: 4122 (if approved)
Published: 26 April 2021
Intended Status: Standards Track
Expires: 28 October 2021
Authors: BGP. Peabody K. Davis

New UUID Formats

Abstract

This document presents new time-based UUID formats which are suited for use as a database key.

A common case for modern applications is to create a unique identifier for use as a primary key in a database table. This identifier usually implements an embedded timestamp that is sortable using the monotonic creation time in the most significant bits. In addition the identifier is highly collision resistant, difficult to guess, and provides minimal security attack surfaces. None of the existing UUID versions, including UUIDv1, fulfill each of these requirements in the most efficient possible way. This document is a proposal to update [RFC4122] with three new UUID versions that address these concerns, each with different trade-offs.

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 28 October 2021.

Copyright Notice

Table of Contents

1. Introduction

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

2. Background

A lot of things have changed in the time since UUIDs were originally created. Modern applications have a need to use (and many have already implemented) UUIDs as database primary keys.

The motivation for using UUIDs as database keys stems primarily from the fact that applications are increasingly distributed in nature. Simplistic "auto increment" schemes with integers in sequence do not work well in a distributed system since the effort required to synchronize such numbers across a network can easily become a burden. The fact that UUIDs can be used to create unique and reasonably short values in distributed systems without requiring synchronization makes them a good candidate for use as a database key in such environments.

However some properties of [RFC4122] UUIDs are not well suited to this task. First, most of the existing 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. As such newly inserted values SHOULD be time-ordered to address this.

While it is true that UUIDv1 does contain an embedded timestamp and can be time-ordered; UUIDv1 has other issues. It is possible to sort Version 1 UUIDs by time but it is a laborious task. The process requires breaking the bytes of the UUID into various pieces, re-ordering the bits, and then determining the order from the reconstructed timestamp. This is not efficient in very large systems. Implementations would be simplified with a sort order where the UUID can simply be treated as an opaque sequence of bytes and ordered as such.

After the embedded timestamp, the remaining 64 bits are in essence used to provide uniqueness both on a global scale and within a given timestamp tick. The clock sequence value ensures that when multiple UUIDs are generated for the same timestamp value are given a monotonic sequence value. This explicit sequencing helps further facilitate sorting. The remaining random bits ensure collisions are minimal.

Furthermore, UUIDv1 utilizes a non-standard timestamp epoch derived from the Gregorian Calendar. More specifically, the Coordinated Universal Time (UTC) as a count of 100-nanosecond intervals since 00:00:00.00, 15 October 1582. Implementations and many languages may find it easier to implement the widely adopted and well known Unix

Epoch, a custom epoch, or another timestamp source with various levels of timestamp precision required by the application.

Lastly, 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). Instead "cryptographically secure" pseudo-random number generators (CSPRNGs) or pseudo-random number generators (PRNG) SHOULD be used within an application context to provide uniqueness and unguessability.

Due to the shortcomings of UUIDv1 and UUIDv4 details so far, 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. [LexicalUUID] by Twitter
2. [Snowflake] by Twitter
3. [Flake] by Boundary
4. [ShardingID] by Instagram
5. [KSUID] by Segment
6. [Elasticflake] by P. Pearcy
7. [FlakeID] by T. Pawlak
8. [Sonyflake] by Sony
9. [orderedUuid] by IT. Cabrera
10. [COMBGUID] by R. Tallent
11. [ULID] by A. Feerasta
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 details the following trends that help define this standard:

- Timestamps MUST be k-sortable. That is, values within or close to the same timestamp are ordered properly by sorting algorithms.
- Timestamps SHOULD be big-endian with the most-significant bits of the time embedded as-is without reordering.

- Timestamps SHOULD utilize millisecond precision and Unix Epoch as timestamp source. Although, there is some variation to this among implementations depending on the application requirements.
- The ID format SHOULD be Lexicographically sortable while in the textual representation.
- IDs MUST ensure proper embedded sequencing to facilitate sorting when multiple UUIDs are created during a given timestamp.
- IDs MUST NOT require unique network identifiers as part of achieving uniqueness.
- Distributed nodes MUST be able to create collision resistant Unique IDs without a consulting a centralized resource.

3. Summary of Changes

In order to solve these challenges this specification introduces three new version identifiers assigned for time-based UUIDs.

The first, UUIDv6, aims to be the easiest to implement for applications which already implement UUIDv1. The UUIDv6 specification keeps the original Gregorian timestamp source but does not reorder the timestamp bits as per the process utilized by UUIDv1. UUIDv6 also requires that pseudo-random data MUST be used in place of the MAC address. The rest of the UUIDv1 format remains unchanged in UUIDv6. See Section 4.3

Next, UUIDv7 introduces an entirely new time-based UUID bit layout utilizing a variable length timestamp sourced from the widely implemented and well known Unix Epoch timestamp source. The timestamp is broken into a 36-bit integer sections part, and is followed by a field of variable length which represents the sub-second timestamp portion, encoded so that each bit from most to least significant adds more precision. See Section 4.4

Finally, UUIDv8 introduces a relaxed time-based UUID format that caters to application implementations that cannot utilize UUIDv1, UUIDv6, or UUIDv7. UUIDv8 also future-proofs this specification by allowing time-based UUID formats from timestamp sources that are not yet be defined. The variable size timestamp offers lots of flexibility to create an implementation specific RFC compliant time-based UUID while retaining the properties that make UUID great. See Section 4.5

4. Format

The UUID length of 16 octets (128 bits) remains unchanged. The textual representation of a UUID consisting of 36 hexadecimal and dash characters in the format 8-4-4-4-12 remains unchanged for human readability. In addition the position of both the Version and Variant bits remain unchanged in the layout.

4.1. Versions

Table 1 defines the 4-bit version found in Bits 48 through 51 within a given UUID.

Msb0	Msb1	Msb2	Msb3	Version	Description
0	1	1	0	6	Reordered Gregorian time-based UUID
0	1	1	1	7	Variable length Unix Epoch time-based UUID
1	0	0	0	8	Custom time-based UUID

Table 1: UUID versions defined by this specification

4.2. Variant

The variant bits utilized by UUIDs in this specification remains the same as [RFC4122], Section 4.1.1.

The Table 2 lists the contents of the variant field, bits 64 and 65, where the letter "x" indicates a "don't-care" value. Common hex values of 8 (1000), 9 (1001), A (1010), and B (1011) frequent the text representation.

Msb0	Msb1	Msb2	Description
1	0	x	The variant specified in this document.

Table 2: UUID Variant defined by this specification

4.3. UUIDv6 Layout and Bit Order

UUIDv6 aims to be the easiest to implement by reusing most of the layout of bits found in UUIDv1 but with changes to bit ordering for the timestamp. Where UUIDv1 splits the timestamp bits into three distinct parts and orders them as time_low, time_mid, time_high_and_version. UUIDv6 instead keeps the source bits from the timestamp intact and changes the order to time_high, time_mid, and time_low. Incidentally this will match the original 60-bit Gregorian timestamp source. The clock sequence bits remain unchanged from their usage and position in [RFC4122]. The 48-bit node MUST be set to a pseudo-random value.

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

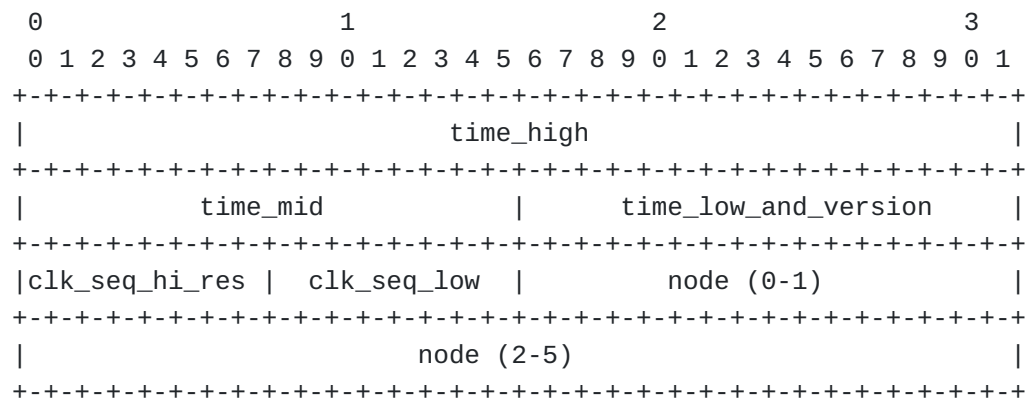

Figure 1: UUIDv6 Field and Bit Layout

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 pseudo-random number used as a spatially unique identifier Occupies bits 80 through 127 (octets 10-15)

4.3.1. UUIDv6 Timestamp Usage

UUIDv6 reuses the 60-bit Gregorian timestamp with 100-nanosecond precision defined in [RFC4122], Section 4.1.4.

4.3.2. UUIDv6 Clock Sequence Usage

UUIDv6 makes no change to the Clock Sequence usage defined by [RFC4122], Section 4.1.5.

4.3.3. UUIDv6 Node Usage

UUIDv6 node bits SHOULD be set to a 48-bit random or pseudo-random number. UUIDv6 nodes SHOULD NOT utilize an IEEE 802 MAC address or the [RFC4122], Section 4.5 method of generating a random multicast IEEE 802 MAC address.

4.3.4. UUIDv6 Basic Creation Algorithm

The following implementation algorithm is based on [RFC4122] but with changes specific to UUIDv6:

1. From a system-wide shared stable store (e.g., a file) or global variable, read the UUID generator state: the values of the timestamp and clock sequence used to generate the last UUID.

2. Obtain the current time as a 60-bit count of 100-nanosecond intervals since 00:00:00.00, 15 October 1582.

3. Set the time_low field to the 12 least significant bits of the starting 60-bit timestamp.

4. Truncate the timestamp to the 48 most significant bits in order to create time_high_and_time_mid.

5. Set the time_high field to the 32 most significant bits of the truncated timestamp.

6. Set the time_mid field to the 16 least significant bits of the truncated timestamp.

7. Create the 16-bit time_low_and_version by concatenating the 4-bit UUIDv6 version with the 12-bit time_low.

8. If the state was unavailable (e.g., non-existent or corrupted) or the timestamp is greater than the current timestamp generate a random 14-bit clock sequence value.

9. If the state was available, but the saved timestamp is less than or equal to the current timestamp, increment the clock sequence value.

10. Complete the 16-bit clock sequence high, low and reserved creation by concatenating the clock sequence onto UUID variant bits which take the most significant position in the 16-bit value.

11. Generate a 48-bit psuedo-random node.

12. Format by concatenating the 128 bits from each parts: time_high|time_mid|time_low_and_version|variant_clk_seq|node

13. Save the state (current timestamp and clock sequence) back to the stable store

The steps for splitting time_high_and_time_mid into time_high and time_mid are optional since the 48-bits of time_high and time_mid will remain in the same order as time_high_and_time_mid during the final concatenation. This extra step of splitting into the most significant 32 bits and least significant 16 bits proves useful when reusing an existing UUIDv1 implementation. In which the following logic can be applied to reshuffle the bits with minimal modifications.

UUIDv1 Field	Bits	UUIDv6 Field
time_low	32	time_high
time_mid	16	time_mid
time_high	12	time_low

Table 3: UUIDv1 to UUIDv6 Field Mappings

4.4. UUIDv7 Layout and Bit Order

The UUIDv7 format is designed to encode a Unix timestamp with arbitrary sub-second precision. The key property provided by UUIDv7 is that timestamp values generated by one system and parsed by another are guaranteed to have sub-section precision of either the generator or the parser, whichever is less. Additionally, the system parsing the UUIDv7 value does not need to know which precision was used during encoding in order to function correctly.

The format for the 16-octet, 128-bit UUIDv6 is shown in Figure 2

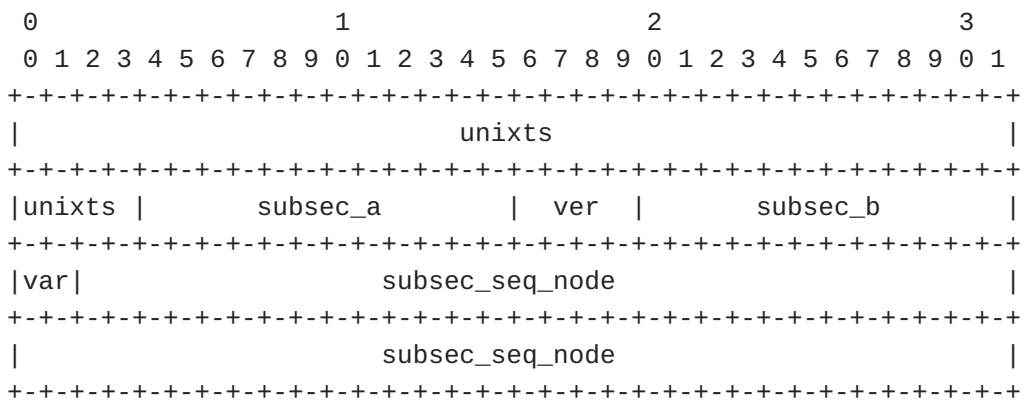

Figure 2: UUIDv7 Field and Bit Layout

unixts:
36-bit big-endian unsigned Unix Timestamp value

subsec_a:
12-bits allocated to sub-section precision values.

ver:
The 4 bit UUIDv8 version (0111)

subsec_b:
12-bits allocated to sub-section precision values.

var:
2-bit UUID variant (10)

subsec_seq_node:
The remaining 62 bits which MAY be allocated to any combination of additional sub-section precision, sequence counter, or pseudo-random data.

4.4.1. UUIDv7 Timestamp Usage

UUIDv7 utilizes a 36-bit big-endian unsigned Unix Timestamp value (number of seconds since the epoch of 1 Jan 1970, leap seconds excluded so each hour is exactly 3600 seconds long).

Additional sub-second precision (millisecond, nanosecond, microsecond, etc) MAY be provided for encoding and decoding in the remaining bits in the layout.

4.4.2. UUIDv7 Clock Sequence Usage

UUIDv7 SHOULD utilize a motonic sequence counter to provide additional sequencing guarantees when multiple UUIDv7 values are created in the same UNIXTS and SUBSEC timestamp. The amount of bits allocates to the sequence counter depend on the precision of the timestamp. For example, a more accurate timestamp source using nanosecond precision will require less clock sequence bits than a timestamp source utilizing seconds for precision. For best sequencing results the sequence counter SHOULD be placed immediately after available sub-second bits.

The clock sequence MUST start at zero and increment monotonically for each new UUID created on by the application on the same timestamp. When the timestamp increments the clock sequence MUST be reset to zero. The clock sequence MUST NOT rollover or reset to zero unless the timestamp has incremented. Care MUST be given to ensure that an adequate sized clock sequence is selected for a given application based on expected timestamp precision and expected UUID generation rates.

4.4.3. UUIDv7 Node Usage

UUIDv7 implementations, even with very detailed sub-second precision and the optional sequence counter, MAY have leftover bits that will be identified as the Node for this section. The UUIDv7 Node MAY contain any set of data an implementation desires however the node MUST NOT be set to all 0s which does not ensure global uniqueness. In most scenarios the node SHOULD be filled with pseudo-random data.

4.4.4. UUIDv7 Encoding and Decoding

The UUIDv7 bit layout for encoding and decoding are described separately in this document.

4.4.4.1. UUIDv7 Encoding

Since the UUIDv7 Unix timestamp is fixed at 36 bits in length the exact layout for encoding UUIDv7 depends on the precision (number of bits) used for the sub-second portion and the sizes of the optionally desired sequence counter and node bits.

Three examples of UUIDv7 encoding are given below as a general guidelines but implementations are not limited to just these three examples.

All of these fields are only used during encoding, and during decoding the system is unaware of the bit layout used for them and considers this information opaque. As such, implementations generating these values can assign whatever lengths to each field it deems applicable, as long as it does not break decoding compatibility (i.e. Unix timestamp (unixts), version (ver) and variant (var) have to stay where they are, and clock sequence counter (seq), random (random) or other implementation specific values must follow the sub-second encoding).

In Figure 3 the UUIDv7 has been created with millisecond precision with the available sub-second precision bits.

Examining Figure 3 one can observe:

- The first 36 bits have been dedicated to the Unix Timestamp (unixts)
- All 12 bits of scenario subsec_a is fully dedicated to millisecond information (msec).
- The 4 Version bits remain unchanged (ver).
- All 12 bits of subsec_b have been dedicated to a motonic clock sequence counter (seq).

*The 2 Variant bits remain unchanged (var).

*Finally the remaining 62 bits in the subsec_seq_node section are layout is filled out with random data to pad the length and provide guaranteed uniqueness (rand).

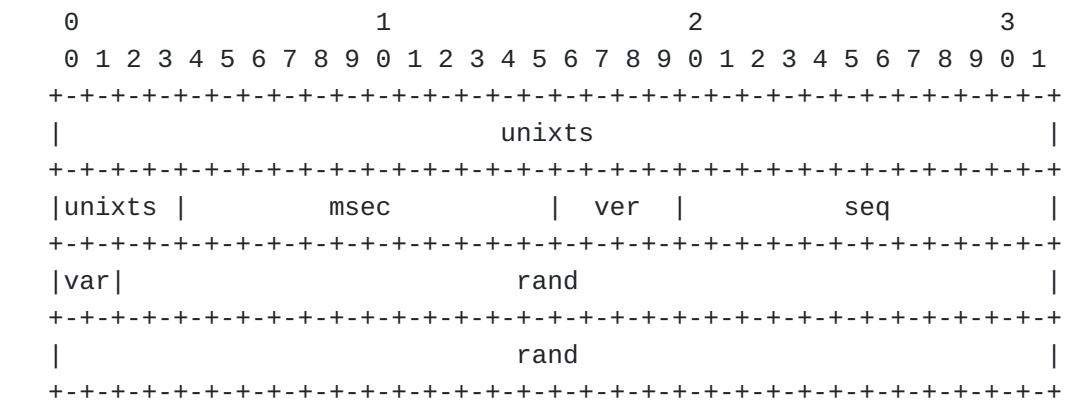

Figure 3: UUIDv7 Field and Bit Layout - Encoding Example (Millisecond Precision)

In Figure 4 the UUIDv7 has been created with Microsecond precision with the available sub-second precision bits.

Examining Figure 4 one can observe:

*The first 36 bits have been dedicated to the Unix Timestamp (unixts)

*All 12 bits of scenario subsec_a is fully dedicated to providing sub-second encoding for the Microsecond precision (usec).

*The 4 Version bits remain unchanged (ver).

*All 12 bits of subsec_b have been dedicated to providing sub-second encoding for the Microsecond precision (usec).

*The 2 Variant bits remain unchanged (var).

*A 14 bit motonic clock sequence counter (seq) has been embedded in the most significant position of subsec_seq_node

*Finally the remaining 48 bits in the subsec_seq_node section are layout is filled out with random data to pad the length and provide guaranteed uniqueness (rand).

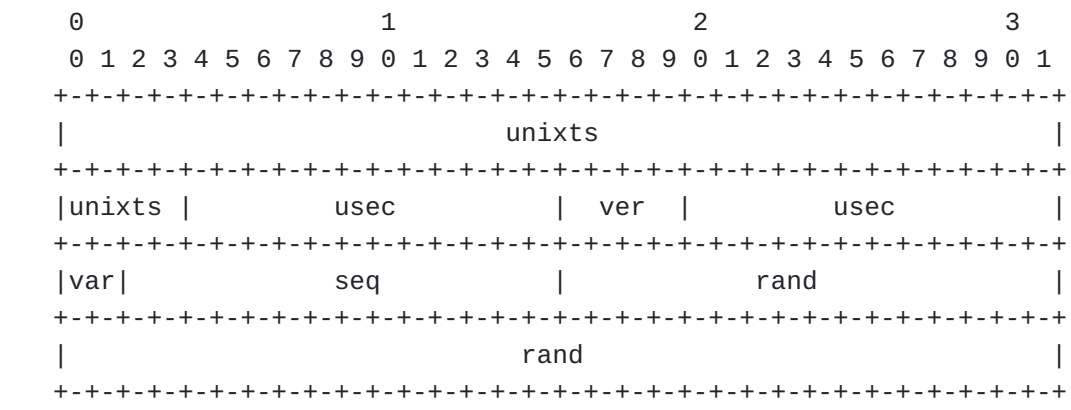

Figure 4: UUIDv7 Field and Bit Layout - Encoding Example (Microsecond Precision)

In Figure 5 the UUIDv7 has been created with Nanosecond precision with the available sub-second precision bits.

Examining Figure 5 one can observe:

*The first 36 bits have been dedicated to the Unix Timestamp (unixts)

*All 12 bits of scenario subsec_a is fully dedicated to providing sub-second encoding for the Nanosecond precision (nsec).

*The 4 Version bits remain unchanged (ver).

*All 12 bits of subsec_b have been dedicated to providing sub-second encoding for the Nanosecond precision (nsec).

*The 2 Variant bits remain unchanged (var).

*The first 14 bit of the subsec_seq_node dedicated to providing sub-second encoding for the Nanosecond precision (nsec).

*The next 8 bits of subsec_seq_node dedicated a motonic clock sequence counter (seq).

*Finally the remaining 40 bits in the subsec_seq_node section are layout is filled out with random data to pad the length and provide guaranteed uniqueness (rand).

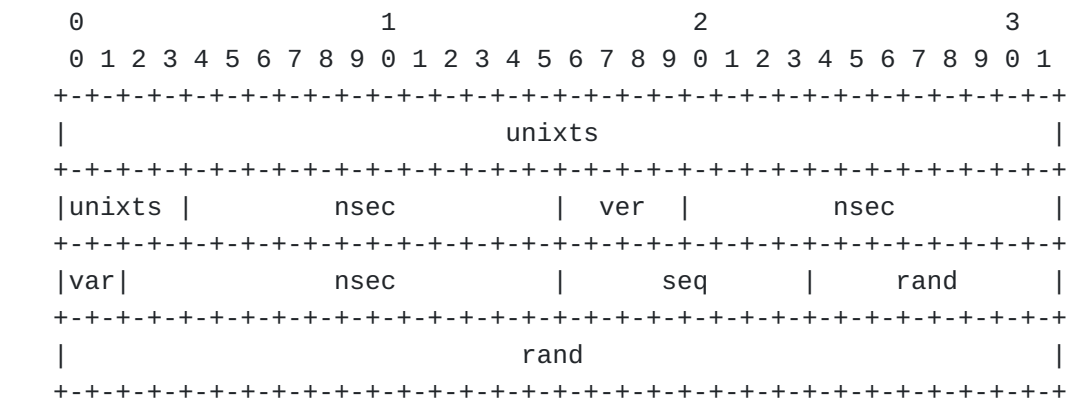

Figure 5: UUIDv7 Field and Bit Layout - Encoding Example (Nanosecond Precision)

4.4.4.2. UUIDv7 Decoding

When decoding or parsing a UUIDv7 value there are only two values to be considered:

1. The unix timestamp defined as unixts

2. The sub-second precision values defined as subsec_a, subsec_b, and subsec_seq_node

As detailed in Figure 2 the unix timestamp (unixts) is always the first 36 bits of the UUIDv7 layout.

Similarly as per Figure 2, the sub-second precision values lie within subsec_a, subsec_b, and subsec_seq_node which are all interpreted as sub-second information after skipping over the version (ver) and (var) bits. These concatenated sub-second information bits are interpreted in a way where most to least significant bits represent a further division by two. This is the same normal place notation used to express fractional numbers, except in binary. For example, in decimal ".1" means one tenth, and ".01" means one hundredth. In this subsec field, a 1 means one half, 01 means one quarter, 001 is one eighth, etc. This scheme can work for any number of bits up to the maximum available, and keeps the most significant data leftmost in the bit sequence.

To perform the sub-second math, simply take the first (most significant/leftmost) N bits of subsec and divide it by 2^N. Take for example:

1. To parse the first 16 bits, extract that value as an integer and divide it by 65536 (2 to the 16th).

2. If these 16 bits are 0101 0101 0101 0101, then treating that as an integer gives 0x5555 or 21845 in decimal, and dividing by 65536 gives 0.3333282

This sub-second encoding scheme provides maximum interoperability across systems where different levels of time precision are required/feasible/available. The timestamp value derived from a UUIDv7 value SHOULD be "as close to the correct value as possible" when parsed, even across disparate systems.

Take for example the starting point for our next two UUIDv7 parsing scenarios:

1. System A produces a UUIDv7 with a microsecond-precise timestamp value.

2. System B is unaware of the precision encoded in the UUIDv7 timestamp by System A.

Scenario 1:

1. System B parses the embedded timestamp with millisecond precision. (Less precision than the encoder)

2. System B SHOULD return the correct millisecond value encoded by system A (truncated to milliseconds).

Scenario 2:

1. System B parses the timestamp with nanosecond precision. (More precision than the encoder)

2. System B's value returned SHOULD have the same microsecond level of precision provided by the encoder with the additional precision down to nanosecond level being essentially random as per the encoded random value at the end of the UUIDv7.

4.5. UUIDv8 Layout and Bit Order

UUIDv8 offers variable-size timestamp, clock sequence, and node values which allow for a highly customizable UUID that fits a given application needs.

UUIDv8 SHOULD only be utilized if an implementation cannot utilize UUIDv1, UUIDv6, or UUIDv8. Some situations in which UUIDv8 usage could occur:

*An implementation would like to utilize a timestamp source not defined by the current time-based UUIDs.

*An implementation would like to utilize a timestamp bit layout not defined by the current time-based UUIDs.

*An implementation would like a specific level of precision within the timestamp not offered by current time-based UUIDs.

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

*An implementation has other application/language restrictions which inhibit the usage of one of the current time-based UUIDs.

Roughly speaking a properly formatted UUIDv8 SHOULD contain the following sections adding up to a total of 128-bits.

- Timestamp Bits (Variable Length)
- Clock Sequence Bits (Variable Length)
- Node Bits (Variable Length)
- UUIDv8 Version Bits (4 bits)
- UUID Variant Bits (2 Bits)

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. UUIDv8's 128 bits (including the version and variant) SHOULD contain at the minimum a timestamp of some format in the most significant bit position followed directly by a clock sequence counter and finally a node containing either random data or implementation specific data.

A sample format in Figure 6 is used to further illustrate the point for the 16-octet, 128-bit UUIDv8.

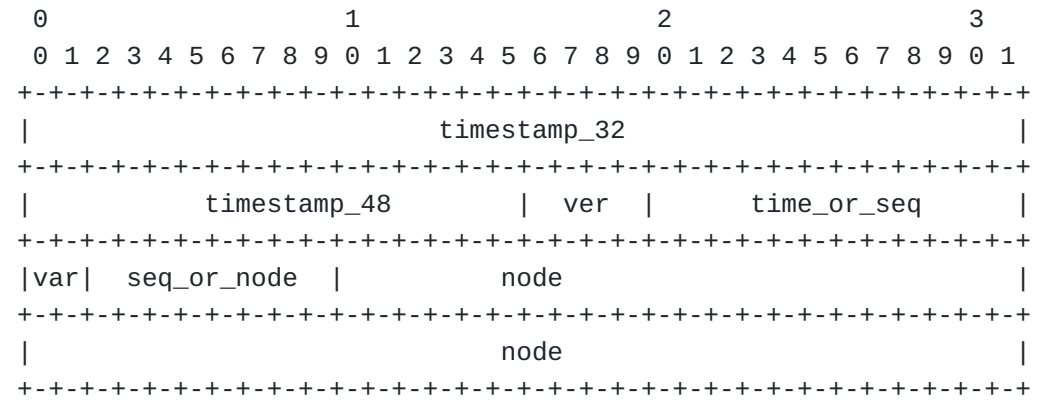

Figure 6: UUIDv8 Field and Bit Layout

timestamp_32:
The most significant 32 bits of the desired timestamp source. Occupies bits 0 through 31 (octets 0-3).

timestamp_48:
The next 16-bits of the timestamp source when a timestamp source with at least 48 bits is used. When a 32-bit timestamp source is utilized, these bits are set to 0. Occupies bits 32 through 47

ver:
The 4 bit UUIDv8 version (1000). Occupies bits 48 through 51.

time_or_seq:
If a 60-bit, or larger, timestamp is used these 12-bits are used to fill out the remaining timestamp. If a 32 or 48-bit timestamp is leveraged a 12-bit clock sequence MAY be used. Together ver and time_or_seq occupy bits 48 through 63 (octets 6-7)

var:
2-bit UUID variant (10)

seq_or_node:
If a 60-bit, or larger, timestamp source is leverages these 8 bits SHOULD be allocated for an 8-bit clock sequence counter. If a 32 or 48 bit timestamp source is used these 8-bits SHOULD be set to random.

node:
In most implementations these bits will likely be set to pseudo-random data. However, implementations utilize the node as they see fit. Together var, seq_or_node, and node occupy Bits 64 through 127 (octets 8-15)

4.5.1. UUIDv8 Timestamp Usage

UUIDv8's usage of timestamp relaxes both the timestamp source and timestamp length. Implementations are free to utilize any monotonically stable timestamp source for UUIDv8.

Some examples include:

- Custom Epoch
- NTP Timestamp
- ISO 8601 timestamp

The relaxed nature UUIDv8 timestamps also works to future proof this specification and allow implementations a method to create compliant time-based UUIDs using timestamp source that might not yet be defined.

Timestamps come in many sizes and UUIDv8 defines three fields that can easily used for the majority of timestamp lengths:

- 32-bit timestamp: using timestamp_32 and setting timestamp_48 to 0s
- 48-bit timestamp: using timestamp_32 and timestamp_48 entirely
- 60-bit timestamp: using timestamp_32, timestamp_48, and time_or_seq
- 64-bit timestamp: using timestamp_32, timestamp_48, and time_or_seq and truncating the timestamp the 60 most significant bits.

Although it is possible to create a timestamp larger than 64-bits in size The usage and bit layout of that timestamp format is up to the implementation. When a timestamp exceeds the 64th bit (octet 7), extra care must be taken to ensure the Variant bits are properly inserted at their respective location in the UUID. Likewise, the Version MUST always be implemented at the appropriate location.

Any timestamps that does not entirely fill the timestamp_32, timestamp_48 or time_or_seq MUST set all leftover bits in the least significant position of the respective field to 0. For example a 36-bit timestamp source would fully utilize timestamp_32 and 4-bits of timestamp_48. The remaining 12-bits in timestamp_48 MUST be set to 0.

By using implementation-specific timestamp sources it is not guaranteed that devices outside of the application context are able to extract and parse the timestamp from UUIDv8 without some pre-existing knowledge of the source timestamp used by the UUIDv8 implementation.

4.5.2. UUIDv8 Clock Sequence Usage

A clock sequence MUST be used with UUIDv8 as added sequencing guarantees when multiple UUIDv8 will be created on the same clock tick. The amount of bits allocated to the clock sequence depends on the precision of the timestamp source. For example, a more accurate timestamp source using nanosecond precision will require less clock sequence bits than a timestamp source utilizing seconds for precision.

The UUIDv8 layout in Figure 6 generically defines two possible clock sequence values that can leveraged:

* 12-bit clock sequence using time_or_seq for use when the timestamp is less than 48-bits which allows for 4095 UUIDs per clock tick.
* 8-bit clock sequence using seq_or_node when the timestamp uses more than 48-bits which allows for 255 UUIDs per clock tick.

An implementation MAY use both time_or_seq and seq_or_node for clock sequencing however it is highly unlikely that 20-bits of clock sequence are needed for a given clock tick. Furthermore, more bits from the node MAY be used for clock sequencing in the event that 8-bits is not sufficient.

The clock sequence MUST start at zero and increment monotonically for each new UUID created on by the application on the same timestamp. When the timestamp increments the clock sequence MUST be reset to zero. The clock sequence MUST NOT rollover or reset to zero unless the timestamp has incremented. Care MUST be given to ensure that an adequate sized clock sequence is selected for a given application based on expected timestamp precision and expected UUID generation rates.

4.5.3. UUIDv8 Node Usage

The UUIDv8 Node MAY contain any set of data an implementation desires however the node MUST NOT be set to all 0s which does not ensure global uniqueness. In most scenarios the node will be filled with pseudo-random data.

The UUIDv8 layout in Figure 6 defines 2 sizes of Node depending on the timestamp size:

* 62-bit node encompassing seq_or_node and node Used when a timestamp of 48-bits or less is leveraged.
* 54-bit node when all 60-bits of the timestamp are in use and the seq_or_node is used as clock sequencing.

An implementation MAY choose to allocate bits from the node to the timestamp, clock sequence or application-specific embedded field. It is recommended that implementation utilize a node of at least 48-bits to ensure global uniqueness can be guaranteed.

4.5.4. UUIDv6 Basic Creation Algorithm

The entire usage of UUIDv8 is meant to be variable and allow as much customization as possible to meet specific application/language requirements. As such any UUIDv8 implementations will likely vary among applications.

The following algorithm is a generic implementation using Figure 6 and the recommendations outlined in this specification.

32-bit timestamp, 12-bit sequence counter, 62-bit node:

1. From a system-wide shared stable store (e.g., a file) or global variable, read the UUID generator state: the values of the timestamp and clock sequence used to generate the last UUID.

2. Obtain the current time from the selected clock source as 32 bits.

3. Set the 32-bit field timestamp_32 to the 32 bits from the timestamp

4. Set 16-bit timestamp_48 to all 0s

5. Set the version to 8 (1000)

6. If the state was unavailable (e.g., non-existent or corrupted) or the timestamp is greater than the current timestamp; set the 12-bit clock sequence value (time_or_node) to 0

7. If the state was available, but the saved timestamp is less than or equal to the current timestamp, increment the clock sequence value (time_or_node).

8. Set the variant to binary 10

9. Generate 62 random bits and fill in 8-bits for seq_or_node and 54-bits for the node.

10. Format by concatenating the 128-bits as: timestamp_32| timestamp_48|version|time_or_node|variant|seq_or_node|node

11. Save the state (current timestamp and clock sequence) back to the stable store

48-bit timestamp, 12-bit sequence counter, 62-bit node:

1. From a system-wide shared stable store (e.g., a file) or global variable, read the UUID generator state: the values of the timestamp and clock sequence used to generate the last UUID.

2. Obtain the current time from the selected clock source as 32 bits.

3. Set the 32-bit field timestamp_32 to the 32 most significant bits from the timestamp

4. Set 16-bit timestamp_48 to the 16 least significant bits from the timestamp

5. The rest of the steps are the same as the previous example.

60-bit timestamp, 8-bit sequence counter, 54-bit node:

1. From a system-wide shared stable store (e.g., a file) or global variable, read the UUID generator state: the values of the timestamp and clock sequence used to generate the last UUID.

2. Obtain the current time from the selected clock source as 32 bits.

3. Set the 32-bit field timestamp_32 to the 32 bits from the timestamp

4. Set 16-bit timestamp_48 to the 16 middle bits from the timestamp

5. Set the version to 8 (1000)

6. Set 12-bit time_or_node to the 12 least significant bits from the timestamp

7. Set the variant to 10

8. If the state was unavailable (e.g., non-existent or corrupted) or the timestamp is greater than the current timestamp; set the 12-bit clock sequence value (seq_or_node) to 0

9. If the state was available, but the saved timestamp is less than or equal to the current timestamp, increment the clock sequence value (seq_or_node).

10. Generate 54 random bits and fill in the node

11. Format by concatenating the 128-bits as: timestamp_32| timestamp_48|version|time_or_node|variant|seq_or_node|node

12. Save the state (current timestamp and clock sequence) back to the stable store

64-bit timestamp, 8-bit sequence counter, 54-bit node:

1. The same steps as the 60-bit timestamp can be utilized if the 64-bit timestamp is truncated to 60-bits.

2. Implementations MAY chose to truncate the most or least significant bits but it is recommended to utilize the most

significant 60-bits and lose 4 bits of precision in the nanoseconds or microseconds position.

General algorithm for generation of UUIDv8 not defined here:

1. From a system-wide shared stable store (e.g., a file) or global variable, read the UUID generator state: the values of the timestamp and clock sequence used to generate the last UUID.

2. Obtain the current time from the selected clock source as desired bit total

3. Set total amount of bits for timestamp as required in the most significant positions of the 128-bit UUID

4. Care MUST be taken to ensure that the UUID Version and UUID Variant are in the correct bit positions.

   UUID Version: Bits 48 through 51

   UUID Variant: Bits 64 and 65

5. If the state was unavailable (e.g., non-existent or corrupted) or the timestamp is greater than the current timestamp; set the desired clock sequence value to 0

6. If the state was available, but the saved timestamp is less than or equal to the current timestamp, increment the clock sequence value.

7. Set the remaining bits to the node as pseudo-random data

8. Format by concatenating the 128-bits together

9. Save the state (current timestamp and clock sequence) back to the stable store

5. Encoding and Storage

The existing UUID hex and dash format of 8-4-4-4-12 is retained for both backwards compatibility and human readability.

For many applications such as databases this format is unnecessarily verbose totaling 288 bits.

*8-bits for each of the 32 hex characters = 256 bits
*8-bits for each of the 4 hyphens = 32 bits

Where possible UUIDs SHOULD be stored within database applications as the underlying 128-bit binary value.

6. Global Uniqueness

UUIDs created by this specification offer the same guarantees for global uniqueness as those found in [RFC4122]. Furthermore, the time-based UUIDs defined in this specification are geared towards database applications but MAY be used for a wide variety of use-cases. Just as global uniqueness is guaranteed, UUIDs are guaranteed to be unique within an application context within the enterprise domain.

7. Distributed UUID Generation

Some implementations might desire to utilize multi-node, clustered, applications which involve 2 or more applications independently generating UUIDs that will be stored in a common location. UUIDs already feature sufficient entropy to ensure that the chances of collision are low. However, implementations MAY dedicate a portion of the node's most significant random bits to a pseudo-random machineID which helps identify UUIDs created by a given node. This works to add an extra layer of collision avoidance.

This machine ID MUST be placed in the UUID proceeding the timestamp and sequence counter bits. This position is selected to ensure that the sorting by timestamp and clock sequence is still possible. The machineID MUST NOT be an IEEE 802 MAC address. The creation and negotiation of the machineID among distributed nodes is out of scope for this specification.

8. IANA Considerations

This document has no IANA actions.

9. Security Considerations

MAC addresses pose inherent security risks and MUST not be used for node generation. As such they have been strictly forbidden from time-based UUIDs within this specification. Instead pseudo-random bits SHOULD selected from a source with sufficient entropy to ensure guaranteed uniqueness among UUID generation.

Timestamps embedded in the UUID do pose a very small attack surface. The timestamp in conjunction with the clock sequence 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.

The machineID portion of node, described in Section 7, does provide small unique identifier which could be used to determine which

application is generating data but this machineID alone is not enough to identify a node on the network without other corresponding data points. Furthermore the machineID, like the timestamp+sequence, does not provide any context about the data the corresponds to the UUID or the current state of the application as a whole.

10. 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. As well as all of those in and outside the IETF community to who contributed to the discussions which resulted in this document.

11. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/info/rfc2119>.

[RFC4122] Leach, P., Mealling, M., and R. Salz, "A Universally Unique IDentifier (UUID) URN Namespace", RFC 4122, DOI 10.17487/RFC4122, July 2005, <https://www.rfc-editor.org/info/rfc4122>.

12. Informative References

[LexicalUUID] Twitter, "A Scala client for Cassandra", commit f6da4e0, November 2012, <https://github.com/twitter-archive/cassie>.

[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>.

[Flake] Boundary, "Flake: A decentralized, k-ordered id generation service in Erlang", Commit 15c933a, February 2017, <https://github.com/boundary/flake>.

[ShardingID] Instagram Engineering, "Sharding & IDs at Instagram", December 2012, <https://instagram-engineering.com/sharding-ids-at-instagram-1cf5a71e5a5c>.

[KSUID] Segment, "K-Sortable Globally Unique IDs", Commit bf376a7, July 2020, <https://github.com/segmentio/ksuid>.

[Elasticflake]
Pearcy, P., "Sequential UUID / Flake ID generator pulled out of elasticsearch common", Commit dd71c21, January 2015, <https://github.com/ppearcy/elasticflake>.

[FlakeID] Pawlak, T., "Flake ID Generator", Commit fcd6a2f, April 2020, <https://github.com/T-PWK/flake-idgen>.

[Sonyflake] Sony, "A distributed unique ID generator inspired by Twitter's Snowflake", Commit 848d664, August 2020, <https://github.com/sony/sonyflake>.

[orderedUuid] Cabrera, IT., "Laravel: The mysterious "Ordered UUID"", January 2020, <https://itnext.io/laravel-the-mysterious-ordered-uuid-29e7500b4f8>.

[COMBGUID] Tallent, R., "Creating sequential GUIDs in C# for MSSQL or PostgreSql", Commit 2759820, December 2020, <https://github.com/richardtallent/RT.Comb>.

[ULID] Feerasta, A., "Universally Unique Lexicographically Sortable Identifier", Commit d0c7170, May 2019, <https://github.com/ulid/spec>.

[SID] Chilton, A., "sid : generate sortable identifiers", Commit 660e947, June 2019, <https://github.com/chilts/sid>.

[pushID] Google, "The 2^120 Ways to Ensure Unique Identifiers", February 2015, <https://firebase.googleblog.com/2015/02/the-2120-ways-to-ensure-unique_68.html>.

[XID] Poitrey, O., "Globally Unique ID Generator", Commit efa678f, October 2020, <https://github.com/rs/xid>.

[ObjectID] MongoDB, "ObjectId - MongoDB Manual", <https://docs.mongodb.com/manual/reference/method/ObjectId/>.

[CUID] Elliott, E., "Collision-resistant ids optimized for horizontal scaling and performance.", Commit 215b27b, October 2020, <https://github.com/ericelliott/cuid>.

Authors' Addresses

Brad G. Peabody

Email: brad@peabody.io

Kyzer R. Davis

Email: kydavis@cisco.com