Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: 26 August 2021

C. Bormann
Universität Bremen TZI
H. Birkholz
Fraunhofer SIT

22 February 2021

On Media-Types, Content-Types, and related terminology
draft-bormann-core-media-content-type-format-04

Abstract

There is a lot of confusion about media-types, content-types, and related terminology.

This memo is an attempt at clearing it up, so we can use consistent terminology in CoRE and related specifications. It also defines some ABNF that can be used in these specifications.

Table of Contents

1. Introduction

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.

[RFC1590] introduced media types and their registration. That document took MIME types from [RFC1521] and gave them a new name. At that time, the term "media type" was often used just for the major type ("text", "audio"), and what we call a media-type now was the combination of a type and a subtype. This lives on in [RFC6838], which does not even have an ABNF [RFC5234] production for media type. [RFC6838]'s predecessor, [RFC4288], supplied the ABNF shown in (Figure 1).

```
type-name = reg-name
subtype-name = reg-name

reg-name = 1*127reg-name-chars
reg-name-chars = ALPHA / DIGIT / "!" /
                 "#" / "$" / "&" / "." /
                 "+" / "-" / "^" / "_"
```

Figure 1: ABNF for type and subtype, cited from RFC 4288

[RFC6838], obsoleting [RFC4288], restricts the first character of a reg-name to alphanumeric. It contains the otherwise semantically equivalent ABNF shown in Figure 2, however adding prose comments that further limit the use of "." and "+".

```
type-name = restricted-name
subtype-name = restricted-name

restricted-name = restricted-name-first *126restricted-name-chars
restricted-name-first  = ALPHA / DIGIT
restricted-name-chars  = ALPHA / DIGIT / "!" / "#" /
                         "$" / "&" / "-" / "^" / "_"
restricted-name-chars =/ "." ; Characters before first dot always
                             ; specify a facet name
restricted-name-chars =/ "+" ; Characters after last plus always
                             ; specify a structured syntax suffix
```

Figure 2: ABNF for type and subtype, as defined from RFC 6838

2. Media-Type

Today, the term "media type" is now generally used for a registered combination of a type-name and a subtype-name, as well as for the specification that defines the semantics of this combination. We further disambiguate by calling the former a _media type name_. An ABNF definition of "Media-Type-Name":

```
Media-Type-Name = type-name "/" subtype-name
```

Figure 3: Definition of Media-Type-Name

For the purposes of this memo, we define:

Media-Type-Name: A combination of a type-name and a subtype-name registered in [IANA.media-types], conventionally identified by the two names separated by a slash.

(This leaves the term "Media Type" for the actual specification that is registered under the Media-Type-Name.)

3. Content-Type

Media types can have parameters [RFC6838], some of which are defined by the media type specification to be mandatory. In HTTP and many other protocols, media-type-names and parameters are then used together in a "Content-Type" header field. HTTP [RFC7231] uses the ABNF in Figure 4:

```
Content-Type = media-type
media-type = type "/" subtype *( OWS ";" OWS parameter )
type       = token
subtype    = token
token          = 1*tchar
tchar          = "!" / "#" / "$" / "%" / "&" / "'" / "*"
               / "+" / "-" / "." / "^" / "_" / "`" / "|" / "~"
               / DIGIT / ALPHA
OWS        = *( SP / HTAB )
```

Figure 4: Content-Type ABNF from RFC 7231

In the ABNF as established by [RFC2616], parts of which became [RFC7231], the rule name media-type is used for a Media-Type-Name with parameters attached. We don't follow this inclusive use of media-type; note that [RFC2616] was quite confused about this term by claiming (Section 3.7 of [RFC2616]):

> Media-type values are registered with the Internet Assigned Number Authority (IANA [19]).

This clearly reverts to the understanding of Media-Type-Name we use.

In order to resolve some of this confusion, we define as a separate term:

Content-Type: A Media-Type-Name, optionally associated with parameters (separated from the media type name and from each other by a semicolon).

Removing the legacy HTAB characters now shunned in polite conversation, as well as some other cobwebs, we define the conventional textual representation of a Content-Type with the ABNF in Figure 5:

```
Content-Type   = Media-Type-Name *( *SP ";" *SP parameter )
parameter      = token "=" ( token / quoted-string )

token          = 1*tchar
tchar          = "!" / "#" / "$" / "%" / "&" / "'" / "*"
               / "+" / "-" / "." / "^" / "_" / "`" / "|" / "~"
               / DIGIT / ALPHA
quoted-string  = %x22 *qdtext %x22
qdtext         = SP / %x21 / %x23-5B / %x5D-7E
```

Figure 5: Definition of Content-Type

Note that there is a slight inconsistency between the "token" used here and the "reg-name"/"restricted-name" used above; since media type parameters probably will be defined within the guard rails set by [RFC7231], we need to use HTTP's more comprehensive definition here.

4. Content-Coding

Section 3.5 of [RFC2616] also introduced the term Content-Coding, a registered name for an encoding transformation that has been or can be applied to a representation:

```
content-coding   = token
```

Figure 6: Definition of content-coding as in RFC 2616

Confusingly, in HTTP the Content-Coding is then given in a header field called "Content-Encoding"; we *never* use this term (except when we are in error). Instead we define:

Content-Coding: a registered name for an encoding transformation that has been or can be applied to a representation.

Content-Codings are registered in the HTTP Content Coding Registry, a subregistry of [IANA.http-parameters]. We often use the "identity" Content-Coding, which is the identity transformation, and often fail to identify that Content-Coding by name, instead calling it "no Content-Coding".

5. Content-Format

CoAP, in Section 1 of [RFC7252], defines a Content-Format as the combination of a Content-Type and a Content-Coding, identified by a numeric identifier defined in the "CoAP Content-Formats" registry (a subregistry of [IANA.core-parameters]), but in more confusing words (it did not have the benefit of the present specifications).

Content-Format: the combination of a Content-Type and a Content-Coding, identified by a numeric identifier defined by the "CoAP Content-Formats" subregistry of [IANA.core-parameters].

Note that there has not been a conventional string representation of just the combination of a Content-Type and a Content-Coding; Content-Formats so far always are identified by their registered Content-Format numbers. However, there are applications where that is useful [I-D.keranen-core-senml-data-ct], so we define:

```
Content-Format = "0" / (POS-DIGIT *DIGIT)
Content-Format-String   = Content-Type ["@" content-coding]
```

Figure 7: Definition of Content-Format/-String

This allows the use of Content-Format-Strings such as "application/json@deflate" in place of the less self-describing content-format "11050", or other combinations that do not have a content-format number defined yet.

Content-Format-Strings MUST NOT explicitly use the content-coding value of "identity" (i.e., if an identity content-coding is desired, the entire optional part including the "@" sign is left out).

Note that a quoted string inside a content-type parameter might contain an "@" sign, so the parsing of Content-Format-Strings cannot be done in a too simplistic way.

6. Remaining ABNF

This specification uses the ABNF given in Figure 8, as originally defined in [RFC5234] and [RFC8866]:

```
DIGIT     =  %x30-39           ; 0  9
POS-DIGIT =  %x31-39           ; 1  9
ALPHA     =  %x41-5A / %x61-7A ; A  Z / a  z
SP        =  %x20
```

Figure 8: Commonly Used ABNF Definitions

7. Abbreviations

Media type names are sometimes abbreviated as "mt", and Content-Types as "ct". We propose not to use those abbreviations: Where the long form of the values can be used, the long form "Content-Type" can also be used to name them.

For historical reasons, both [RFC6690] and [RFC7252] use the abbreviation "ct" for Content-Format (think first and last character).

For Content-Coding, the abbreviation "cc" can be used.

8. Discussion

The ABNF given here is provisional and may need some more cleanup, such as unifying the various forms of reg-name, token, etc.

(ABNF just shown for illustration is centered, in a blockquote, and tagged with "<artwork type="abnf;old"...>" in the XML, while the normative ABNF of this memo is left-aligned and tagged with "<sourcecode type="abnf"...>".)

The XPath expression "//sourcecode[@type='abnf']/text()" can be used on the XML form of this specification to extract the ABNF defined here.

We need to discuss case-insensitivity at some point, which is usually rather insensitive.

9. Suggested usage

9.1. COSE

Section 3.1 of [RFC8152] defines a common COSE header parameter (number 3) called "content type" in the description, to indicate the type of the data in the payload or ciphertext fields.

This header parameter can either be an unsigned integer, indicating a CoRE Content-Format number, or a text string. The latter alternative is only defined in general terms. It points to Section 4.2 of [RFC6838] for 'text values following the syntax of "<type-name>/<subtype-name>"...', but also discusses the use of parameters and subparameters; no ABNF or similar detail specification is provided. The text does not discuss the use of Content-Coding in the text string form, probably because nothing like the present document existed at the time, creating a weird gap compared with numeric Content-Format values. (The text only has trivial changes in its updated version in Section 3.1 of [I-D.ietf-cose-rfc8152bis-struct-15].)

The present specification suggests using the production "Content-Format-String" as a more formal definition of the text string that can go into the "content type" (number 3) common header parameter in COSE.

9.2. SenML

As discussed above, Section 3 of [I-D.keranen-core-senml-data-ct] makes use of the present specification.

9.3. ...

(to be filled in along further use cases)

10. IANA Considerations

While this memo talks a lot about IANA registries, it does not require any action from IANA.

11. Security Considerations

Confusion about terminology may, in the worst case, cause security problems, as can loosely defined syntax elements of a specification. No other security considerations are known to be raised by the present specification.

12. References

12.1. Normative References

[IANA.core-parameters]
IANA, "Constrained RESTful Environments (CoRE) Parameters", <http://www.iana.org/assignments/core-parameters>.

[IANA.http-parameters]
IANA, "Hypertext Transfer Protocol (HTTP) Parameters", <http://www.iana.org/assignments/http-parameters>.

[IANA.media-types]
IANA, "Media Types", <http://www.iana.org/assignments/media-types>.

[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>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

12.2. Informative References

[I-D.ietf-cose-rfc8152bis-struct-15]
Schaad, J., "CBOR Object Signing and Encryption (COSE): Structures and Process", Work in Progress, Internet-Draft, draft-ietf-cose-rfc8152bis-struct-15, 1 February 2021, <https://www.ietf.org/archive/id/draft-ietf-cose-rfc8152bis-struct-15.txt>.

[I-D.keranen-core-senml-data-ct]
Keranen, A. and C. Bormann, "SenML Data Value Content-Format Indication", Work in Progress, Internet-Draft, draft-keranen-core-senml-data-ct-02, 8 July 2019, <https://www.ietf.org/archive/id/draft-keranen-core-senml-data-ct-02.txt>.

[RFC1521] Borenstein, N. and N. Freed, "MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies", RFC 1521, DOI 10.17487/RFC1521, September 1993, <https://www.rfc-editor.org/info/rfc1521>.

[RFC1590] Postel, J., "Media Type Registration Procedure", RFC 1590, DOI 10.17487/RFC1590, March 1994, <https://www.rfc-editor.org/info/rfc1590>.

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, DOI 10.17487/RFC2616, June 1999, <https://www.rfc-editor.org/info/rfc2616>.

[RFC4288] Freed, N. and J. Klensin, "Media Type Specifications and Registration Procedures", RFC 4288, DOI 10.17487/RFC4288, December 2005, <https://www.rfc-editor.org/info/rfc4288>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

[RFC6690] Shelby, Z., "Constrained RESTful Environments (CoRE) Link Format", RFC 6690, DOI 10.17487/RFC6690, August 2012, <https://www.rfc-editor.org/info/rfc6690>.

[RFC6838] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, DOI 10.17487/RFC6838, January 2013, <https://www.rfc-editor.org/info/rfc6838>.

[RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <https://www.rfc-editor.org/info/rfc7231>.

[RFC7252] Shelby, Z., Hartke, K., and C. Bormann, "The Constrained Application Protocol (CoAP)", RFC 7252, DOI 10.17487/RFC7252, June 2014, <https://www.rfc-editor.org/info/rfc7252>.

[RFC8152] Schaad, J., "CBOR Object Signing and Encryption (COSE)", RFC 8152, DOI 10.17487/RFC8152, July 2017, <https://www.rfc-editor.org/info/rfc8152>.

[RFC8866] Begen, A., Kyzivat, P., Perkins, C., and M. Handley, "SDP: Session Description Protocol", RFC 8866, DOI 10.17487/RFC8866, January 2021, <https://www.rfc-editor.org/info/rfc8866>.

Acknowledgements

Matthias Kovatsch forced the authors to make up their minds about this. Ari Keränen forced them to write it up, then, and created a convincing use case of Content-Format-Strings. John Mattsson alerted us to a mistake. Alexey Melnikov suggested reviving this draft after a year of dormancy.

Authors' Addresses

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

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

Henk Birkholz
Fraunhofer SIT
Rheinstrasse 75
64295 Darmstadt
Germany

Email: henk.birkholz@sit.fraunhofer.de

Multiparty Multimedia Session Control
Internet-Draft
Intended status: Standards Track
Expires: August 10, 2021

J. Gruessing

February 6, 2021

SDP Mapping into HTTP structured headers
draft-gruessing-sdp-http-02

Abstract

This document specifies a HTTP header based representation of the Session Description Protocol which can be used in describing media being negotiated or delivered via HTTP.

Note to Readers

RFC Editor: please remove this section before publication

Source code and issues for this draft can be found at https://github.com/fiestajetsam/I-D/tree/main/draft-gruessing-sdp-http [1].

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 August 10, 2021.

Table of Contents

1. Introduction

Services either negotiating or offering media over HTTP may want to express a greater amount of information beyond a MIME type of content and its preference.

The Session Description Protocol [RFC8866] describes multimedia sessions for the purpose of session announcement and initiation.

The Session-Description and Session-Media headers may be used for either a HTTP request or response and may be included as part of any HTTP method.

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. The Session-Description Header

The Session-Description header field conveys the entire session description information, using [I-D.ietf-httpbis-header-structure] to describe the structure. Its value MUST be a dictionary containing containing only the following keys, receivers MUST ignore all other values. Dictionary keys MUST be ordered in the order presented in this document, but MAY be omitted where they are explicitly declared as OPTIONAL.

2.1. Time Description

All values within time description fields that represent wall time MUST be values shown as integers which represent NTP timestamps with second resolution. To facilitate ease of parsing, fields that are used to represet a time duration or offset as described in Section 5.10 of [RFC8866] MUST NOT use the compact version e.g "1h" instead of "3600".

2.2. Session Description

- v The version, represented as an sh-integer that MUST be set 0.
- o The originator of the session, whose value is an sh-list. The order of values present in the list MUST map to the order specified in Section 5.2 of [RFC8866].
- s The name of the SDP, whose values is a string and MUST NOT be empty.
- i An OPTIONAL description of the session, whose value is a string.
- u An OPTIONAL URI reference containing additional information about the session, whose value is a string and SHOULD be represented as [RFC3986].
- e An OPTIONAL email address, whose value is a string and SHOULD be represented using [RFC5322] address semantics.
- p An OPTIONAL phone number whose value is a string, which SHOULD be represented as [E.164].
- c An OPTIONAL field containing connection data, whose value is an inner-list, and whose elements are each a string type matching the order of elements as defined in Section 5.7 of [RFC8866]. This field MUST be set if no media entities in the description contain connection information.
- b An OPTIONAL field containing the proposed bandwidth to be used by the session, whose value is an inner-list with only two elements, the first being a string type whose value corresponds to a "bwtype" as listed in the IANA registry, or a string prefixed "X-" to denote an experiemental value. The second element is an integer type and MUST NOT be negative.
- k The key-field field is obsolete and MUST NOT be used. Implementations MUST discard the field if it is received.

t An OPTIONAL field containing the start and end times of the session, whose value is an inner-list containing two elements - the first being the wall time start time, and the latter being the stopping time.

r An OPTIONAL field containing the repeat times, whose value is an inner-list containing three elements. The first element contains the repeat interval whose value is an integer, and the second element containing the active duration whose value is an integer. The third element of the offsets from start-time whose value is an inner-list containing two elements containing integer values representing the offsets.

z An OPTIONAL field containing time zone adjustment information, whose value is an inner-list containing elements where each is an inner-list with two elements; the first element being an integer representing the wall time which the adjustment should take place, and the second element whose value is an integer representing the offset.

a An OPTIONAL field containing session-level attributes, whose value is an inner-list. Each element may either be a sh-dictionary when representing a value attribute, or a string where it is a property attribute. For value attributes, the contents MUST be a single name/value pair with the name being the attribute name, and the value as a string.

3. The Session-Media Header

The Session-Media header describes each media element within the session, and at the top level is an sh-list.

Each media representation may additionally contain a media title (i), connection information (c), or bandwidth (b).

3.1. Implementation Considerations

The Session-Description header MAY be sent at the same time in a response with a HTTP body that also contains the SDP payload for backwards compatibility. In such case the values of the header MUST be identical in semantic meaning to the body payload and not include additional information or redaction. It may also, dependant on implementation be sent in response to a HEAD request - in such cases the body MUST be omitted but the server MUST also send the "application/sdp" "Content-Type" HTTP header.

3.2. Character set usage

TODO: Cover character sets

4. Examples

TODO: Examples

5. IANA Considerations

This specification registers the following entry in the Permanent Message Header Field Names registry established by [RFC3864]:

- Header field name: Session-Description
- Applicable protocol: http
- Status: standard
- Author/Change Controller: IETF
- Specification document(s): [this document]
- Related information:
- Header field name: Session-Media
- Applicable protocol: http
- Status: standard
- Author/Change Controller: IETF
- Specification document(s): [this document]
- Related information:

6. Security Considerations

TODO: Incorporate things like secure transport (HTTPS), in addition to considerations raised in the structured header draft.

7. References

7.1. Normative References

[E.164] "The international public telecommunication numbering plan", November 2010, <https://www.itu.int/rec/dologin_pub.asp?lang=e&id=T-REC-E.164-201011-I!!PDF-E&type=items>.

[I-D.ietf-httpbis-header-structure]
Nottingham, M. and P. Kamp, "Structured Field Values for HTTP", draft-ietf-httpbis-header-structure-19 (work in progress), June 2020.

[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>.

[RFC3864] Klyne, G., Nottingham, M., and J. Mogul, "Registration Procedures for Message Header Fields", BCP 90, RFC 3864, DOI 10.17487/RFC3864, September 2004, <https://www.rfc-editor.org/info/rfc3864>.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

[RFC5322] Resnick, P., Ed., "Internet Message Format", RFC 5322, DOI 10.17487/RFC5322, October 2008, <https://www.rfc-editor.org/info/rfc5322>.

[RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <https://www.rfc-editor.org/info/rfc8174>.

[RFC8866] Begen, A., Kyzivat, P., Perkins, C., and M. Handley, "SDP: Session Description Protocol", RFC 8866, DOI 10.17487/RFC8866, January 2021, <https://www.rfc-editor.org/info/rfc8866>.

7.2. URIs

[1] https://github.com/fiestajetsam/I-D/tree/main/draft-gruessing-sdp-http

Acknowledgements

The author would like to thank Colin Perkins for very early feedback on this document.

Author's Address

James Gruessing

Email: james.ietf@gmail.com

QUIC
Internet-Draft
Intended status: Standards Track
Expires: 6 May 2021

R. Marx
Hasselt University

2 November 2020

QUIC and HTTP/3 event definitions for qlog
draft-marx-qlog-event-definitions-quic-h3-02

Abstract

This document describes concrete qlog event definitions and their metadata for QUIC and HTTP/3-related events. These events can then be embedded in the higher level schema defined in [QLOG-MAIN].

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 6 May 2021.

Copyright Notice

Table of Contents

1. Introduction

This document describes the values of the qlog name ("category" + "event") and "data" fields and their semantics for the QUIC and HTTP/3 protocols. This document is based on draft-29 of the QUIC and HTTP/3 I-Ds QUIC-TRANSPORT [QUIC-HTTP] and draft-16 of the QPACK I-D [QUIC-QPACK].

Feedback and discussion welcome at https://github.com/quiclog/internet-drafts (https://github.com/quiclog/internet-drafts). Readers are advised to refer to the "editor's draft" at that URL for an up-to-date version of this document.

Concrete examples of integrations of this schema in various programming languages can be found at https://github.com/quiclog/qlog/ (https://github.com/quiclog/qlog/).

1.1. Notational Conventions

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

The examples and data definitions in ths document are expressed in a custom data definition language, inspired by JSON and TypeScript, and described in [QLOG-MAIN].

2. Overview

This document describes the values of the qlog "name" ("category" + "event") and "data" fields and their semantics for the QUIC and HTTP/3 protocols.

This document assumes the usage of the encompassing main qlog schema defined in [QLOG-MAIN]. Each subsection below defines a separate category (for example connectivity, transport, http) and each subsubsection is an event type (for example "packet_received").

For each event type, its importance and data definition is laid out, often accompanied by possible values for the optional "trigger" field. For the definition and semantics of "trigger", see the main schema document.

Most of the complex datastructures, enums and re-usable definitions are grouped together on the bottom of this document for clarity.

2.1. Importance

Many of the events defined in this document map directly to concepts seen in the QUIC and HTTP/3 documents, while others act as aggregating events that combine data from several possible protocol behaviours or code paths into one. This is done to reduce the amount of unique event definitions, as reflecting each possible protocol event as a separate qlog entity would cause an explosion of event types. Similarly, we prevent logging duplicate packet data as much as possible. As such, especially packet header value updates are split out into separate events (for example spin_bit_updated, connection_id_updated), as they are expected to change sparingly.

Consequently, many events that can be directly inferred from data on the wire (for example flow control limit changes) if the implementation is bug-free, are currently not explicitly defined as stand-alone events. Exceptions can be made for common events that benefit from being easily identifiable or individually logged (for example the "packets_acked" event). This can in turn give rise to separate events logging similar data, where it is not always clear which event should be logged (for example the separate "connection_started" event, whereas the more general "connection_state_updated" event also allows indicating that a connection was started).

To aid in this decision making, each event has an "importance indicator" with one of three values, in decreasing order of importance and exptected usage:

* Core

* Base

* Extra

The "Core" events are the events that SHOULD be present in all qlog files. These are mostly tied to basic packet and frame parsing and creation, as well as listing basic internal metrics. Tool implementers SHOULD expect and add support for these events, though SHOULD NOT expect all Core events to be present in each qlog trace.

The "Base" events add additional debugging options and CAN be present in qlog files. Most of these can be implicitly inferred from data in Core events (if those contain all their properties), but for many it is better to log the events explicitly as well, making it clearer how the implementation behaves. These events are for example tied to passing data around in buffers, to how internal state machines change and help show when decisions are actually made based on received data. Tool implementers SHOULD at least add support for showing the contents of these events, if they do not handle them explicitly.

The "Extra" events are considered mostly useful for low-level debugging of the implementation, rather than the protocol. They allow more fine-grained tracking of internal behaviour. As such, they CAN be present in qlog files and tool implementers CAN add support for these, but they are not required to.

Note that in some cases, implementers might not want to log for example frame-level details in the "Core" events due to performance or privacy considerations. In this case, they SHOULD use (a subset of) relevant "Base" events instead to ensure usability of the qlog output. As an example, implementations that do not log "packet_received" events and thus also not which (if any) ACK frames the packet contain, SHOULD log "packets_acked" events instead.

Finally, for event types who's data (partially) overlap with other event types' definitions, where necessary this document includes guidance on which to use in specific situations.

2.2. Custom fields

Note that implementers are free to define new category and event types, as well as values for the "trigger" property within the "data" field, or other member fields of the "data" field, as they see fit. They SHOULD NOT however expect non-specialized tools to recognize or visualize this custom data. However, tools SHOULD make an effort to visualize even unknown data if possible in the specific tool's context.

3. Events not belonging to a single connection

For several types of events, it is sometimes impossible to tie them to a specific conceptual QUIC connection (e.g., a packet_dropped event triggered because the packet has an unknown connection_id in the header). Since qlog events in a trace are typically associated with a single connection, it is unclear how to log these events.

Ideally, implementers SHOULD create a separate, individual "endpoint-level" trace file (or group_id value), not associated with a specific connection (for example a "server.qlog" or group_id = "client"), and log all events that do not belong to a single connection to this grouping trace. However, this is not always practical, depending on the implementation. Because the semantics of most of these events are well-defined in the protocols and because they are difficult to mis-interpret as belonging to a connection, implementers MAY choose to log events not belonging to a particular connection in any other trace, even those strongly associated with a single connection.

Note that this can make it difficult to match logs from different vantage points with each other. For example, from the client side, it is easy to log connections with version negotiation or retry in the same trace, while on the server they would most likely be logged in separate traces. Servers can take extra efforts (and keep additional state) to keep these events combined in a single trace however (for example by also matching connections on their four-tuple instead of just the connection ID).

4. QUIC and HTTP/3 fields

This document re-uses all the fields defined in the main qlog schema (e.g., name, category, type, data, group_id, protocol_type, the time-related fields, etc.).

The value of the "protocol_type" qlog field MUST be "QUIC_HTTP3".

When the qlog "group_id" field is used, it is recommended to use QUIC's Original Destination Connection ID (ODCID, the CID chosen by the client when first contacting the server), as this is the only value that does not change over the course of the connection and can be used to link more advanced QUIC packets (e.g., Retry, Version Negotiation) to a given connection. Similarly, the ODCID should be used as the qlog filename or file identifier, potentially suffixed by the vantagepoint type (For example, abcd1234_server.qlog would contain the server-side trace of the connection with ODCID abcd1234).

4.1. Raw packet and frame information

While qlog is a more high-level logging format, it also allows the inclusion of most raw wire image information, such as byte lengths and even raw byte values. This can be useful when for example investigating or tuning packetization behaviour or determining encoding/framing overheads. However, these fields are not always necessary and can take up considerable space if logged for each packet or frame. As such, they are grouped in a separate optional field called "raw" of type RawInfo (where applicable).

```
class RawInfo {
    length?:uint64; // full packet/frame length, including header and AEAD authen
tication tag lengths (where applicable)
    payload_length?:uint64; // length of the packet/frame payload, excluding AEAD
 tag. For many control frames, this will have a value of zero

    data?:bytes; // full packet/frame contents, including header and AEAD authent
ication tag (where applicable)
}
```

Note: QUIC packets always include an AEAD authentication tag at the end. As this tag is always the same size for a given connection (it depends on the used TLS cipher), we do not have a separate "aead_tag_length" field here. Instead, this field is reflected in "transport:parameters_set" and can be logged only once.

Note: There is intentionally no explicit header_length field in RawInfo. QUIC and HTTP/3 use many Variable-Length Integer Encoded (VLIE) values in their packet and frame headers, which are of a dynamic length. Note too that because of this, we cannot deterministally reconstruct the header encoding/length from qlog data, as implementations might not necessarily employ the most efficient VLIE scheme for all values. As such, it is typically easier to log just the total packet/frame length and the payload length. The header length can be calculated by tools as:

For QUIC packets: header_length = length - payload_length - aead_tag_length

For QUIC and HTTP/3 frames: header_length = length - payload_length

For UDP datagrams: header_length = length - payload_length

Note: In some cases, the length fields are also explicitly reflected inside of frame/packet headers. For example, the QUIC STREAM frame has a "length" field indicating its payload size. Similarly, all HTTP/3 frames include their explicit payload lengths in the frame header. Finally, the QUIC Long Header has a "length" field which is equal to the payload length plus the packet number length. In these cases, those fields are intentionally preserved in the event definitions. Even though this can lead to duplicate data when the full RawInfo is logged, it allows a more direct mapping of the QUIC and HTTP/3 specifications to qlog, making it easier for users to interpret.

Note: as described in [QLOG-MAIN], the RawInfo:data field can be truncated for privacy or security purposes (for example excluding payload data). In this case, the length properties should still indicate the non-truncated lengths.

5. QUIC event definitions

Each subheading in this section is a qlog event category, while each sub-subheading is a qlog event type. Concretely, for the following two items, we have the category "connectivity" and event type "server_listening", resulting in a concatenated qlog "name" field value of "connectivity:server_listening".

5.1. connectivity

5.1.1. server_listening

Importance: Extra

Emitted when the server starts accepting connections.

Data:

```
{
    ip_v4?: IPAddress,
    ip_v6?: IPAddress,
    port_v4?: uint32,
    port_v6?: uint32,

    retry_required?:boolean // the server will always answer client initials with
 a retry (no 1-RTT connection setups by choice)
}
```

Note: some QUIC stacks do not handle sockets directly and are thus unable to log IP and/or port information.

5.1.2. connection_started

Importance: Base

Used for both attempting (client-perspective) and accepting (server-perspective) new connections. Note that this event has overlap with connection_state_updated and this is a separate event mainly because of all the additional data that should be logged.

Data:

```
{
    ip_version?: "v4" | "v6",
    src_ip?: IPAddress,
    dst_ip?: IPAddress,

    protocol?: string, // transport layer protocol (default "QUIC")
    src_port?: uint32,
    dst_port?: uint32,

    src_cid?: bytes,
    dst_cid?: bytes,

}
```

Note: some QUIC stacks do not handle sockets directly and are thus unable to log IP and/or port information.

5.1.3. connection_closed

Importance: Base

Used for logging when a connection was closed, typically when an error or timeout occurred. Note that this event has overlap with connectivity:connection_state_updated, as well as the CONNECTION_CLOSE frame. However, in practice, when analyzing large deployments, it can be useful to have a single event representing a connection_closed event, which also includes an additional reason field to provide additional information. Additionally, it is useful to log closures due to timeouts, which are difficult to reflect using the other options.

In QUIC there are two main connection-closing error categories: connection and application errors. They have well-defined error codes and semantics. Next to these however, there can be internal errors that occur that may or may not get mapped to the official error codes in implementation-specific ways. As such, multiple error codes can be set on the same event to reflect this.

```
{
    owner?:"local"|"remote", // which side closed the connection

    connection_code?:TransportError | CryptoError | uint32,
    application_code?:ApplicationError | uint32,
    internal_code?:uint32,

    reason?:string
}
```

Triggers: * clean * handshake_timeout * idle_timeout * error // this is called the "immediate close" in the QUIC specification * stateless_reset * version_mismatch * application // for example HTTP/3's GOAWAY frame

5.1.4. connection_id_updated

Importance: Base

This event is emitted when either party updates their current Connection ID. As this typically happens only sparingly over the course of a connection, this event allows loggers to be more efficient than logging the observed CID with each packet in the .header field of the "packet_sent" or "packet_received" events.

This is viewed from the perspective of the one applying the new id. As such, if we receive a new connection id from our peer, we will see the dst_ fields are set. If we update our own connection id (e.g., NEW_CONNECTION_ID frame), we log the src_ fields.

Data:

```
{
    owner: "local" | "remote",

    old?:bytes,
    new?:bytes,
}
```

5.1.5. spin_bit_updated

Importance: Base

To be emitted when the spin bit changes value. It SHOULD NOT be emitted if the spin bit is set without changing its value.

Data:

```
{
    state: boolean
}
```

5.1.6. connection_retried

TODO

5.1.7. connection_state_updated

Importance: Base

This event is used to track progress through QUIC's complex handshake and connection close procedures. It is intended to provide exhaustive options to log each state individually, but also provides a more basic, simpler set for implementations less interested in tracking each smaller state transition. As such, users should not expect to see -all- these states reflected in all qlogs and implementers should focus on support for the SimpleConnectionState set.

Data: ~~~ { old?: ConnectionState | SimpleConnectionState, new: ConnectionState | SimpleConnectionState }

enum ConnectionState { attempted, // initial sent/received peer_validated, // peer address validated by: client sent Handshake packet OR client used CONNID chosen by the server. transport-draft-32, section-8.1 handshake_started, early_write, // 1 RTT can be sent, but handshake isn't done yet handshake_complete, // TLS handshake complete: Finished received and sent. tls-draft-32, section-4.1.1 handshake_confirmed, // HANDSHAKE_DONE sent/received (connection is now "active", 1RTT can be sent). tls-draft-32, section-4.1.2 closing, draining, // connection_close sent/received closed // draining period done, connection state discarded }

enum SimpleConnectionState { attempted, handshake_started, handshake_confirmed, closed } ~~~

These states correspond to the following transitions for both client and server:

Client:

* send initial
  - state = attempted
* get initial
  - state = validated _(not really "needed" at the client, but somewhat useful to indicate progress nonetheless)_
* get first Handshake packet
  - state = handshake_started

* get Handshake packet containing ServerFinished
  - state = handshake_complete
* send ClientFinished
  - state = early_write (1RTT can now be sent)
* get HANDSHAKE_DONE
  - state = handshake_confirmed

Server:

* get initial
  - state = attempted
* send initial _(don't think this needs a separate state, since some handshake will always be sent in the same flight as this?)_
* send handshake EE, CERT, CV, ...
  - state = handshake_started
* send ServerFinished
  - state = early_write (1RTT can now be sent)
* get first handshake packet / something using a server-issued CID of min length
  - state = validated
* get handshake packet containing ClientFinished
  - state = handshake_complete
* send HANDSHAKE_DONE
  - state = handshake_confirmed

Note: connection_state_changed with a new state of "attempted" is the same conceptual event as the connection_started event above from the client's perspective. Similarly, a state of "closing" or "draining" corresponds to the connection_closed event.

5.1.8. MIGRATION-related events

e.g., path_updated

TODO: read up on the draft how migration works and whether to best fit this here or in TRANSPORT TODO: integrate https://tools.ietf.org/html/draft-deconinck-quic-multipath-02

For now, infer from other connectivity events and path_challenge/ path_response frames

5.2. security

5.2.1. key_updated

Importance: Base

Note: secret_updated would be more correct, but in the draft it's called KEY_UPDATE, so stick with that for consistency

Data:

```
{
    key_type:KeyType,
    old?:bytes,
    new:bytes,
    generation?:uint32 // needed for 1RTT key updates
}
```

Triggers:

* "tls" // (e.g., initial, handshake and 0-RTT keys are generated by TLS)

* "remote_update"

* "local_update"

5.2.2. key_retired

Importance: Base

Data:

```
{
    key_type:KeyType,
    key?:bytes,
    generation?:uint32 // needed for 1RTT key updates
}
```

Triggers:

* "tls" // (e.g., initial, handshake and 0-RTT keys are dropped implicitly)

* "remote_update"

* "local_update"

5.3. transport

5.3.1. version_information

Importance: Core

QUIC endpoints each have their own list of of QUIC versions they support. The client uses the most likely version in their first initial. If the server does support that version, it replies with a version_negotiation packet, containing supported versions. From this, the client selects a version. This event aggregates all this information in a single event type. It also allows logging of supported versions at an endpoint without actual version negotiation needing to happen.

Data:

```
{
    server_versions?:Array<bytes>,
    client_versions?:Array<bytes>,
    chosen_version?:bytes
}
```

Intended use:

* When sending an initial, the client logs this event with client_versions and chosen_version set

* Upon receiving a client initial with a supported version, the server logs this event with server_versions and chosen_version set

* Upon receiving a client initial with an unsupported version, the server logs this event with server_versions set and client_versions to the single-element array containing the client's attempted version. The absence of chosen_version implies no overlap was found.

* Upon receiving a version negotiation packet from the server, the client logs this event with client_versions set and server_versions to the versions in the version negotiation packet and chosen_version to the version it will use for the next initial packet

5.3.2. alpn_information

Importance: Core

QUIC implementations each have their own list of application level protocols and versions thereof they support. The client includes a list of their supported options in its first initial as part of the TLS Application Layer Protocol Negotiation (alpn) extension. If there are common option(s), the server chooses the most optimal one and communicates this back to the client. If not, the connection is closed.

Data:

```
{
    server_alpns?:Array<string>,
    client_alpns?:Array<string>,
    chosen_alpn?:string
}
```

Intended use:

* When sending an initial, the client logs this event with client_alpns set

* When receiving an initial with a supported alpn, the server logs this event with server_alpns set, client_alpns equalling the client-provided list, and chosen_alpn to the value it will send back to the client.

* When receiving an initial with an alpn, the client logs this event with chosen_alpn to the received value.

* Alternatively, a client can choose to not log the first event, but wait for the receipt of the server initial to log this event with both client_alpns and chosen_alpn set.

5.3.3. parameters_set

Importance: Core

This event groups settings from several different sources (transport parameters, TLS ciphers, etc.) into a single event. This is done to minimize the amount of events and to decouple conceptual setting impacts from their underlying mechanism for easier high-level reasoning.

All these settings are typically set once and never change. However, they are typically set at different times during the connection, so there will typically be several instances of this event with different fields set.

Note that some settings have two variations (one set locally, one requested by the remote peer). This is reflected in the "owner" field. As such, this field MUST be correct for all settings included a single event instance. If you need to log settings from two sides, you MUST emit two separate event instances.

In the case of connection resumption and 0-RTT, some of the server's parameters are stored up-front at the client and used for the initial connection startup. They are later updated with the server's reply. In these cases, utilize the separate "parameters_restored" event to indicate the initial values, and this event to indicate the updated values, as normal.

Data:

```
{
    owner?:"local" | "remote",

    resumption_allowed?:boolean, // valid session ticket was received
    early_data_enabled?:boolean, // early data extension was enabled on the TLS l
ayer
    tls_cipher?:string, // (e.g., "AES_128_GCM_SHA256")
    aead_tag_length?:uint8, // depends on the TLS cipher, but it's easier to be e
xplicit. Default value is 16

    // transport parameters from the TLS layer:
    original_destination_connection_id?:bytes,
    initial_source_connection_id?:bytes,
    retry_source_connection_id?:bytes,
    stateless_reset_token?:Token,
    disable_active_migration?:boolean,

    max_idle_timeout?:uint64,
    max_udp_payload_size?:uint32,
    ack_delay_exponent?:uint16,
    max_ack_delay?:uint16,
    active_connection_id_limit?:uint32,

    initial_max_data?:uint64,
    initial_max_stream_data_bidi_local?:uint64,
    initial_max_stream_data_bidi_remote?:uint64,
    initial_max_stream_data_uni?:uint64,
    initial_max_streams_bidi?:uint64,
    initial_max_streams_uni?:uint64,

    preferred_address?:PreferredAddress
}

interface PreferredAddress {
    ip_v4:IPAddress,
    ip_v6:IPAddress,

    port_v4:uint16,
    port_v6:uint16,

    connection_id:bytes,
    stateless_reset_token:Token
}
```

Additionally, this event can contain any number of unspecified fields. This is to reflect setting of for example unknown (greased) transport parameters or employed (proprietary) extensions.

5.3.4. parameters_restored

Importance: Base

When using QUIC 0-RTT, clients are expected to remember and restore the server's transport parameters from the previous connection. This event is used to indicate which parameters were restored and to which values when utilizing 0-RTT. Note that not all transport parameters should be restored (many are even prohibited from being re-utilized). The ones listed here are the ones expected to be useful for correct 0-RTT usage.

Data:

```
{
    disable_active_migration?:boolean,

    max_idle_timeout?:uint64,
    max_udp_payload_size?:uint32,
    active_connection_id_limit?:uint32,

    initial_max_data?:uint64,
    initial_max_stream_data_bidi_local?:uint64,
    initial_max_stream_data_bidi_remote?:uint64,
    initial_max_stream_data_uni?:uint64,
    initial_max_streams_bidi?:uint64,
    initial_max_streams_uni?:uint64,
}
```

Note that, like parameters_set above, this event can contain any number of unspecified fields to allow for additional/custom parameters.

5.3.5. packet_sent

Importance: Core

Data:

```
{
    header:PacketHeader,

    frames?:Array<QuicFrame>, // see appendix for the definitions

    is_coalesced?:boolean, // default value is false

    retry_token?:Token, // only if header.packet_type === retry

    stateless_reset_token?:bytes, // only if header.packet_type === stateless_res
et. Is always 128 bits in length.

    supported_versions:Array<bytes>, // only if header.packet_type === version_ne
gotiation

    raw?:RawInfo,
    datagram_id?:uint32
}
```

Note: We do not explicitly log the encryption_level or packet_number_space: the header.packet_type specifies this by inference (assuming correct implementation)

Triggers:

* "retransmit_reordered" // draft-23 5.1.1

* "retransmit_timeout" // draft-23 5.1.2

* "pto_probe" // draft-23 5.3.1

* "retransmit_crypto" // draft-19 6.2

* "cc_bandwidth_probe" // needed for some CCs to figure out bandwidth allocations when there are no normal sends

Note: for more details on "datagram_id", see Section 5.3.10. It is only needed when keeping track of packet coalescing.

5.3.6. packet_received

Importance: Core

Data:

```
{
    header:PacketHeader,

    frames?:Array<QuicFrame>, // see appendix for the definitions

    is_coalesced?:boolean,

    retry_token?:Token, // only if header.packet_type === retry

    stateless_reset_token?:bytes, // only if header.packet_type === stateless_res
et. Is always 128 bits in length.

    supported_versions:Array<bytes>, // only if header.packet_type === version_ne
gotiation

    raw?:RawInfo,
    datagram_id?:uint32
}
```

Note: We do not explicitly log the encryption_level or packet_number_space: the header.packet_type specifies this by inference (assuming correct implementation)

Triggers:

* "keys_available" // if packet was buffered because it couldn't be decrypted before

Note: for more details on "datagram_id", see Section 5.3.10. It is only needed when keeping track of packet coalescing.

5.3.7. packet_dropped

Importance: Base

This event indicates a QUIC-level packet was dropped after partial or no parsing.

Data:

```
{
    header?:PacketHeader, // primarily packet_type should be filled here, as othe
r fields might not be parseable

    raw?:RawInfo,
    datagram_id?:uint32
}
```

For this event, the "trigger" field SHOULD be set (for example to one of the values below), as this helps tremendously in debugging.

Triggers:

* "key_unavailable"
* "unknown_connection_id"
* "header_parse_error"
* "payload_decrypt_error"
* "protocol_violation"
* "dos_prevention"
* "unsupported_version"
* "unexpected_packet"
* "unexpected_source_connection_id"
* "unexpected_version"
* "duplicate"
* "invalid_initial"

Note: sometimes packets are dropped before they can be associated with a particular connection (e.g., in case of "unsupported_version"). This situation is discussed more in Section 3.

Note: for more details on "datagram_id", see Section 5.3.10. It is only needed when keeping track of packet coalescing.

5.3.8. packet_buffered

Importance: Base

This event is emitted when a packet is buffered because it cannot be processed yet. Typically, this is because the packet cannot be parsed yet, and thus we only log the full packet contents when it was parsed in a packet_received event.

Data:

```
{
    header?:PacketHeader, // primarily packet_type and possible packet_number sho
uld be filled here, as other elements might not be available yet

    raw?:RawInfo,
    datagram_id?:uint32
}
```

Note: for more details on "datagram_id", see Section 5.3.10. It is only needed when keeping track of packet coalescing.

Triggers:

* "backpressure" // indicates the parser cannot keep up, temporarily buffers packet for later processing

* "keys_unavailable" // if packet cannot be decrypted because the proper keys were not yet available

5.3.9. packets_acked

Importance: Extra

This event is emitted when a (group of) sent packet(s) is acknowledged by the remote peer _for the first time_. This information could also be deduced from the contents of received ACK frames. However, ACK frames require additional processing logic to determine when a given packet is acknowledged for the first time, as QUIC uses ACK ranges which can include repeated ACKs. Additionally, this event can be used by implementations that do not log frame contents.

Data: ~~~ { packet_number_space?:PacketNumberSpace,

packet_numbers?:Array<uint64> } ~~~

Note: if packet_number_space is omitted, it assumes the default value of PacketNumberSpace.application_data, as this is by far the most prevalent packet number space a typical QUIC connection will use.

5.3.10. datagrams_sent

Importance: Extra

When we pass one or more UDP-level datagrams to the socket. This is useful for determining how QUIC packet buffers are drained to the OS.

Data:

```
{
    count?:uint16, // to support passing multiple at once
    raw?:Array<RawInfo>, // RawInfo:length field indicates total length of the da
tagrams, including UDP header length

    datagram_ids?:Array<uint32>
}
```

Note: QUIC itself does not have a concept of a "datagram_id". This field is a purely qlog-specific construct to allow tracking how multiple QUIC packets are coalesced inside of a single UDP datagram, which is an important optimization during the QUIC handshake. For this, implementations assign a (per-endpoint) unique ID to each datagram and keep track of which packets were coalesced into the same datagram. As packet coalescing typically only happens during the handshake (as it requires at least one long header packet), this can be done without much overhead.

5.3.11. datagrams_received

Importance: Extra

When we receive one or more UDP-level datagrams from the socket. This is useful for determining how datagrams are passed to the user space stack from the OS.

Data:

```
{
    count?:uint16, // to support passing multiple at once
    raw?:Array<RawInfo>, // RawInfo:length field indicates total length of the da
tagrams, including UDP header length

    datagram_ids?:Array<uint32>
}
```

Note: for more details on "datagram_ids", see Section 5.3.10.

5.3.12. datagram_dropped

Importance: Extra

When we drop a UDP-level datagram. This is typically if it does not contain a valid QUIC packet (in that case, use packet_dropped instead).

Data:

```
{
    raw?:RawInfo
}
```

5.3.13. stream_state_updated

Importance: Base

This event is emitted whenever the internal state of a QUIC stream is updated, as described in QUIC transport draft-23 section 3. Most of this can be inferred from several types of frames going over the wire, but it's much easier to have explicit signals for these state changes.

Data:

```
{
    stream_id:uint64,
    stream_type?:"unidirectional"|"bidirectional", // mainly useful when opening
the stream

    old?:StreamState,
    new:StreamState,

    stream_side?:"sending"|"receiving"
}

enum StreamState {
    // bidirectional stream states, draft-23 3.4.
    idle,
    open,
    half_closed_local,
    half_closed_remote,
    closed,

    // sending-side stream states, draft-23 3.1.
    ready,
    send,
    data_sent,
    reset_sent,
    reset_received,

    // receive-side stream states, draft-23 3.2.
    receive,
    size_known,
    data_read,
    reset_read,

    // both-side states
    data_received,

    // qlog-defined
    destroyed // memory actually freed
}
```

   Note: QUIC implementations SHOULD mainly log the simplified bidirectional (HTTP/2-alike) stream states (e.g., idle, open, closed) instead of the more finegrained stream states (e.g., data_sent, reset_received). These latter ones are mainly for more in-depth debugging. Tools SHOULD be able to deal with both types equally.

5.3.14. frames_processed

   Importance: Extra

This event's main goal is to prevent a large proliferation of specific purpose events (e.g., packets_acknowledged, flow_control_updated, stream_data_received). We want to give implementations the opportunity to (selectively) log this type of signal without having to log packet-level details (e.g., in packet_received). Since for almost all cases, the effects of applying a frame to the internal state of an implementation can be inferred from that frame's contents, we aggregate these events in this single "frames_processed" event.

Note: This event can be used to signal internal state change not resulting directly from the actual "parsing" of a frame (e.g., the frame could have been parsed, data put into a buffer, then later processed, then logged with this event).

Note: Implementations logging "packet_received" and which include all of the packet's constituent frames therein, are not expected to emit this "frames_processed" event (contrary to the HTTP-level "frames_parsed" event). Rather, implementations not wishing to log full packets or that wish to explicitly convey extra information about when frames are processed (if not directly tied to their reception) can use this event.

Note: for some events, this approach will lose some information (e.g., for which encryption level are packets being acknowledged?). If this information is important, please use the packet_received event instead.

Note: in some implementations, it can be difficult to log frames directly, even when using packet_sent and packet_received events. For these cases, this event also contains the direct packet_number field, which can be used to more explicitly link this event to the packet_sent/received events.

Data:

```
{
    frames:Array<QuicFrame>, // see appendix for the definitions

    packet_number?:uint64
}
```

5.3.15. data_moved

Importance: Base

Used to indicate when data moves between the different layers (for example passing from HTTP/3 to QUIC stream buffers and vice versa) or between HTTP/3 and the actual user application on top (for example a browser engine). This helps make clear the flow of data, how long data remains in various buffers and the overheads introduced by individual layers.

For example, this helps make clear whether received data on a QUIC stream is moved to the HTTP layer immediately (for example per received packet) or in larger batches (for example, all QUIC packets are processed first and afterwards the HTTP layer reads from the streams with newly available data). This in turn can help identify bottlenecks or scheduling problems.

Data:

```
{
    stream_id?:uint64,
    offset?:uint64,
    length?:uint64, // byte length of the moved data

    from?:string, // typically: use either of "application","http","transport"
    to?:string, // typically: use either of "application","http","transport"

    data?:bytes // raw bytes that were transferred
}
```

Note: we do not for example use a "direction" field (with values "up" and "down") to specify the data flow. This is because in some optimized implementations, data might skip some individual layers. Additionally, using explicit "from" and "to" fields is more flexible and allows the definition of other conceptual "layers" (for example to indicate data from QUIC CRYPTO frames being passed to a TLS library ("security") or from HTTP/3 to QPACK ("qpack")).

Note: this event type is part of the "transport" category, but really spans all the different layers. This means we have a few leaky abstractions here (for example, the stream_id or stream offset might not be available at some logging points, or the raw data might not be in a byte-array form). In these situations, implementers can decide to define new, in-context fields to aid in manual debugging.

5.4. recovery

Note: most of the events in this category are kept generic to support different recovery approaches and various congestion control algorithms. Tool creators SHOULD make an effort to support and visualize even unknown data in these events (e.g., plot unknown congestion states by name on a timeline visualization).

5.4.1. parameters_set

Importance: Base

This event groups initial parameters from both loss detection and congestion control into a single event. All these settings are typically set once and never change. Implementation that do, for some reason, change these parameters during execution, MAY emit the parameters_set event twice.

Data:

```
{
    // Loss detection, see recovery draft-23, Appendix A.2
    reordering_threshold?:uint16, // in amount of packets
    time_threshold?:float, // as RTT multiplier
    timer_granularity?:uint16, // in ms
    initial_rtt?:float, // in ms

    // congestion control, Appendix B.1.
    max_datagram_size?:uint32, // in bytes // Note: this could be updated after p
mtud
    initial_congestion_window?:uint64, // in bytes
    minimum_congestion_window?:uint32, // in bytes // Note: this could change whe
n max_datagram_size changes
    loss_reduction_factor?:float,
    persistent_congestion_threshold?:uint16 // as PTO multiplier
}
```

Additionally, this event can contain any number of unspecified fields to support different recovery approaches.

5.4.2. metrics_updated

Importance: Core

This event is emitted when one or more of the observable recovery metrics changes value. This event SHOULD group all possible metric updates that happen at or around the same time in a single event (e.g., if min_rtt and smoothed_rtt change at the same time, they should be bundled in a single metrics_updated entry, rather than split out into two). Consequently, a metrics_updated event is only guaranteed to contain at least one of the listed metrics.

Data:

```
{
    // Loss detection, see recovery draft-23, Appendix A.3
    min_rtt?:float, // in ms or us, depending on the overarching qlog's configura
tion
    smoothed_rtt?:float, // in ms or us, depending on the overarching qlog's conf
iguration
    latest_rtt?:float, // in ms or us, depending on the overarching qlog's config
uration
    rtt_variance?:float, // in ms or us, depending on the overarching qlog's conf
iguration

    pto_count?:uint16,

    // Congestion control, Appendix B.2.
    congestion_window?:uint64, // in bytes
    bytes_in_flight?:uint64,

    ssthresh?:uint64, // in bytes

    // qlog defined
    packets_in_flight?:uint64, // sum of all packet number spaces

    pacing_rate?:uint64 // in bps
}
```

Note: to make logging easier, implementations MAY log values even if they are the same as previously reported values (e.g., two subsequent METRIC_UPDATE entries can both report the exact same value for min_rtt). However, applications SHOULD try to log only actual updates to values.

Additionally, this event can contain any number of unspecified fields to support different recovery approaches.

5.4.3. congestion_state_updated

Importance: Base

This event signifies when the congestion controller enters a significant new state and changes its behaviour. This event's definition is kept generic to support different Congestion Control algorithms. For example, for the algorithm defined in the Recovery draft ("enhanced" New Reno), the following states are defined:

* slow_start
* congestion_avoidance
* application_limited
* recovery

Data:

```
{
    old?:string,
    new:string
}
```

The "trigger" field SHOULD be logged if there are multiple ways in which a state change can occur but MAY be omitted if a given state can only be due to a single event occuring (e.g., slow start is exited only when ssthresh is exceeded).

Some triggers for ("enhanced" New Reno):

* persistent_congestion
* ECN

5.4.4. loss_timer_updated

Importance: Extra

This event is emitted when a recovery loss timer changes state. The three main event types are:

* set: the timer is set with a delta timeout for when it will trigger next
* expired: when the timer effectively expires after the delta timeout
* cancelled: when a timer is cancelled (e.g., all outstanding packets are acknowledged, start idle period)

Note: to indicate an active timer's timeout update, a new "set" event is used.

Data:

```
{
    timer_type?:"ack"|"pto", // called "mode" in draft-23 A.9.
    packet_number_space?: PacketNumberSpace,

    event_type:"set"|"expired"|"cancelled",

    delta?:float // if event_type === "set": delta time in ms or us (see configur
ation) from this event's timestamp until when the timer will trigger
}
```

TODO: how about CC algo's that use multiple timers? How generic do these events need to be? Just support QUIC-style recovery from the spec or broader?

TODO: read up on the loss detection logic in draft-27 onward and see if this suffices

5.4.5. packet_lost

Importance: Core

This event is emitted when a packet is deemed lost by loss detection.

Data:

```
{
    header?:PacketHeader, // should include at least the packet_type and packet_n
umber

    // not all implementations will keep track of full packets, so these are opti
onal
    frames?:Array<QuicFrame> // see appendix for the definitions
}
```

For this event, the "trigger" field SHOULD be set (for example to one of the values below), as this helps tremendously in debugging.

Triggers:

* "reordering_threshold",

* "time_threshold"

* "pto_expired" // draft-23 section 5.3.1, MAY

5.4.6. marked_for_retransmit

Importance: Extra

This event indicates which data was marked for retransmit upon detecing a packet loss (see packet_lost). Similar to our reasoning for the "frames_processed" event, in order to keep the amount of different events low, we group this signal for all types of retransmittable data in a single event based on existing QUIC frame definitions.

Implementations retransmitting full packets or frames directly can just log the consituent frames of the lost packet here (or do away with this event and use the contents of the packet_lost event instead). Conversely, implementations that have more complex logic (e.g., marking ranges in a stream's data buffer as in-flight), or that do not track sent frames in full (e.g., only stream offset + length), can translate their internal behaviour into the appropriate frame instance here even if that frame was never or will never be put on the wire.

Note: much of this data can be inferred if implementations log packet_sent events (e.g., looking at overlapping stream data offsets and length, one can determine when data was retransmitted).

Data:

```
{
    frames:Array<QuicFrame>, // see appendix for the definitions
}
```

6. HTTP/3 event definitions

6.1. http

Note: like all category values, the "http" category is written in lowercase.

6.1.1. parameters_set

Importance: Base

This event contains HTTP/3 and QPACK-level settings, mostly those received from the HTTP/3 SETTINGS frame. All these parameters are typically set once and never change. However, they are typically set at different times during the connection, so there can be several instances of this event with different fields set.

Note that some settings have two variations (one set locally, one requested by the remote peer). This is reflected in the "owner" field. As such, this field MUST be correct for all settings included a single event instance. If you need to log settings from two sides, you MUST emit two separate event instances.

Data:

```
{
    owner?:"local" | "remote",

    max_header_list_size?:uint64, // from SETTINGS_MAX_HEADER_LIST_SIZE
    max_table_capacity?:uint64, // from SETTINGS_QPACK_MAX_TABLE_CAPACITY
    blocked_streams_count?:uint64, // from SETTINGS_QPACK_BLOCKED_STREAMS

    // qlog-defined
    waits_for_settings?:boolean // indicates whether this implementation waits fo
r a SETTINGS frame before processing requests
}
```

Note: enabling server push is not explicitly done in HTTP/3 by use of a setting or parameter. Instead, it is communicated by use of the MAX_PUSH_ID frame, which should be logged using the frame_created and frame_parsed events below.

Additionally, this event can contain any number of unspecified fields. This is to reflect setting of for example unknown (greased) settings or parameters of (proprietary) extensions.

6.1.2. parameters_restored

Importance: Base

When using QUIC 0-RTT, clients are expected to remember and reuse the server's SETTINGs from the previous connection. This event is used to indicate which settings were restored and to which values when utilizing 0-RTT.

Data:

```
{
    max_header_list_size?:uint64,
    max_table_capacity?:uint64,
    blocked_streams_count?:uint64
}
```

Note that, like for parameters_set above, this event can contain any number of unspecified fields to allow for additional and custom settings.

6.1.3. stream_type_set

Importance: Base

Emitted when a stream's type becomes known. This is typically when a stream is opened and the stream's type indicator is sent or received.

Note: most of this information can also be inferred by looking at a stream's id, since id's are strictly partitioned at the QUIC level. Even so, this event has a "Base" importance because it helps a lot in debugging to have this information clearly spelled out.

Data:

```
{
    stream_id:uint64,

    owner?:"local"|"remote"

    old?:StreamType,
    new:StreamType,

    associated_push_id?:uint64 // only when new == "push"
}

enum StreamType {
    data, // bidirectional request-response streams
    control,
    push,
    reserved,
    qpack_encode,
    qpack_decode
}
```

6.1.4. frame_created

Importance: Core

HTTP equivalent to the packet_sent event. This event is emitted when the HTTP/3 framing actually happens. Note: this is not necessarily the same as when the HTTP/3 data is passed on to the QUIC layer. For that, see the "data_moved" event.

Data:

```
{
    stream_id:uint64,
    length?:uint64, // payload byte length of the frame
    frame:HTTP3Frame, // see appendix for the definitions,

    raw?:RawInfo
}
```

Note: in HTTP/3, DATA frames can have arbitrarily large lengths to reduce frame header overhead. As such, DATA frames can span many QUIC packets and can be created in a streaming fashion. In this case, the frame_created event is emitted once for the frame header, and further streamed data is indicated using the data_moved event.

6.1.5. frame_parsed

Importance: Core

HTTP equivalent to the packet_received event. This event is emitted when we actually parse the HTTP/3 frame. Note: this is not necessarily the same as when the HTTP/3 data is actually received on the QUIC layer. For that, see the "data_moved" event.

Data:

```
{
    stream_id:uint64,
    length?:uint64, // payload byte length of the frame
    frame:HTTP3Frame, // see appendix for the definitions,

    raw?:RawInfo
}
```

Note: in HTTP/3, DATA frames can have arbitrarily large lengths to reduce frame header overhead. As such, DATA frames can span many QUIC packets and can be processed in a streaming fashion. In this case, the frame_parsed event is emitted once for the frame header, and further streamed data is indicated using the data_moved event.

6.1.6. push_resolved

Importance: Extra

This event is emitted when a pushed resource is successfully claimed (used) or, conversely, abandoned (rejected) by the application on top of HTTP/3 (e.g., the web browser). This event is added to help debug problems with unexpected PUSH behaviour, which is commonplace with HTTP/2.

```
{
    push_id?:uint64,
    stream_id?:uint64, // in case this is logged from a place that does not have
access to the push_id

    decision:"claimed"|"abandoned"
}
```

6.2. qpack

Note: like all category values, the "qpack" category is written in lowercase.

The QPACK events mainly serve as an aid to debug low-level QPACK issues. The higher-level, plaintext header values SHOULD (also) be logged in the http.frame_created and http.frame_parsed event data (instead).

Note: qpack does not have its own parameters_set event. This was merged with http.parameters_set for brevity, since qpack is a required extension for HTTP/3 anyway. Other HTTP/3 extensions MAY also log their SETTINGS fields in http.parameters_set or MAY define their own events.

6.2.1. state_updated

Importance: Base

This event is emitted when one or more of the internal QPACK variables changes value. Note that some variables have two variations (one set locally, one requested by the remote peer). This is reflected in the "owner" field. As such, this field MUST be correct for all variables included a single event instance. If you need to log settings from two sides, you MUST emit two separate event instances.

Data:

```
{
    owner:"local" | "remote",

    dynamic_table_capacity?:uint64,
    dynamic_table_size?:uint64, // effective current size, sum of all the entries

    known_received_count?:uint64,
    current_insert_count?:uint64
}
```

6.2.2. stream_state_updated

Importance: Core

This event is emitted when a stream becomes blocked or unblocked by header decoding requests or QPACK instructions.

Note: This event is of "Core" importance, as it might have a large impact on HTTP/3's observed performance.

Data:

```
{
    stream_id:uint64,

    state:"blocked"|"unblocked" // streams are assumed to start "unblocked" until
 they become "blocked"
}
```

6.2.3. dynamic_table_updated

Importance: Extra

This event is emitted when one or more entries are inserted or evicted from QPACK's dynamic table.

Data:

```
{
    owner:"local" | "remote", // local = the encoder's dynamic table. remote = th
e decoder's dynamic table

    update_type:"inserted"|"evicted",

    entries:Array<DynamicTableEntry>
}

class DynamicTableEntry {
    index:uint64;
    name?:string | bytes;
    value?:string | bytes;
}
```

6.2.4. headers_encoded

Importance: Base

This event is emitted when an uncompressed header block is encoded successfully.

Note: this event has overlap with http.frame_created for the HeadersFrame type. When outputting both events, implementers MAY omit the "headers" field in this event.

Data:

```
{
    stream_id?:uint64,

    headers?:Array<HTTPHeader>,

    block_prefix:QPackHeaderBlockPrefix,
    header_block:Array<QPackHeaderBlockRepresentation>,

    length?:uint32,
    raw?:bytes
}
```

6.2.5. headers_decoded

Importance: Base

This event is emitted when a compressed header block is decoded successfully.

Note: this event has overlap with http.frame_parsed for the HeadersFrame type. When outputting both events, implementers MAY omit the "headers" field in this event.

Data:

```
{
    stream_id?:uint64,

    headers?:Array<HTTPHeader>,

    block_prefix:QPackHeaderBlockPrefix,
    header_block:Array<QPackHeaderBlockRepresentation>,

    length?:uint32,
    raw?:bytes
}
```

6.2.6. instruction_created

Importance: Base

This event is emitted when a QPACK instruction (both decoder and encoder) is created and added to the encoder/decoder stream.

Data:

```
{
    instruction:QPackInstruction // see appendix for the definitions,

    length?:uint32,
    raw?:bytes
}
```

Note: encoder/decoder semantics and stream_id's are implicit in either the instruction types or can be logged via other events (e.g., http.stream_type_set)

6.2.7. instruction_parsed

Importance: Base

This event is emitted when a QPACK instruction (both decoder and encoder) is read from the encoder/decoder stream.

Data:

```
{
    instruction:QPackInstruction // see appendix for the definitions,

    length?:uint32,
    raw?:bytes
}
```

Note: encoder/decoder semantics and stream_id's are implicit in either the instruction types or can be logged via other events (e.g., http.stream_type_set)

7. Generic events and Simulation indicators

7.1. generic

The main goal of the events in this category is to allow implementations to fully replace their existing text-based logging by qlog. This is done by providing events to log generic strings for typical well-known logging levels (error, warning, info, debug, verbose).

7.1.1. error

Importance: Core

Used to log details of an internal error. For errors that effectively lead to the closure of a QUIC connection, it is recommended to use transport:connection_closed instead.

Data:

```
{
    code?:uint32,
    message?:string
}
```

7.1.2. warning

Importance: Base

Used to log details of an internal warning that might not get reflected on the wire.

Data:

```
{
    code?:uint32,
    message?:string
}
```

7.1.3. info

Importance: Extra

Used mainly for implementations that want to use qlog as their one and only logging format but still want to support unstructured string messages.

Data:

```
{
    message:string
}
```

7.1.4. debug

Importance: Extra

Used mainly for implementations that want to use qlog as their one and only logging format but still want to support unstructured string messages.

Data:

```
{
    message:string
}
```

7.1.5. verbose

Importance: Extra

Used mainly for implementations that want to use qlog as their one and only logging format but still want to support unstructured string messages.

Data:

```
{
    message:string
}
```

7.2. simulation

When evaluating a protocol evaluation, one typically sets up a series of interoperability or benchmarking tests, in which the test situations can change over time. For example, the network bandwidth or latency can vary during the test, or the network can be fully disable for a short time. In these setups, it is useful to know when exactly these conditions are triggered, to allow for proper correlation with other events.

7.2.1. scenario

Importance: Extra

Used to specify which specific scenario is being tested at this particular instance. This could also be reflected in the top-level qlog's "summary" or "configuration" fields, but having a separate event allows easier aggregation of several simulations into one trace.

```
{
    name?:string,
    details?:any
}
```

7.2.2. marker

Importance: Extra

Used to indicate when specific emulation conditions are triggered at set times (e.g., at 3 seconds in 2% packet loss is introduced, at 10s a NAT rebind is triggered).

```
{
    type?:string,
    message?:string
}
```

8. Security Considerations

TBD

9. IANA Considerations

TBD

10. References

10.1. Normative References

[QLOG-MAIN]
Marx, R., Ed., "Main logging schema for qlog", Work in Progress, Internet-Draft, draft-marx-qlog-main-schema-02, 2 November 2020, <https://tools.ietf.org/html/draft-marx-qlog-main-schema-02>.

[QUIC-HTTP]
Bishop, M., Ed., "Hypertext Transfer Protocol Version 3 (HTTP/3)", Work in Progress, Internet-Draft, draft-ietf-quic-http-32, 1 October 2020, <https://tools.ietf.org/html/draft-ietf-quic-http-32>.

[QUIC-QPACK]
Frindell, A., Ed., "QPACK: Header Compression for HTTP/3", Work in Progress, Internet-Draft, draft-ietf-quic-qpack-19, 20 October 2020, <https://tools.ietf.org/html/draft-ietf-quic-qpack-19>.

[QUIC-TRANSPORT]
Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", Work in Progress, Internet-Draft, draft-ietf-quic-transport-32, 1 October 2020, <https://tools.ietf.org/html/draft-ietf-quic-transport-32>.

10.2. Informative 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>.

Appendix A. QUIC data field definitions

A.1. IPAddress

```
class IPAddress : string | bytes;

// an IPAddress can either be a "human readable" form (e.g., "127.0.0.1" for v4 o
r "2001:0db8:85a3:0000:0000:8a2e:0370:7334" for v6) or use a raw byte-form (as th
e string forms can be ambiguous)
```

A.2. PacketType

```
enum PacketType {
    initial,
    handshake,
    zerortt = "0RTT",
    onertt = "1RTT",
    retry,
    version_negotiation,
    stateless_reset,
    unknown
}
```

A.3. PacketNumberSpace

```
enum PacketNumberSpace {
    initial,
    handshake,
    application_data
}
```

A.4. PacketHeader

```
class PacketHeader {
    // Note: short vs long header is implicit through PacketType

    packet_type: PacketType;
    packet_number: uint64;

    flags?: uint8; // the bit flags of the packet headers (spin bit, key update b
it, etc. up to and including the packet number length bits if present) interprete
d as a single 8-bit integer

    token?:Token; // only if packet_type == initial

    length?: uint16, // only if packet_type == initial || handshake || 0RTT. Sign
ifies length of the packet_number plus the payload.

    // only if present in the header
    // if correctly using transport:connection_id_updated events,
    // dcid can be skipped for 1RTT packets
    version?: bytes; // e.g., "ff00001d" for draft-29
    scil?: uint8;
    dcil?: uint8;
    scid?: bytes;
    dcid?: bytes;
}
```

A.5. Token

```
class Token {
    type?:"retry"|"resumption"|"stateless_reset";

    length?:uint32; // byte length of the token
    data?:bytes; // raw byte value of the token

    details?:any; // decoded fields included in the token (typically: peer's IP a
ddress, creation time)
}
```

The token carried in an Initial packet can either be a retry token from a Retry packet, a stateless reset token from a Stateless Reset packet or one originally provided by the server in a NEW_TOKEN frame used when resuming a connection (e.g., for address validation purposes). Retry and resumption tokens typically contain encoded metadata to check the token's validity when it is used, but this metadata and its format is implementation specific. For that, this field includes a general-purpose "details" field.

A.6. KeyType

```
   enum KeyType {
       server_initial_secret,
       client_initial_secret,

       server_handshake_secret,
       client_handshake_secret,

       server_0rtt_secret,
       client_0rtt_secret,

       server_1rtt_secret,
       client_1rtt_secret
   }
```

A.7. QUIC Frames

```
type QuicFrame = PaddingFrame | PingFrame | AckFrame | ResetStreamFrame | StopSen
dingFrame | CryptoFrame | NewTokenFrame | StreamFrame | MaxDataFrame | MaxStreamD
ataFrame | MaxStreamsFrame | DataBlockedFrame | StreamDataBlockedFrame | StreamsB
lockedFrame | NewConnectionIDFrame | RetireConnectionIDFrame | PathChallengeFrame
 | PathResponseFrame | ConnectionCloseFrame | HandshakeDoneFrame | UnknownFrame;
```

A.7.1. PaddingFrame

In QUIC, PADDING frames are simply identified as a single byte of value 0. As such, each padding byte could be theoretically interpreted and logged as an individual PaddingFrame.

However, as this leads to heavy logging overhead, implementations SHOULD instead emit just a single PaddingFrame and set the payload_length property to the amount of PADDING bytes/frames included in the packet.

```
   class PaddingFrame{
       frame_type:string = "padding";

       length?:uint32; // total frame length, including frame header
       payload_length?:uint32;
   }
```

A.7.2. PingFrame

```
   class PingFrame{
       frame_type:string = "ping";

       length?:uint32; // total frame length, including frame header
       payload_length?:uint32;
   }
```

A.7.3. AckFrame

```
class AckFrame{
    frame_type:string = "ack";

    ack_delay?:float; // in ms

    // first number is "from": lowest packet number in interval
    // second number is "to": up to and including // highest packet number in int
erval
    // e.g., looks like [[1,2],[4,5]]
    acked_ranges?:Array<[uint64, uint64]|[uint64]>;

    // ECN (explicit congestion notification) related fields (not always present)
    ect1?:uint64;
    ect0?:uint64;
    ce?:uint64;

    length?:uint32; // total frame length, including frame header
    payload_length?:uint32;
}
```

Note: the packet ranges in AckFrame.acked_ranges do not necessarily have to be ordered (e.g., [[5,9],[1,4]] is a valid value).

Note: the two numbers in the packet range can be the same (e.g., [120,120] means that packet with number 120 was ACKed). However, in that case, implementers SHOULD log [120] instead and tools MUST be able to deal with both notations.

A.7.4. ResetStreamFrame

```
class ResetStreamFrame{
    frame_type:string = "reset_stream";

    stream_id:uint64;
    error_code:ApplicationError | uint32;
    final_size:uint64; // in bytes

    length?:uint32; // total frame length, including frame header
    payload_length?:uint32;
}
```

A.7.5. StopSendingFrame

```
   class StopSendingFrame{
       frame_type:string = "stop_sending";

       stream_id:uint64;
       error_code:ApplicationError | uint32;

       length?:uint32; // total frame length, including frame header
       payload_length?:uint32;
   }
```

A.7.6. CryptoFrame

```
   class CryptoFrame{
       frame_type:string = "crypto";

       offset:uint64;
       length:uint64;

       payload_length?:uint32;
   }
```

A.7.7. NewTokenFrame

```
   class NewTokenFrame{
     frame_type:string = "new_token";

     token:Token
   }
```

A.7.8. StreamFrame

```
class StreamFrame{
    frame_type:string = "stream";

    stream_id:uint64;

    // These two MUST always be set
    // If not present in the Frame type, log their default values
    offset:uint64;
    length:uint64;

    // this MAY be set any time, but MUST only be set if the value is "true"
    // if absent, the value MUST be assumed to be "false"
    fin?:boolean;

    raw?:bytes;
}
```

A.7.9. MaxDataFrame

```
class MaxDataFrame{
  frame_type:string = "max_data";

  maximum:uint64;
}
```

A.7.10. MaxStreamDataFrame

```
class MaxStreamDataFrame{
  frame_type:string = "max_stream_data";

  stream_id:uint64;
  maximum:uint64;
}
```

A.7.11. MaxStreamsFrame

```
class MaxStreamsFrame{
  frame_type:string = "max_streams";

  stream_type:string = "bidirectional" | "unidirectional";
  maximum:uint64;
}
```

A.7.12. DataBlockedFrame

```
class DataBlockedFrame{
  frame_type:string = "data_blocked";

  limit:uint64;
}
```

A.7.13. StreamDataBlockedFrame

```
class StreamDataBlockedFrame{
  frame_type:string = "stream_data_blocked";

  stream_id:uint64;
  limit:uint64;
}
```

A.7.14. StreamsBlockedFrame

```
   class StreamsBlockedFrame{
     frame_type:string = "streams_blocked";

     stream_type:string = "bidirectional" | "unidirectional";
     limit:uint64;
   }
```

A.7.15. NewConnectionIDFrame

```
   class NewConnectionIDFrame{
     frame_type:string = "new_connection_id";

     sequence_number:uint32;
     retire_prior_to:uint32;

     connection_id_length?:uint8;
     connection_id:bytes;

     stateless_reset_token?:Token;
   }
```

A.7.16. RetireConnectionIDFrame

```
   class RetireConnectionIDFrame{
     frame_type:string = "retire_connection_id";

     sequence_number:uint32;
   }
```

A.7.17. PathChallengeFrame

```
   class PathChallengeFrame{
     frame_type:string = "path_challenge";

     data?:bytes; // always 64-bit
   }
```

A.7.18. PathResponseFrame

```
   class PathResponseFrame{
     frame_type:string = "path_response";

     data?:bytes; // always 64-bit
   }
```

A.7.19. ConnectionCloseFrame

raw_error_code is the actual, numerical code. This is useful because some error types are spread out over a range of codes (e.g., QUIC's crypto_error).

```
type ErrorSpace = "transport" | "application";

class ConnectionCloseFrame{
    frame_type:string = "connection_close";

    error_space?:ErrorSpace;
    error_code?:TransportError | ApplicationError | uint32;
    raw_error_code?:uint32;
    reason?:string;

    trigger_frame_type?:uint64 | string; // For known frame types, the appropriat
e "frame_type" string. For unknown frame types, the hex encoded identifier value
}
```

A.7.20. HandshakeDoneFrame

```
class HandshakeDoneFrame{
  frame_type:string = "handshake_done";
}
```

A.7.21. UnknownFrame

```
class UnknownFrame{
    frame_type:string = "unknown";
    raw_frame_type:uint64;

    raw_length?:uint32;
    raw?:bytes;
}
```

A.7.22. TransportError

```
   enum TransportError {
       no_error,
       internal_error,
       connection_refused,
       flow_control_error,
       stream_limit_error,
       stream_state_error,
       final_size_error,
       frame_encoding_error,
       transport_parameter_error,
       connection_id_limit_error,
       protocol_violation,
       invalid_token,
       application_error,
       crypto_buffer_exceeded
   }
```

A.7.23. CryptoError

These errors are defined in the TLS document as "A TLS alert is turned into a QUIC connection error by converting the one-byte alert description into a QUIC error code. The alert description is added to 0x100 to produce a QUIC error code from the range reserved for CRYPTO_ERROR."

This approach maps badly to a pre-defined enum. As such, we define the crypto_error string as having a dynamic component here, which should include the hex-encoded value of the TLS alert description.

```
   enum CryptoError {
       crypto_error_{TLS_ALERT}
   }
```

Appendix B. HTTP/3 data field definitions

B.1. HTTP/3 Frames

```
type HTTP3Frame = DataFrame | HeadersFrame | PriorityFrame | CancelPushFrame | Se
ttingsFrame | PushPromiseFrame | GoAwayFrame | MaxPushIDFrame | DuplicatePushFram
e | ReservedFrame | UnknownFrame;
```

B.1.1. DataFrame

```
   class DataFrame{
       frame_type:string = "data";

       raw?:bytes;
   }
```

B.1.2. HeadersFrame

This represents an _uncompressed_, plaintext HTTP Headers frame (e.g., no QPACK compression is applied).

For example:

```
headers: [{"name":":path","value":"/"},{"name":":method","value":"GET"},{"name":"
:authority","value":"127.0.0.1:4433"},{"name":":scheme","value":"https"}]
```

```
class HeadersFrame{
    frame_type:string = "header";
    headers:Array<HTTPHeader>;
}

class HTTPHeader {
    name:string;
    value:string;
}
```

B.1.3. CancelPushFrame

```
class CancelPushFrame{
    frame_type:string = "cancel_push";
    push_id:uint64;
}
```

B.1.4. SettingsFrame

```
class SettingsFrame{
    frame_type:string = "settings";
    settings:Array<Setting>;
}

class Setting{
    name:string;
    value:string;
}
```

B.1.5. PushPromiseFrame

```
class PushPromiseFrame{
    frame_type:string = "push_promise";
    push_id:uint64;

    headers:Array<HTTPHeader>;
}
```

B.1.6. GoAwayFrame

```
class GoAwayFrame{
    frame_type:string = "goaway";
    stream_id:uint64;
}
```

B.1.7. MaxPushIDFrame

```
class MaxPushIDFrame{
    frame_type:string = "max_push_id";
    push_id:uint64;
}
```

B.1.8. DuplicatePushFrame

```
class DuplicatePushFrame{
    frame_type:string = "duplicate_push";
    push_id:uint64;
}
```

B.1.9. ReservedFrame

```
class ReservedFrame{
    frame_type:string = "reserved";
}
```

B.1.10. UnknownFrame

HTTP/3 re-uses QUIC's UnknownFrame definition, since their values and usage overlaps.

B.2. ApplicationError

```
   enum ApplicationError{
       http_no_error,
       http_general_protocol_error,
       http_internal_error,
       http_stream_creation_error,
       http_closed_critical_stream,
       http_frame_unexpected,
       http_frame_error,
       http_excessive_load,
       http_id_error,
       http_settings_error,
       http_missing_settings,
       http_request_rejected,
       http_request_cancelled,
       http_request_incomplete,
       http_early_response,
       http_connect_error,
       http_version_fallback
   }
```

Appendix C. QPACK DATA type definitions

C.1. QPACK Instructions

   Note: the instructions do not have explicit encoder/decoder types, since there is no overlap between the insturctions of both types in neither name nor function.

```
type QPackInstruction = SetDynamicTableCapacityInstruction | InsertWithNameRefere
nceInstruction | InsertWithoutNameReferenceInstruction | DuplicateInstruction | H
eaderAcknowledgementInstruction | StreamCancellationInstruction | InsertCountIncr
ementInstruction;
```

C.1.1. SetDynamicTableCapacityInstruction

```
   class SetDynamicTableCapacityInstruction {
       instruction_type:string = "set_dynamic_table_capacity";

       capacity:uint32;
   }
```

C.1.2. InsertWithNameReferenceInstruction

```
   class InsertWithNameReferenceInstruction {
       instruction_type:string = "insert_with_name_reference";

       table_type:"static"|"dynamic";

       name_index:uint32;

       huffman_encoded_value:boolean;

       value_length?:uint32;
       value?:string;
   }
```

C.1.3. InsertWithoutNameReferenceInstruction

```
   class InsertWithoutNameReferenceInstruction {
       instruction_type:string = "insert_without_name_reference";

       huffman_encoded_name:boolean;

       name_length?:uint32;
       name?:string;

       huffman_encoded_value:boolean;

       value_length?:uint32;
       value?:string;
   }
```

C.1.4. DuplicateInstruction

```
   class DuplicateInstruction {
       instruction_type:string = "duplicate";

       index:uint32;
   }
```

C.1.5. HeaderAcknowledgementInstruction

```
   class HeaderAcknowledgementInstruction {
       instruction_type:string = "header_acknowledgement";

       stream_id:uint64;
   }
```

C.1.6. StreamCancellationInstruction

```
   class StreamCancellationInstruction {
       instruction_type:string = "stream_cancellation";

       stream_id:uint64;
   }
```

C.1.7. InsertCountIncrementInstruction

```
   class InsertCountIncrementInstruction {
       instruction_type:string = "insert_count_increment";

       increment:uint32;
   }
```

C.2. QPACK Header compression

```
type QPackHeaderBlockRepresentation = IndexedHeaderField | LiteralHeaderFieldWith
Name | LiteralHeaderFieldWithoutName;
```

C.2.1. IndexedHeaderField

Note: also used for "indexed header field with post-base index"

```
class IndexedHeaderField {
    header_field_type:string = "indexed_header";

    table_type:"static"|"dynamic"; // MUST be "dynamic" if is_post_base is true
    index:uint32;

    is_post_base:boolean = false; // to represent the "indexed header field with
post-base index" header field type
}
```

C.2.2. LiteralHeaderFieldWithName

Note: also used for "Literal header field with post-base name reference"

```
class LiteralHeaderFieldWithName {
    header_field_type:string = "literal_with_name";

    preserve_literal:boolean; // the 3rd "N" bit
    table_type:"static"|"dynamic"; // MUST be "dynamic" if is_post_base is true
    name_index:uint32;

    huffman_encoded_value:boolean;
    value_length?:uint32;
    value?:string;

    is_post_base:boolean = false; // to represent the "Literal header field with
post-base name reference" header field type
}
```

C.2.3. LiteralHeaderFieldWithoutName

```
class LiteralHeaderFieldWithoutName {
    header_field_type:string = "literal_without_name";

    preserve_literal:boolean; // the 3rd "N" bit

    huffman_encoded_name:boolean;
    name_length?:uint32;
    name?:string;

    huffman_encoded_value:boolean;
    value_length?:uint32;
    value?:string;
}
```

C.2.4. QPackHeaderBlockPrefix

```
class QPackHeaderBlockPrefix {
    required_insert_count:uint32;
    sign_bit:boolean;
    delta_base:uint32;
}
```

Appendix D. Change Log

D.1. Since draft-01:

Major changes:

* Moved data_moved from http to transport. Also made the "from" and "to" fields flexible strings instead of an enum (#111,#65)

* Moved packet_type fields to PacketHeader. Moved packet_size field out of PacketHeader to RawInfo:length (#40)

* Made events that need to log packet_type and packet_number use a header field instead of logging these fields individually

* Added support for logging retry, stateless reset and initial tokens (#94,#86,#117)

* Moved separate general event categories into a single category "generic" (#47)

* Added "transport:connection_closed" event (#43,#85,#78,#49)

* Added version_information and alpn_information events (#85,#75,#28)

* Added parameters_restored events to help clarify 0-RTT behaviour (#88)

Smaller changes:

* Merged loss_timer events into one loss_timer_updated event

* Field data types are now strongly defined (#10,#39,#36,#115)

* Renamed qpack instruction_received and instruction_sent to instruction_created and instruction_parsed (#114)

* Updated qpack:dynamic_table_updated.update_type. It now has the value "inserted" instead of "added" (#113)

* Updated qpack:dynamic_table_updated. It now has an "owner" field to differentiate encoder vs decoder state (#112)

* Removed push_allowed from http:parameters_set (#110)

* Removed explicit trigger field indications from events, since this was moved to be a generic property of the "data" field (#80)

* Updated transport:connection_id_updated to be more in line with other similar events. Also dropped importance from Core to Base (#45)

* Added length property to PaddingFrame (#34)

* Added packet_number field to transport:frames_processed (#74)

* Added a way to generically log packet header flags (first 8 bits) to PacketHeader

* Added additional guidance on which events to log in which situations (#53)

* Added "simulation:scenario" event to help indicate simulation details

* Added "packets_acked" event (#107)

* Added "datagram_ids" to the datagram_X and packet_X events to allow tracking of coalesced QUIC packets (#91)

* Extended connection_state_updated with more fine-grained states (#49)

D.2. Since draft-00:

* Event and category names are now all lowercase

* Added many new events and their definitions

* "type" fields have been made more specific (especially important for PacketType fields, which are now called packet_type instead of type)

* Events are given an importance indicator (issue #22)

* Event names are more consistent and use past tense (issue #21)

* Triggers have been redefined as properties of the "data" field and updated for most events (issue #23)

Appendix E. Design Variations

TBD

Appendix F. Acknowledgements

Thanks to Marten Seemann, Jana Iyengar, Brian Trammell, Dmitri Tikhonov, Stephen Petrides, Jari Arkko, Marcus Ihlar, Victor Vasiliev, Mirja Kuehlewind, Jeremy Laine, Kazu Yamamoto, Christian Huitema, and Lucas Pardue for their feedback and suggestions.

Author's Address

Robin Marx
Hasselt University

Email: robin.marx@uhasselt.be

QUIC
Internet-Draft
Intended status: Standards Track
Expires: 6 May 2021

R. Marx
Hasselt University

2 November 2020

Main logging schema for qlog
draft-marx-qlog-main-schema-02

Abstract

This document describes a high-level schema for a standardized logging format called qlog. This format allows easy sharing of data and the creation of reusable visualization and debugging tools. The high-level schema in this document is intended to be protocol-agnostic. Separate documents specify how the format should be used for specific protocol data. The schema is also format-agnostic, and can be represented in for example JSON, csv or protobuf.

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 6 May 2021.

Copyright Notice

Table of Contents

1. Introduction

There is currently a lack of an easily usable, standardized endpoint logging format. Especially for the use case of debugging and evaluating modern Web protocols and their performance, it is often difficult to obtain structured logs that provide adequate information for tasks like problem root cause analysis.

This document aims to provide a high-level schema and harness that describes the general layout of an easily usable, shareable, aggregatable and structured logging format. This high-level schema is protocol agnostic, with logging entries for specific protocols and use cases being defined in other documents (see for example [QLOG-QUIC-HTTP3] for QUIC and HTTP/3-related event definitions).

The goal of this high-level schema is to provide amenities and default characteristics that each logging file should contain (or should be able to contain), such that generic and reusable toolsets can be created that can deal with logs from a variety of different protocols and use cases.

As such, this document contains concepts such as versioning, metadata inclusion, log aggregation, event grouping and log file size reduction techniques.

Feedback and discussion welcome at https://github.com/quiclog/internet-drafts (https://github.com/quiclog/internet-drafts). Readers are advised to refer to the "editor's draft" at that URL for an up-to-date version of this document.

1.1. Notational Conventions

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].

While the qlog schema's are format-agnostic, for readability the qlog documents will use a JSON-inspired format ([RFC8259]) for examples and definitions.

As qlog can be serialized both textually but also in binary, we employ a custom datatype definition language, inspired loosely by the "TypeScript" language (https://www.typescriptlang.org/).

This document describes how to employ JSON and NDJSON as textual serializations for qlog in Section 4. Other documents will describe how to utilize other concrete serialization options, though tips and requirements for these are also listed in this document (Section 4).

The main general conventions in this document a reader should be aware of are:

* obj? : this object is optional
* type1 | type2 : a union of these two types (object can be either type1 OR type2)
* obj:type : this object has this concrete type
* obj:array<type> : this object is an array of this type
* class : defines a new type
* // : single-line comment

The main data types are:

* int8 : signed 8-bit integer
* int16 : signed 16-bit integer
* int32 : signed 32-bit integer
* int64 : signed 64-bit integer
* uint8 : unsigned 8-bit integer
* uint16 : unsigned 16-bit integer
* uint32 : unsigned 32-bit integer
* uint64 : unsigned 64-bit integer
* float : 32-bit floating point value

* double : 64-bit floating point value

* byte : an individual raw byte (8-bit) value (use array<byte> or the shorthand "bytes" to specify a binary blob)

* string : list of Unicode (typically UTF-8) encoded characters

* boolean : boolean

* enum: fixed list of values (Unless explicity defined, the value of an enum entry is the string version of its name (e.g., initial = "initial"))

* any : represents any object type. Mainly used here as a placeholder for more concrete types defined in related documents (e.g., specific event types)

All timestamps and time-related values (e.g., offsets) in qlog are logged as doubles in the millisecond resolution.

Other qlog documents can define their own data types (e.g., separately for each Packet type that a protocol supports).

2. Design goals

The main tenets for the qlog schema design are:

* Streamable, event-based logging

* Flexibility in the format, complexity in the tooling (e.g., few components are a MUST, tools need to deal with this)

* Extensible and pragmatic (e.g., no complex fixed schema with extension points)

* Aggregation and transformation friendly (e.g., the top-level element is a container for individual traces, group_id can be used to tag events to a particular context)

* Metadata is stored together with event data

3. The high level qlog schema

A qlog file should be able to contain several indivdual traces and logs from multiple vantage points that are in some way related. To that end, the top-level element in the qlog schema defines only a small set of "header" fields and an array of component traces. For this document, the required "qlog_version" field MUST have a value of "draft-02".

As qlog can be serialized in a variety of ways, the "qlog_format" field is used to indicate which serialization option was chosen. Its value MUST either be one of the options defined in this document (e.g., Section 4) or the field must be omitted entirely, in which case it assumes the default value of "JSON".

In order to make it easier to parse and identify qlog files and their serialization format, the "qlog_version" and "qlog_format" fields and their values SHOULD be in the first 256 characters/bytes of the resulting log file.

An example of the qlog file's top-level structure is shown in Figure 1.

Definition:

```
class QlogFile {
    qlog_version:string,
    qlog_format?:string,
    title?:string,
    description?:string,
    summary?: Summary,
    traces: array<Trace|TraceError>
}
```

JSON serialization:

```
{
    "qlog_version": "draft-02",
    "qlog_format": "JSON",
    "title": "Name of this particular qlog file (short)",
    "description": "Description for this group of traces (long)",
    "summary": {
        ...
    },
    "traces": [...]
}
```

Figure 1: Top-level element

3.1. summary

In a real-life deployment with a large amount of generated logs, it can be useful to sort and filter logs based on some basic summarized or aggregated data (e.g., log length, packet loss rate, log location, presence of error events, ...). The summary field (if present) SHOULD be on top of the qlog file, as this allows for the file to be processed in a streaming fashion (i.e., the implementation could just read up to and including the summary field and then only load the full logs that are deemed interesting by the user).

As the summary field is highly deployment-specific, this document does not specify any default fields or their semantics. Some examples of potential entries are shown in Figure 2.

Definition (purely illustrative example):

```
class Summary {
    "trace_count":uint32, // amount of traces in this file
    "max_duration":uint64, // time duration of the longest trace in ms
    "max_outgoing_loss_rate":float, // highest loss rate for outgoing packets ove
r all traces
    "total_event_count":uint64, // total number of events across all traces,
    "error_count":uint64 // total number of error events in this trace
}
```

JSON serialization:

```
{
    "trace_count": 1,
    "max_duration": 5006,
    "max_outgoing_loss_rate": 0.013,
    "total_event_count": 568,
    "error_count": 2
}
```

Figure 2: Summary example definition

3.2. traces

It is often advantageous to group several related qlog traces together in a single file. For example, we can simultaneously perform logging on the client, on the server and on a single point on their common network path. For analysis, it is useful to aggregate these three individual traces together into a single file, so it can be uniquely stored, transferred and annotated.

As such, the "traces" array contains a list of individual qlog traces. Typical qlogs will only contain a single trace in this array. These can later be combined into a single qlog file by taking the "traces" entry/entries for each qlog file individually and copying them to the "traces" array of a new, aggregated qlog file. This is typically done in a post-processing step.

The "traces" array can thus contain both normal traces (for the definition of the Trace type, see Section 3.3), but also "error" entries. These indicate that we tried to find/convert a file for inclusion in the aggregated qlog, but there was an error during the process. Rather than silently dropping the erroneous file, we can opt to explicitly include it in the qlog file as an entry in the "traces" array, as shown in Figure 3.

Definition:

```
class TraceError {
    error_description: string, // A description of the error
    uri?: string, // the original URI at which we attempted to find the file
    vantage_point?: VantagePoint // see {{vantage_point}}: the vantage point we w
ere expecting to include here
}
```

JSON serialization:

```
{
    "error_description": "File could not be found",
    "uri": "/srv/traces/today/latest.qlog",
    "vantage_point": { type: "server" }
}
```

Figure 3: TraceError definition

Note that another way to combine events of different traces in a single qlog file is through the use of the "group_id" field, discussed in Section 3.4.7.

3.3. Individual Trace containers

The exact conceptual definition of a Trace can be fluid. For example, a trace could contain all events for a single connection, for a single endpoint, for a single measurement interval, for a single protocol, etc. As such, a Trace container contains some metadata in addition to the logged events, see Figure 4.

In the normal use case however, a trace is a log of a single data flow collected at a single location or vantage point. For example, for QUIC, a single trace only contains events for a single logical QUIC connection for either the client or the server.

The semantics and context of the trace can mainly be deduced from the entries in the "common_fields" list and "vantage_point" field.

Definition:

```
class Trace {
    title?: string,
    description?: string,
    configuration?: Configuration,
    common_fields?: CommonFields,
    vantage_point: VantagePoint,
    events: array<Event>
}
```

JSON serialization:

```
{
    "title": "Name of this particular trace (short)",
    "description": "Description for this trace (long)",
    "configuration": {
        "time_offset": 150
    },
    "common_fields": {
        "ODCID": "abcde1234",
        "time_format": "absolute"
    },
    "vantage_point": {
        "name": "backend-67",
        "type": "server"
    },
    "events": [...]
}
```

Figure 4: Trace container definition

3.3.1. configuration

We take into account that a qlog file is usually not used in isolation, but by means of various tools. Especially when aggregating various traces together or preparing traces for a demonstration, one might wish to persist certain tool-based settings inside the qlog file itself. For this, the configuration field is used.

The configuration field can be viewed as a generic metadata field that tools can fill with their own fields, based on per-tool logic. It is best practice for tools to prefix each added field with their tool name to prevent collisions across tools. This document only defines two optional, standard, tool-independent configuration settings: "time_offset" and "original_uris".

Definition:

```
class Configuration {
    time_offset:double, // in ms,
    original_uris: array<string>,

    // list of fields with any type
}
```

JSON serialization:

```
{
    "time_offset": 150, // starts 150ms after the first timestamp indicates
    "original_uris": [
        "https://example.org/trace1.qlog",
        "https://example.org/trace2.qlog"
    ]
}
```

Figure 5: Configuration definition

3.3.1.1. time_offset

The time_offset field indicates by how many milliseconds the starting time of the current trace should be offset. This is useful when comparing logs taken from various systems, where clocks might not be perfectly synchronous. Users could use manual tools or automated logic to align traces in time and the found optimal offsets can be stored in this field for future usage. The default value is 0.

3.3.1.2. original_uris

The original_uris field is used when merging multiple individual qlog files or other source files (e.g., when converting .pcaps to qlog). It allows to keep better track where certain data came from. It is a simple array of strings. It is an array instead of a single string, since a single qlog trace can be made up out of an aggregation of multiple component qlog traces as well. The default value is an empty array.

3.3.1.3. custom fields

Tools can add optional custom metadata to the "configuration" field to store state and make it easier to share specific data viewpoints and view configurations.

Two examples from the qvis toolset (https://qvis.edm.uhasselt.be) are shown in Figure 6.

```
{
    "configuration" : {
        "qvis" : {
            // when loaded into the qvis toolsuite's congestion graph tool
            // zoom in on the period between 1s and 2s and select the 124th event
 defined in this trace
            "congestion_graph": {
                "startX": 1000,
                "endX": 2000,
                "focusOnEventIndex": 124
            }


            // when loaded into the qvis toolsuite's sequence diagram tool
            // automatically scroll down the timeline to the 555th event defined
in this trace
            "sequence_diagram" : {
                "focusOnEventIndex": 555
            }
        }
    }
}
```

Figure 6: Custom configuration fields example

3.3.2. vantage_point

The vantage_point field describes the vantage point from which the trace originates, see Figure 7. Each trace can have only a single vantage_point and thus all events in a trace MUST BE from the perspective of this vantage_point. To include events from multiple vantage_points, implementers can for example include multiple traces, split by vantage_point, in a single qlog file.

Definition:

```
class VantagePoint {
    name?: string,
    type: VantagePointType,
    flow?: VantagePointType
}

class VantagePointType {
    server, // endpoint which initiates the connection.
    client, // endpoint which accepts the connection.
    network, // observer in between client and server.
    unknown
}
```

JSON serialization examples:

```
{
    "name": "aioquic client",
    "type": "client",
}

{
    "name": "wireshark trace",
    "type": "network",
    "flow": "client"
}
```

Figure 7: VantagePoint definition

The flow field is only required if the type is "network" (for example, the trace is generated from a packet capture). It is used to disambiguate events like "packet sent" and "packet received". This is indicated explicitly because for multiple reasons (e.g., privacy) data from which the flow direction can be otherwise inferred (e.g., IP addresses) might not be present in the logs.

Meaning of the different values for the flow field: * "client" indicates that this vantage point follows client data flow semantics (a "packet sent" event goes in the direction of the server). * "server" indicates that this vantage point follow server data flow semantics (a "packet sent" event goes in the direction of the client). * "unknown" indicates that the flow's direction is unknown.

Depending on the context, tools confronted with "unknown" values in the vantage_point can either try to heuristically infer the semantics from protocol-level domain knowledge (e.g., in QUIC, the client always sends the first packet) or give the user the option to switch between client and server perspectives manually.

3.4. Field name semantics

Inside of the "events" field of a qlog trace is a list of events logged by the endpoint. Each event is specified as a generic object with a number of member fields and their associated data. Depending on the protocol and use case, the exact member field names and their formats can differ across implementations. This section lists the main, pre-defined and reserved field names with specific semantics and expected corresponding value formats.

Each qlog event at minimum requires the "time" (Section 3.4.1), "name" (Section 3.4.2) and "data" (Section 3.4.3) fields. Other typical fields are "time_format" (Section 3.4.1), "protocol_type" (Section 3.4.4), "trigger" (Section 3.4.6), and "group_id" Section 3.4.7. As especially these later fields typically have identical values across individual event instances, they are normally logged separately in the "common_fields" (Section 3.4.8).

The specific values for each of these fields and their semantics are defined in separate documents, specific per protocol or use case. For example: event definitions for QUIC and HTTP/3 can be found in [QLOG-QUIC-HTTP3].

Other fields are explicitly allowed by the qlog approach, and tools SHOULD allow for the presence of unknown event fields, but their semantics depend on the context of the log usage (e.g., for QUIC, the ODCID field is used), see [QLOG-QUIC-HTTP3].

An example of a qlog event with its component fields is shown in Figure 8.

```
Definition:

class Event {
    time: double,
    name: string,
    data: any,

    protocol_type?: string,
    group_id?: string|uint32,

    time_format?: "absolute"|"delta"|"relative",

    // list of fields with any type
}

JSON serialization:

{
    time: 1553986553572,

    name: "transport:packet_sent",
    event: "packet_sent",
    data: { ... }

    protocol_type:  "QUIC_HTTP3",
    group_id: "127ecc830d98f9d54a42c4f0842aa87e181a",

    time_format: "absolute",

    ODCID: "127ecc830d98f9d54a42c4f0842aa87e181a", // QUIC specific
}
```

Figure 8: Event fields definition

3.4.1. timestamps

The "time" field indicates the timestamp at which the event occured. Its value is typically the Unix timestamp since the 1970 epoch (number of milliseconds since midnight UTC, January 1, 1970, ignoring leap seconds). However, qlog supports two more succint timestamps formats to allow reducing file size. The employed format is indicated in the "time_format" field, which allows one of three values: "absolute", "delta" or "relative":

* Absolute: Include the full absolute timestamp with each event. This approach uses the largest amount of characters. This is also the default value of the "time_format" field.

* Delta: Delta-encode each time value on the previously logged value. The first event in a trace typically logs the full absolute timestamp. This approach uses the least amount of characters.

* Relative: Specify a full "reference_time" timestamp (typically this is done up-front in "common_fields", see Section 3.4.8) and include only relatively-encoded values based on this reference_time with each event. The "reference_time" value is typically the first absolute timestamp. This approach uses a medium amount of characters.

The first option is good for stateless loggers, the second and third for stateful loggers. The third option is generally preferred, since it produces smaller files while being easier to reason about. An example for each option can be seen in Figure 9.

```
The absolute approach will use:
1500, 1505, 1522, 1588

The delta approach will use:
1500, 5, 17, 66

The relative approach will:
- set the reference_time to 1500 in "common_fields"
- use: 0, 5, 22, 88
```

Figure 9: Three different approaches for logging timestamps

One of these options is typically chosen for the entire trace (put differently: each event has the same value for the "time_format" field). Each event MUST include a timestamp in the "time" field.

Events in each individual trace SHOULD be logged in strictly ascending timestamp order (though not necessarily absolute value, for the "delta" format). Tools CAN sort all events on the timestamp before processing them, though are not required to (as this could impose a significant processing overhead). This can be a problem especially for multi-threaded and/or streaming loggers, who could consider using a separate postprocesser to order qlog events in time if a tool do not provide this feature.

Timestamps do not have to use the UNIX epoch timestamp as their reference. For example for privacy considerations, any initial reference timestamps (for example "endpoint uptime in ms" or "time since connection start in ms") can be chosen. Tools SHOULD NOT assume the ability to derive the absolute Unix timestamp from qlog traces, nor allow on them to relatively order events across two or more separate traces (in this case, clock drift should also be taken into account).

3.4.2. category and event

Events differ mainly in the type of metadata associated with them. To help identify a given event and how to interpret its metadata in the "data" field (see Section 3.4.3), each event has an associated "name" field. This can be considered as a concatenation of two other fields, namely event "category" and event "type".

Category allows a higher-level grouping of events per specific event type. For example for QUIC and HTTP/3, the different categories could be "transport", "http", "qpack", and "recovery". Within these categories, the event Type provides additional granularity. For example for QUIC and HTTP/3, within the "transport" Category, there would be "packet_sent" and "packet_received" events.

Logging category and type separately conceptually allows for fast and high-level filtering based on category and the re-use of event types across categories. However, it also considerably inflates the log size and this flexibility is not used extensively in practice at the time of writing.

As such, the default approach in qlog is to concatenate both field values using the ":" character in the "name" field, as can be seen in Figure 10. As such, qlog category and type names MUST NOT include this character.

```
JSON serialization using separate fields:
{
    category: "transport",
    type: "packet_sent"
}

JSON serialization using ":" concatenated field:
{
    name: "transport:packet_sent"
}
```

Figure 10: Ways of logging category, type and name of an event.

Certain serializations CAN emit category and type as separate fields, and qlog tools SHOULD be able to deal with both the concatenated "name" field, and the separate "category" and "type" fields. Text-based serializations however are encouraged to employ the concatenated "name" field for efficiency.

3.4.3. data

The data field is a generic object. It contains the per-event metadata and its form and semantics are defined per specific sort of event. For example, data field value definitons for QUIC and HTTP/3, see [QLOG-QUIC-HTTP3].

One purely illustrative example for a QUIC "packet_sent" event is shown in Figure 11.

Definition:

```
class TransportPacketSentEvent {
    packet_size?:uint32,
    header:PacketHeader,
    frames?:Array<QuicFrame>
}
```

JSON serialization:

```
{
    packet_size: 1280,
    header: {
        packet_type: "1RTT",
        packet_number: 123
    },
    frames: [
        {
            frame_type: "stream",
            length: 1000,
            offset: 456
        },
        {
            frame_type: "padding"
        }
    ]
}
```

Figure 11: Example of the 'data' field for a QUIC packet_sent event

3.4.4. protocol_type

The "protocol_type" field indicates to which protocol (or protocol "stack") this event belongs. This allows a single qlog file to aggregate traces of different protocols (e.g., a web server offering both TCP+HTTP/2 and QUIC+HTTP/3 connections).

For example, QUIC and HTTP/3 events have the "QUIC_HTTP3" protocol_type value, see [QLOG-QUIC-HTTP3].

Typically however, all events in a single trace are of the same protocol, and this field is logged once in "common_fields", see Section 3.4.8.

3.4.5. custom fields

Note that qlog files can always contain custom fields (e.g., a per-event field indicating its privacy properties or path_id in multipath protocols) and assign custom values to existing fields (e.g., new categories/types for implemenation-specific events). Loggers are free to add such fields and field values and tools MUST either ignore these unknown fields or show them in a generic fashion.

3.4.6. triggers

Sometimes, additional information is needed in the case where a single event can be caused by a variety of other events. In the normal case, the context of the surrounding log messages gives a hint as to which of these other events was the cause. However, in highly-parallel and optimized implementations, corresponding log messages might separated in time. Another option is to explicitly indicate these "triggers" in a high-level way per-event to get more fine-grained information without much additional overhead.

In qlog, the optional "trigger" field contains a string value describing the reason (if any) for this event instance occuring. While this "trigger" field could be a property of the qlog Event itself, it is instead a property of the "data" field instead. This choice was made because many event types do not include a trigger value, and having the field at the Event-level would cause overhead in some serializations. Additional information on the trigger can be added in the form of additional member fields of the "data" field value, yet this is highly implementation-specific, as are the trigger field's string values.

One purely illustrative example of some potential triggers for QUIC's "packet_dropped" event is shown in Figure 12.

Definition:

```
class QuicPacketDroppedEvent {
    packet_type?:PacketType,
    raw_length?:uint32,

    trigger?: "key_unavailable" | "unknown_connection_id" | "decrypt_error" | "un
supported_version"
}
```

Figure 12: Trigger example

3.4.7. group_id

As discussed in Section 3.3, a single qlog file can contain several traces taken from different vantage points. However, a single trace from one endpoint can also contain events from a variety of sources. For example, a server implementation might choose to log events for all incoming connections in a single large (streamed) qlog file. As such, we need a method for splitting up events belonging to separate logical entities.

The simplest way to perform this splitting is by associating a "group identifier" to each event that indicates to which conceptual "group" each event belongs. A post-processing step can then extract events per group. However, this group identifier can be highly protocol and context-specific. In the example above, we might use QUIC's "Original Destination Connection ID" to uniquely identify a connection. As such, they might add a "ODCID" field to each event. However, a middlebox logging IP or TCP traffic might rather use four-tuples to identify connections, and add a "four_tuple" field.

As such, to provide consistency and ease of tooling in cross-protocol and cross-context setups, qlog instead defines the common "group_id" field, which contains a string value. Implementations are free to use their preferred string serialization for this field, so long as it contains a unique value per logical group. Some examples can be seen in Figure 13.

JSON serialization for events grouped by four tuples and QUIC connection IDs:

```
events: [
    {
        time: 1553986553579,
        protocol_type: "TCP_HTTPS2",
        group_id: "ip1=2001:67c:1232:144:9498:6df6:f450:110b,ip2=2001:67c:2b0:1c1
::198,port1=59105,port2=80",
        name: "transport:packet_received",
        data: { ... },
    },
    {
        time: 1553986553581,
        protocol_type: "QUIC_HTTP3",
        group_id: "127ecc830d98f9d54a42c4f0842aa87e181a",
        name: "transport:packet_sent",
        data: { ... },
    }
]
```

Figure 13: Example of group_id usage

Note that in some contexts (for example a Multipath transport protocol) it might make sense to add additional contextual per-event fields (for example "path_id"), rather than use the group_id field for that purpose.

Note also that, typically, a single trace only contains events belonging to a single logical group (for example, an individual QUIC connection). As such, instead of logging the "group_id" field with an identical value for each event instance, this field is typically logged once in "common_fields", see Section 3.4.8.

3.4.8. common_fields

As discussed in the previous sections, information for a typical qlog event varies in three main fields: "time", "name" and associated data. Additionally, there are also several more advanced fields that allow mixing events from different protocols and contexts inside of the same trace (for example "protocol_type" and "group_id"). In most "normal" use cases however, the values of these advanced fields are consistent for each event instance (for example, a single trace contains events for a single QUIC connection).

To reduce file size and making logging easier, qlog uses the "common_fields" list to indicate those fields and their values that are shared by all events in this component trace. This prevents these fields from being logged for each individual event. An example of this is shown in Figure 14.

JSON serialization with repeated field values per-event instance:

```
{
    events: [{
            group_id: "127ecc830d98f9d54a42c4f0842aa87e181a",
            protocol_type: "QUIC_HTTP3",
            time_format: "relative",
            reference_time: "1553986553572",

            time: 2,
            name: "transport:packet_received",
            data: { ... }
        },{
            group_id: "127ecc830d98f9d54a42c4f0842aa87e181a",
            protocol_type: "QUIC_HTTP3",
            time_format: "relative",
            reference_time: "1553986553572",

            time: 7,
            name: "http:frame_parsed",
            data: { ... }
        }
    ]
}
```

JSON serialization with repeated field values extracted to common_fields:

```
{
    common_fields: {
        group_id: "127ecc830d98f9d54a42c4f0842aa87e181a",
        protocol_type:  "QUIC_HTTP3",
        time_format: "relative",
        reference_time: "1553986553572"
    },
    events: [
        {
            time: 2,
            name: "transport:packet_received",
            data: { ... }
        },{
            7,
            name: "http:frame_parsed",
            data: { ... }
        }
    ]
}
```

Figure 14: Example of common_fields usage

The "common_fields" field is a generic dictionary of key-value pairs, where the key is always a string and the value can be of any type, but is typically also a string or number. As such, unknown entries in this dictionary MUST be disregarded by the user and tools (i.e., the presence of an uknown field is explicitly NOT an error).

The list of default qlog fields that are typically logged in common_fields (as opposed to as individual fields per event instance) are:

* time_format
* reference_time
* protocol_type
* group_id

Tools MUST be able to deal with these fields being defined either on each event individually or combined in common_fields. Note that if at least one event in a trace has a different value for a given field, this field MUST NOT be added to common_fields but instead defined on each event individually. Good example of such fields are "time" and "data", who are divergent by nature.

4. Serializing qlog

This document and other related qlog schema definitions are intentionally serialization-format agnostic. This means that implementers themselves can choose how to represent and serialize qlog data practically on disk or on the wire. Some examples of possible formats are JSON, CBOR, CSV, protocol buffers, flatbuffers, etc.

All these formats make certain tradeoffs between flexibility and efficiency, with textual formats like JSON typically being more flexible but also less efficient than binary formats like protocol buffers. The format choice will depend on the practical use case of the qlog user. For example, for use in day to day debugging, a plaintext readable (yet relatively large) format like JSON is probably preferred. However, for use in production, a more optimized yet restricted format can be better. In this latter case, it will be more difficult to achieve interoperability between qlog implementations of various protocol stacks, as some custom or tweaked events from one might not be compatible with the format of the other. This will also reflect in tooling: not all tools will support all formats.

This being said, the authors prefer JSON as the basis for storing qlog, as it retains full flexibility and maximum interoperability. Storage overhead can be managed well in practice by employing compression. For this reason, this document details both how to practically transform qlog schema definitions to JSON and to the streamable NDJSON. We discuss concrete options to bring down JSON size and processing overheads in Section 4.3.

As depending on the employed format different deserializers/parsers should be used, the "qlog_format" field is used to indicate the chosen serialization approach. This field is always a string, but can be made hierarchical by the use of the "." separator between entries. For example, a value of "JSON.optimizationA" can indicate that a default JSON format is being used, but that a certain optimization of type A was applied to the file as well (see also Section 4.3).

4.1. qlog to JSON mapping

When mapping qlog to normal JSON, the "qlog_format" field MUST have the value "JSON". This is also the default qlog serialization and default value of this field.

To facilitate this mapping, the qlog documents employ a format that is close to pure JSON for its examples and data definitions. Still, as JSON is not a typed format, there are some practical peculiarities to observe.

4.1.1. numbers

While JSON has built-in support for integers up to 64 bits in size, not all JSON parsers do. For example, none of the major Web browsers support full 64-bit integers at this time, as all numerical values (both floating-point numbers and integers) are internally represented as floating point IEEE 754 (https://en.wikipedia.org/wiki/Floating-point_arithmetic) values. In practice, this limits their integers to a maximum value of 2^53-1. Integers larger than that are either truncated or produce a JSON parsing error. While this is expected to improve in the future (as "BigInt" support (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/BigInt) has been introduced in most Browsers, though not yet integrated into JSON parsers), we still need to deal with it here.

When transforming an int64, uint64 or double from qlog to JSON, the implementer can thus choose to either log them as JSON numbers (taking the risk of truncation or un-parseability) or to log them as strings instead. Logging as strings should however only be

practically needed if the value is likely to exceed 2^53-1. In practice, even though protocols such as QUIC allow 64-bit values for for example stream identifiers, these high numbers are unlikely to be reached for the overwhelming majority of cases. As such, it is probably a valid trade-off to take the risk and log 64-bit values as JSON numbers instead of strings.

Tools processing JSON-based qlog SHOULD however be able to deal with 64-bit fields being serialized as either strings or numbers.

4.1.2. bytes

Unlike most binary formats, JSON does not allow the logging of raw binary blobs directly. As such, when serializing a byte or array<byte>, a scheme needs to be chosen.

To represent qlog bytes in JSON, they MUST be serialized to their lowercase hexadecimal equivalents (with 0 prefix for values lower than 10). All values are directly appended to each other, without delimiters. The full value is not prefixed with 0x (as is sometimes common). An example is given in Figure 15.

For the five raw unsigned byte input values of: 5 20 40 171 255, the JSON seriali
zation is:

```
{
    raw: "051428abff"
}
```

Figure 15: Example for serializing bytes

As such, the resulting string will always have an even amount of characters and the original byte-size can be retrieved by dividing the string length by 2.

4.1.2.1. Truncated values

In some cases, it can be interesting not to log a full raw blob but instead a truncated value (for example, only the first 100 bytes of an HTTP response body to be able to discern which file it actually contained). In these cases, the original byte-size length cannot be obtained from the serialized value directly. As such, all qlog schema definitions SHOULD include a separate, length-indicating field for all fields of type array<byte> they specify. This allows always retrieving the original length, but also allows the omission of any raw value bytes of the field completely (e.g., out of privacy or security considerations).

To reduce overhead however and in the case the full raw value is logged, the extra length-indicating field can be left out. As such, tools MUST be able to deal with this situation and derive the length of the field from the raw value if no separate length-indicating field is present. All possible permutations are shown by example in Figure 16.

```
// both the full raw value and its length are present (length is redundant)
{
    "raw_length": 5,
    "raw": "051428abff"
}

// only the raw value is present, indicating it represents the fields full value
// the byte length is obtained by calculating raw.length / 2
{
    "raw": "051428abff"
}

// only the length field is present, meaning the value was omitted
{
    "raw_length": 5,
}

// both fields are present and the lengths do not match: the value was truncated
to the first three bytes.
{
    "raw_length": 5,
    "raw": "051428"
}
```

Figure 16: Example for serializing truncated bytes

4.1.3. Summarizing table

By definition, JSON strings are serialized surrounded by quotes. Numbers without.

qlog type	JSON type
int8	number
int16	number
int32	number
uint8	number
uint16	number
uint32	number
float	number
int64	number or string
uint64	number or string
double	number or string
bytes	string (lowercase hex value)
string	string
boolean	string ("true" or "false")
enum	string (full value/name, not index)
any	object ({...})
array	array ([...])

Table 1

4.1.4. Other JSON specifics

JSON files by definition ([RFC8259]) MUST utilize the UTF-8 encoding, both for the file itself and the string values.

Most JSON parsers strictly follow the JSON specification. This includes the rule that trailing comma's are not allowed. As it is frequently annoying to remove these trailing comma's when logging events in a streaming fashion, tool implementers SHOULD allow the last event entry of a qlog trace to be an empty object. This allows loggers to simply close the qlog file by appending "{}]}]}" after their last added event.

Finally, while not specifically required by the JSON specification, all qlog field names in a JSON serialization MUST be lowercase.

4.2. qlog to NDJSON mapping

One of the downsides of using pure JSON is that it is inherently a non-streamable format. Put differently, it is not possible to simply append new qlog events to a log file without "closing" this file at the end by appending "]}]}". Without these closing tags, most JSON parsers will be unable to parse the file entirely. As most platforms do not provide a standard streaming JSON parser (which would be able to deal with this problem), this document also provides a qlog mapping to a streamable JSON format called Newline-Delimited JSON (NDJSON) (http://ndjson.org/).

When mapping qlog to NDJSON, the "qlog_format" field MUST have the value "NDJSON".

NDJSON is very similar to JSON, except that it interprets each line in a file as a fully separate JSON object. Put differently, unlike default JSON, it does not require a file to be wrapped as a full object with "{ ... }" or "[...]". Using this setup, qlog events can simply be appended as individually serialized lines at the back of a streamed logging file.

For this to work, some qlog definitions have to be adjusted however. Mainly, events are no longer part of the "events" array in the Trace object, but are instead logged separately from the qlog "file header" (QlogFile class in Section 3). Additionally, qlog's NDJSON mapping does not allow logging multiple individual traces in a single qlog file. As such, the QlogFile:traces field is replaced by the singular "trace" field, which simply contains the Trace data directly. An example can be seen in Figure 17. Note that the "group_id" field can still be used on a per-event basis to include events from conceptually different sources in a single NDJSON qlog file.

Note as well from Figure 17 that the file's header (QlogFileNDJSON) also needs to be fully serialized on a single line to be NDJSON compatible.

Definition:

```
class QlogFileNDJSON {
    qlog_format: "NDJSON",

    qlog_version:string,
    title?:string,
    description?:string,
    summary?: Summary,
    trace: Trace
}
// list of qlog events, separated by newlines
```

NDJSON serialization:

```
{"qlog_format":"NDJSON","qlog_version":"draft-02","title":"Name of this particula
r NDJSON qlog file (short)","description":"Description for this NDJSON qlog file
(long)","trace":{"common_fields":{"protocol_type":"QUIC_HTTP3","group_id":"127ecc
830d98f9d54a42c4f0842aa87e181a","time_format":"relative","reference_time":"155398
6553572"},"vantage_point":{"name":"backend-67","type":"server"}}}
{"time": 2, "name": "transport:packet_received", "data": { ... } }
{"time": 7, "name": "http:frame_parsed", "data": { ... } }
```

Figure 17: Top-level element

Finally, while not specifically required by the NDJSON specification, all qlog field names in a NDJSON serialization MUST be lowercase.

4.2.1. Supporting NDJSON in tooling

Note that NDJSON is not supported in most default programming environments (unlike normal JSON). However, several custom NDJSON parsing libraries exist (http://ndjson.org/libraries.html) that can be used and the format is easy enough to parse with existing implementations (i.e., by splitting the file into its component lines and feeding them to a normal JSON parser individually, as each line by itself is a valid JSON object).

4.3. Other optimizated formatting options

Both the JSON and NDJSON formatting options described above are serviceable in general small to medium scale (debugging) setups. However, these approaches tend to be relatively verbose, leading to larger file sizes. Additionally, generalized (ND)JSON (de)serialization performance is typically (slightly) lower than that of more optimized and predictable formats. Both aspects make these formats more challenging (though still practical (https://qlog.edm.uhasselt.be/anrw/)) to use in large scale setups.

During the development of qlog, we compared a multitude of alternative formatting and optimization options. The results of this study are summarized on the qlog github repository

(https://github.com/quiclog/internet-drafts/issues/30#issuecomment-617675097). The rest of this section discusses some of these approaches implementations could choose and the expected gains and tradeoffs inherent therein. Tools SHOULD support mainly the compression options listed in Section 4.3.2, as they provide the largest wins for the least cost overall.

Over time, specific qlog formats and encodings can be created that more formally define and combine some of the discussed optimizations or add new ones. We choose to define these schemes in separate documents to keep the main qlog definition clean and generalizable, as not all contexts require the same performance or flexibility as others and qlog is intended to be a broadly usable and extensible format (for example more flexibility is needed in earlier stages of protocol development, while more performance is typically needed in later stages). This is also the main reason why the general qlog format is the less optimized JSON instead of a more performant option.

To be able to easily distinguish between these options in qlog compatible tooling (without the need to have the user provide out-of-band information or to (heuristically) parse and process files in a multitude of ways, see also Section 6), we recommend using explicit file extensions to indicate specific formats. As there are no standards in place for this type of extension to format mapping, we employ a commonly used scheme here. Our approach is to list the applied optimizations in the extension in ascending order of application (e.g., if a qlog file is first optimized with technique A and then compressed with technique B, the resulting file would have the extension ".qlog.A.B"). This allows tooling to start at the back of the extension to "undo" applied optimizations to finally arrive at the expected qlog representation.

4.3.1. Data structure optimizations

The first general category of optimizations is to alter the representation of data within an (ND)JSON qlog file to reduce file size.

The first option is to employ a scheme similar to the CSV (comma separated value [rfc4180]) format, which utilizes the concept of column "headers" to prevent repeating field names for each datapoint instance. Concretely for JSON qlog, several field names are repeated with each event (i.e., time, name, data). These names could be extracted into a separate list, after which qlog events could be serialized as an array of values, as opposed to a full object. This approach was a key part of the original qlog format (prior to draft 02) using the "event_fields" field. However, tests showed that this

optimization only provided a mean file size reduction of 5% (100MB to 95MB) while significantly increasing the implementation complexity, and this approach was abandoned in favor of the default JSON setup. Implementations using this format should not employ a separate file extension (as it still uses JSON), but rather employ a new value of "JSON.namedheaders" (or "NDJSON.namedheaders") for the "qlog_format" field (see Section 3).

The second option is to replace field values and/or names with indices into a (dynamic) lookup table. This is a common compression technique and can provide significant file size reductions (up to 50% in our tests, 100MB to 50MB). However, this approach is even more difficult to implement efficiently and requires either including the (dynamic) table in the resulting file (an approach taken by for example Chromium's NetLog format (https://www.chromium.org/developers/design-documents/network-stack/netlog)) or defining a (static) table up-front and sharing this between implementations. Implementations using this approach should not employ a separate file extension (as it still uses JSON), but rather employ a new value of "JSON.dictionary" (or "NDJSON.dictionary") for the "qlog_format" field (see Section 3).

As both options either proved difficult to implement, reduced qlog file readability, and provided too little improvement compared to other more straightforward options (for example Section 4.3.2), these schemes are not inherently part of qlog.

4.3.2. Compression

The second general category of optimizations is to utilize a (generic) compression scheme for textual data. As qlog in the (ND)JSON format typically contains a large amount of repetition, off-the-shelf (text) compression techniques typically succeed very well in bringing down file sizes (regularly with up to two orders of magnitude in our tests, even for "fast" compression levels). As such, utilizing compression is recommended before attempting other optimization options, even though this might (somewhat) increase processing costs due to the additional compression step.

The first option is to use GZIP compression ([RFC1952]). This generic compression scheme provides multiple compression levels (providing a trade-off between compression speed and size reduction). Utilized at level 6 (a medium setting thought to be applicable for streaming compression of a qlog stream in commodity devices), gzip compresses qlog JSON files to 7% of their initial size on average (100MB to 7MB). For this option, the file extension .qlog.gz SHOULD BE used. The "qlog_format" field should still reflect the original JSON formatting of the qlog data (e.g., "JSON" or "NDJSON").

The second option is to use Brotli compression ([RFC7932]). While similar to gzip, this more recent compression scheme provides a better efficiency. It also allows multiple compression levels. Utilized at level 4 (a medium setting thought to be applicable for streaming compression of a qlog stream in commodity devices), brotli compresses qlog JSON files to 7% of their initial size on average (100MB to 7MB). For this option, the file extension .qlog.br SHOULD BE used. The "qlog_format" field should still reflect the original JSON formatting of the qlog data (e.g., "JSON" or "NDJSON").

Other compression algorithms of course exist (for example xz, zstd, and lz4). We mainly recommend gzip and brotli because of their tweakable behaviour and wide support in web-based environments, which we envision as the main tooling ecosystem (see also Section 6).

4.3.3. Binary formats

The third general category of optimizations is to use a more optimized (often binary) format instead of the textual JSON format. This approach inherently produces smaller files and often has better (de)serialization performance. However, the resultant files are no longer human readable and some formats require hard tradeoffs between flexibility for performance.

The first option is to use the CBOR (Concise Binary Object Representation [rfc7049]) format. For our purposes, CBOR can be viewed as a straighforward binary variant of JSON. As such, existing JSON qlog files can be trivially converted to and from CBOR (though slightly more work is needed for NDJSON qlogs). While CBOR thus does retain the full qlog flexibility, it only provides a 25% file size reduction (100MB to 75MB) compared to textual (ND)JSON. As CBOR support in programming environments is not as widespread as that of textual JSON and the format lacks human readability, CBOR was not chosen as the default qlog format. For this option, the file extension .qlog.cbor SHOULD BE used. The "qlog_format" field should still reflect the original JSON formatting of the qlog data (e.g., "JSON" or "NDJSON").

A second option is to use a more specialized binary format, such as Protocol Buffers (https://developers.google.com/protocol-buffers) (protobuf). This format is battle-tested, has support for optional fields and has libraries in most programming languages. Still, it is significantly less flexible than textual JSON or CBOR, as it relies on a separate, pre-defined schema (a .proto file). As such, it it not possible to (easily) log new event types in protobuf files without adjusting this schema as well, which has its own practical challenges. As qlog is intended to be a flexible, general purpose format, this type of format was not chosen as its basic

serialization. The lower flexibility does lead to significantly reduced file sizes. Our straightforward mapping of the qlog main schema and QUIC/HTTP3 event types to protobuf created qlog files 24% as large as the raw JSON equivalents (100MB to 24MB). For this option, the file extension .qlog.protobuf SHOULD BE used. The "qlog_format" field should reflect the different internal format, for example: "qlog_format": "protobuf".

Note that binary formats can (and should) also be used in conjunction with compression (see Section 4.3.2). For example, CBOR compresses well (to about 6% of the original textual JSON size (100MB to 6MB) for both gzip and brotli) and so does protobuf (5% (gzip) to 3% (brotli)). However, these gains are similar to the ones achieved by simply compression the textual JSON equivalents directly (7%, see Section 4.3.2). As such, since compression is still needed to achieve optimal file size reductions event with binary formats, we feel the more flexible compressed textual JSON options are a better default for the qlog format in general.

4.3.4. Overview and summary

In summary, textual JSON was chosen as the main qlog format due to its high flexibility and because its inefficiencies can be largely solved by the utilization of compression techniques (which are needed to achieve optimal results with other formats as well).

Still, qlog implementers are free to define other qlog formats depending on their needs and context of use. These formats should be described in their own documents, the discussion in this document mainly acting as inspiration and high-level guidance. Implementers are encouraged to add concrete qlog formats and definitions to the designated public repository (https://github.com/quiclog/qlog).

The following table provides an overview of all the discussed qlog formatting options with examples:

format	qlog_format	extension
JSON Section 4.1	JSON	.qlog
NDJSON Section 4.2	NDJSON	.qlog
named headers Section 4.3.1	(ND)JSON.namedheaders	.qlog
dictionary Section 4.3.1	(ND)JSON.dictionary	.qlog
CBOR Section 4.3.3	(ND)JSON	.qlog.cbor
protobuf Section 4.3.3	protobuf	.qlog.protobuf
gzip Section 4.3.2	no change	.gz suffix
brotli Section 4.3.2	no change	.br suffix

Table 2

4.4. Conversion between formats

As discussed in the previous sections, a qlog file can be serialized in a multitude of formats, each of which can conceivably be transformed into or from one another without loss of information. For example, a number of NDJSON streamed qlogs could be combined into a JSON formatted qlog for later processing. Similarly, a captured binary qlog could be transformed to JSON for easier interpretation and sharing.

Secondly, we can also consider other structured logging approaches that contain similar (though typically not identical) data to qlog, like raw packet capture files (for example .pcap files from tcpdump) or endpoint-specific logging formats (for example the NetLog format in Google Chrome). These are sometimes the only options, if an implementation cannot or will not support direct qlog output for any reason, but does provide other internal or external (e.g., SSLKEYLOGFILE export to allow decryption of packet captures) logging options For this second category, a (partial) transformation from/to qlog can also be defined.

As such, when defining a new qlog serialization format or wanting to utilize qlog-compatible tools with existing codebases lacking qlog support, it is recommended to define and provide a concrete mapping from one format to default JSON-serialized qlog. Several of such mappings exist. Firstly, [pcap2qlog]((https://github.com/quiclog/pcap2qlog) transforms QUIC and HTTP/3 packet capture files to qlog. Secondly, netlog2qlog (https://github.com/quiclog/qvis/tree/master/visualizations/src/components/filemanager/netlogconverter) converts chromium's internal dictionary-encoded JSON format to qlog. Finally, quictrace2qlog (https://github.com/quiclog/quictrace2qlog) converts the older quictrace format to JSON qlog. Tools can then easily integrate with these converters (either by incorporating them directly or for example using them as a (web-based) API) so users can provide different file types with ease. For example, the qvis (https://qvis.edm.uhasselt.be) toolsuite supports a multitude of formats and qlog serializations.

5. Methods of access and generation

Different implementations will have different ways of generating and storing qlogs. However, there is still value in defining a few default ways in which to steer this generation and access of the results.

5.1. Set file output destination via an environment variable

To provide users control over where and how qlog files are created, we define two environment variables. The first, QLOGFILE, indicates a full path to where an individual qlog file should be stored. This path MUST include the full file extension. The second, QLOGDIR, sets a general directory path in which qlog files should be placed. This path MUST include the directory separator character at the end.

In general, QLOGDIR should be preferred over QLOGFILE if an endpoint is prone to generate multiple qlog files. This can for example be the case for a QUIC server implementation that logs each QUIC connection in a separate qlog file. An alternative that uses QLOGFILE would be a QUIC server that logs all connections in a single file and uses the "group_id" field (Section 3.4.7) to allow post-hoc separation of events.

Implementations SHOULD provide support for QLOGDIR and MAY provide support for QLOGFILE.

When using QLOGDIR, it is up to the implementation to choose an appropriate naming scheme for the qlog files themselves. The chosen scheme will typically depend on the context or protocols used. For

example, for QUIC, it is recommended to use the Original Destination Connection ID (ODCID), followed by the vantage point type of the logging endpoint. Examples of all options for QUIC are shown in Figure 18.

```
Command: QLOGFILE=/srv/qlogs/client.qlog quicclientbinary

Should result in the the quicclientbinary executable logging a single qlog file n
amed client.qlog in the /srv/qlogs directory.
This is for example useful in tests when the client sets up just a single connect
ion and then exits.

Command: QLOGDIR=/srv/qlogs/ quicserverbinary

Should result in the quicserverbinary executable generating several logs files, o
ne for each QUIC connection.
Given two QUIC connections, with ODCID values "abcde" and "12345" respectively, t
his would result in two files:
/srv/qlogs/abcde_server.qlog
/srv/qlogs/12345_server.qlog

Command: QLOGFILE=/srv/qlogs/server.qlog quicserverbinary

Should result in the the quicserverbinary executable logging a single qlog file n
amed server.qlog in the /srv/qlogs directory.
Given that the server handled two QUIC connections before it was shut down, with
ODCID values "abcde" and "12345" respectively,
this would result in event instances in the qlog file being tagged with the "grou
p_id" field with values "abcde" and "12345".
```

Figure 18: Environment variable examples for a QUIC implementation

5.2. Access logs via a well-known endpoint

After generation, qlog implementers MAY make available generated logs and traces on an endpoint (typically the server) via the following .well-known URI:

```
.well-known/qlog/IDENTIFIER.extension
```

The IDENTIFIER variable depends on the context and the protocol. For example for QUIC, the lowercase Original Destination Connection ID (ODCID) is recommended, as it can uniquely identify a connection. Additionally, the extension depends on the chosen format (see Section 4.3.4). For example, for a QUIC connection with ODCID "abcde", the endpoint for fetching its default JSON-formatted .qlog file would be:

```
.well-known/qlog/abcde.qlog
```

Implementers SHOULD allow users to fetch logs for a given connection on a 2nd, separate connection. This helps prevent pollution of the logs by fetching them over the same connection that one wishes to observe through the log. Ideally, for the QUIC use case, the logs should also be approachable via an HTTP/2 or HTTP/1.1 endpoint (i.e., on TCP port 443), to for example aid debugging in the case where QUIC/UDP is blocked on the network.

qlog implementers SHOULD NOT enable this .well-known endpoint in typical production settings to prevent (malicious) users from downloading logs from other connections. Implementers are advised to disable this endpoint by default and require specific actions from the end users to enable it (and potentially qlog itself). Implementers MUST also take into account the general privacy and security guidelines discussed in Section 7 before exposing qlogs to outside actors.

6. Tooling requirements

Tools ingestion qlog MUST indicate which qlog version(s), qlog format(s), compression methods and potentially other input file formats (for example .pcap) they support. Tools SHOULD at least support .qlog files in the default JSON format (Section 4.1). Additionally, they SHOULD indicate exactly which values for and properties of the name (category and type) and data fields they look for to execute their logic. Tools SHOULD perform a (high-level) check if an input qlog file adheres to the expected qlog schema. If a tool determines a qlog file does not contain enough supported information to correctly execute the tool's logic, it SHOULD generate a clear error message to this effect.

Tools MUST NOT produce breaking errors for any field names and/or values in the qlog format that they do not recognize. Tools SHOULD indicate even unknown event occurences within their context (e.g., marking unknown events on a timeline for manual interpretation by the user).

Tool authors should be aware that, depending on the logging implementation, some events will not always be present in all traces. For example, using a circular logging buffer of a fixed size, it could be that the earliest events (e.g., connection setup events) are later overwritten by "newer" events. Alternatively, some events can be intentionally omitted out of privacy or file size considerations. Tool authors are encouraged to make their tools robust enough to still provide adequate output for incomplete logs.

7. Security and privacy considerations

TODO : discuss privacy and security considerations (e.g., what NOT to log, what to strip out of a log before sharing, ...)

TODO: strip out/don't log IPs, ports, specific CIDs, raw user data, exact times, HTTP HEADERS (or at least :path), SNI values

TODO: see if there is merit in encrypting the logs and having the server choose an encryption key (e.g., sent in transport parameters)

Good initial reference: Christian Huitema's blogpost (https://huitema.wordpress.com/2020/07/21/scrubbing-quic-logs-for-privacy/)

8. IANA Considerations

TODO: primarily the .well-known URI

9. References

9.1. Normative References

[QLOG-QUIC-HTTP3]
Marx, R., Ed., "QUIC and HTTP/3 event definitions for qlog", Work in Progress, Internet-Draft, draft-marx-qlog-event-definitions-quic-h3-02, 2 November 2020, <https://tools.ietf.org/html/draft-marx-qlog-event-definitions-quic-h3-02>.

9.2. Informative References

[RFC1952] Deutsch, P., "GZIP file format specification version 4.3", RFC 1952, DOI 10.17487/RFC1952, May 1996, <https://www.rfc-editor.org/info/rfc1952>.

[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>.

[rfc4180] Shafranovich, Y., "Common Format and MIME Type for Comma-Separated Values (CSV) Files", RFC 4180, DOI 10.17487/RFC4180, October 2005, <https://www.rfc-editor.org/info/rfc4180>.

[rfc7049] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", RFC 7049, DOI 10.17487/RFC7049, October 2013, <https://www.rfc-editor.org/info/rfc7049>.

[RFC7932] Alakuijala, J. and Z. Szabadka, "Brotli Compressed Data Format", RFC 7932, DOI 10.17487/RFC7932, July 2016, <https://www.rfc-editor.org/info/rfc7932>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

Appendix A. Change Log

A.1. Since draft-marx-qlog-main-schema-01:

* Decoupled qlog from the JSON format and described a mapping instead (#89)
  - Data types are now specified in this document and proper definitions for fields were added in this format
  - 64-bit numbers can now be either strings or numbers, with a preference for numbers (#10)
  - binary blobs are now logged as lowercase hex strings (#39, #36)
  - added guidance to add length-specifiers for binary blobs (#102)
* Removed "time_units" from Configuration. All times are now in ms instead (#95)
* Removed the "event_fields" setup for a more straightforward JSON format (#101,#89)
* Added a streaming option using the NDJSON format (#109,#2,#106)
* Described optional optimization options for implementers (#30)
* Added QLOGDIR and QLOGFILE environment variables, clarified the .well-known URL usage (#26,#33,#51)
* Overall tightened up the text and added more examples

A.2. Since draft-marx-qlog-main-schema-00:

* All field names are now lowercase (e.g., category instead of CATEGORY)

* Triggers are now properties on the "data" field value, instead of separate field types (#23)

* group_ids in common_fields is now just also group_id

Appendix B. Design Variations

* Quic-trace (https://github.com/google/quic-trace) takes a slightly different approach based on protocolbuffers.

* Spindump (https://github.com/EricssonResearch/spindump) also defines a custom text-based format for in-network measurements

* Wireshark (https://www.wireshark.org/) also has a QUIC dissector and its results can be transformed into a json output format using tshark.

The idea is that qlog is able to encompass the use cases for both of these alternate designs and that all tooling converges on the qlog standard.

Appendix C. Acknowledgements

Thanks to Jana Iyengar, Brian Trammell, Dmitri Tikhonov, Stephen Petrides, Jari Arkko, Marcus Ihlar, Victor Vasiliev, Mirja Kuehlewind, Jeremy Laine and Lucas Pardue for their feedback and suggestions.

Author's Address

Robin Marx
Hasselt University

Email: robin.marx@uhasselt.be

Network Working Group S. Peng
Internet-Draft Z. Li
Intended status: Informational Huawei Technologies
Expires: August 26, 2021 February 22, 2021

APN Scope and Gap Analysis
draft-peng-apn-scope-gap-analysis-01

Abstract

The APN work in IETF is focused on developing a framework and set of mechanisms to derive, convey and use an identifier to allow for the implementation of fine-grain user (group)-, application (group)-, and service-level requirements at the network layer. APN aims to apply various policies in different nodes along a network path onto a traffic flow altogether, that is, at the headend to steer into corresponding path, at the midpoint to collect corresponding performance measurement data, and at the service function to execute particular policies. Currently there is still no way to realise this composite network service provisioning along the path very efficiently. This document further clarifies the scope of the APN work and describesthe solution gap analysis.

Requirements Language

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 RFC 2119 [RFC2119].

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 August 26, 2021.

Table of Contents

1. Introduction

Application-aware Networking (APN) is introduced in [I-D.li-apn-framework] and [I-D.li-apn-problem-statement-usecases]. APN conveys an identifier along with data packets into network and make the network aware of fine-grain user (group)-, application (group)-, and service-level requirements.

Such an identifier is acquired, constructed in a structured value, and then encapsulated in the packets. Such structured value is treated as an opaque object in the network, to which the network operator applies policies in various nodes/service functions along the path and provide corresponding services. The identifier may represent the traffic of a particular user/application group and/or

service-level requirement but does not directly identify the actual user nor the actual application on the wire.

This structured identifier can be encapsulated in various data plane adopted within a Network Operator controlled limited domain, e.g. MPLS, VXLAN, SR/SRv6 and other tunnel technologies, which waits to be further specified. In an IPv6 network, a design proposal of the structured value can refer to [I-D.li-6man-app-aware-ipv6-network].

With APN, it becomes possible to apply various policies in different nodes along a network path onto a traffic flow altogether in a more efficient way, that is, at the headend to steer into corresponding path, at the midpoint to collect corresponding performance measurement data, and at the service function to execute particular policies. Currently there is still no way to realise this composite network service provisioning along the path very efficiently. It may be possible to stack those various policies in a list of TLVs at the headend. However, this approach would introduce great complexities and impose big challenges on the hardware processing and forwarding.

The example use-case presented in this draft further expands on the rationale for such an identifier and how it can be derived and used in that specific context.

This document further clarifies the scope of the APN work and describes the solution gap analysis.

2. Terminologies

APN: Application-aware Networking

CPE: Customer Premises Equipment

DPI: Deep Packet Inspection

OS: Operating System

3. APN Framework and Scope

The APN framework is introduced in [I-D.li-apn-framework], as shown in the Figure 1.

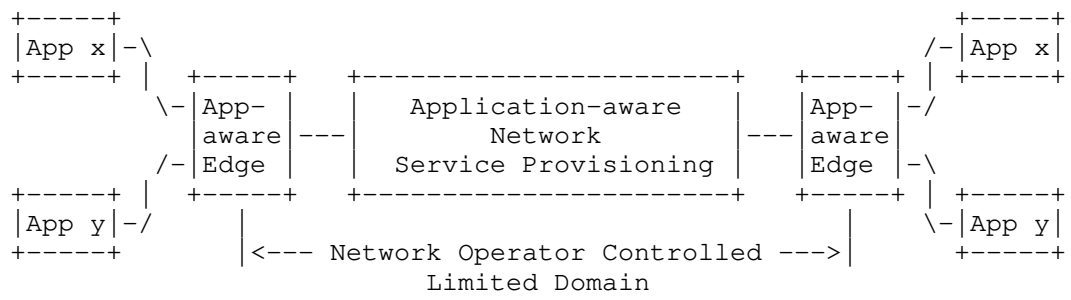


Figure 1. APN Framework and Scope

With APN, the identifier is added to the data packets (e.g. in the IPv6 extensions headers [I-D.li-6man-app-aware-ipv6-network]) and delivered to the network, wherein, according to this identifier, corresponding network services are provisioned.

The identifier can be added either directly by the application (e.g. App x in the Figure 1) or at the network edge devices (i.e. App-aware Edge in the Figure 1), named as host-side solution and network-side solution, respectively.

With the host-side solution, after the identifier is obtained by application, it will be added to the data packets during its encapsulation process going through the protocol stack in the OS. The host-side solution may require an update of the underlying operating system in order to allow the application element to pass the identifier to the socket service when building the packet header.

With the network-side solution, the identifier is added according to the configured policy at the network edge device. For the APN work to be conducted in IETF, we will focus on the network-side solution.

APN works within a limited trusted domain. Typically, an APN domain is defined as a Network Operator controlled limited domain (see Figure 1), in which MPLS, VXLAN, SR/SRv6 and other tunnel technologies are adopted to provide network services.

4. Example Use Case and Existing Issues

To be more specific and more concrete, here we use SD-WAN as an example use case to further expand on the rationale for such identifier and how it can be derived and used in that specific context.

In the case of SD-WAN, an enterprise obtains WAN services from an SD-WAN provider so that its employees have access to the applications in the Cloud, and then the SD-WAN provider may buy WAN lines from a Network Operator. The enterprise may know what applications will use the SD-WAN services, but it will only provide the 5 tuples (i.e. source IP address, source port, destination IP address, destination port, transport protocol) of those applications to the SD-WAN provider. So, the SD-WAN provider does not know what applications it is serving, and will only provide 5 tuples to the Network Operator and the service performance requirements for steering their customer's traffic. In this way, the Network Operator does not know anything else about the traffic except the 5 tuples and requirements. Nowadays, SD-WAN is usually using 5-tuple to steer the traffic into corresponding WAN lines across the Network Operator's network [SD-WAN].

However, there are two main issues in the current SD-WAN deployments.

1) It is complicated to resolve the 5 tuples. Even worse, as the traffic is encrypted, it becomes impossible to obtain any transport layer information. Moreover, in the IPv6 data plane, with the extension headers being added before the upper layer, in some implementations it becomes very difficult and even impossible to obtain transport layer information because that information is so deep in the packet. So, there is no 5 tuples anymore, and maybe only 2 tuples are available.

2) Currently there is still no way to apply various policies in different nodes along the network path onto a traffic flow altogether, that is, at the headend to steer into corresponding path, at the midpoint to collect corresponding performance measurement data, and at the service function to execute particular policies. It may be possible to stack those various policies in a list of TLVs at the headend. However, this approach would introduce great complexities and impose big challenges on the hardware processing and forwarding.

5. Basic Solution and Benefits

With APN, at the edge node, i.e. CPE, of the SD-WAN (see Figure 2), the 5-tuple, plus information related to user or application requirements is constructed into a structured value. Please note, here the structured value is just a bit string and does not indicate actual application or user identification. This information is only meaningful for the network operators to apply various policies in different nodes/service functions, which can be enforced from the Controllers.

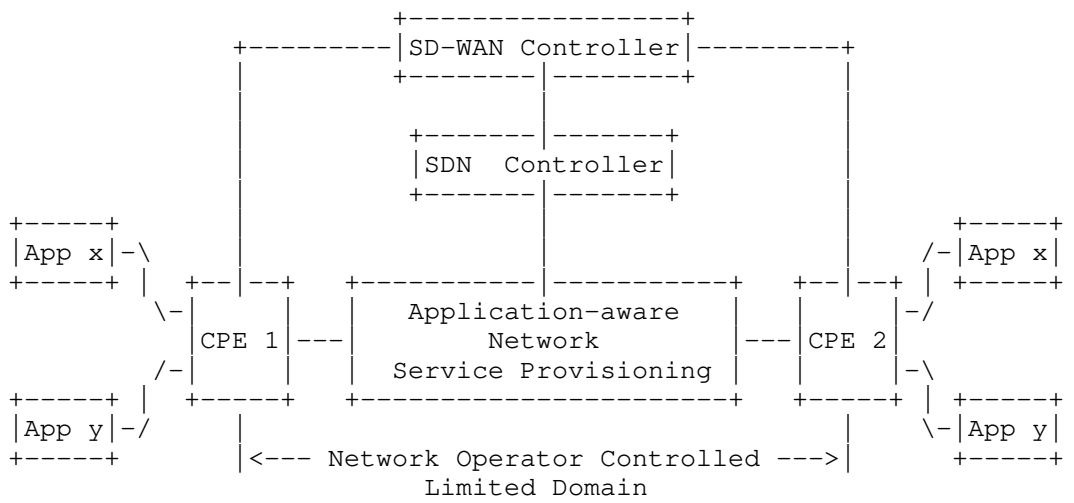


Figure 2. SD-WAN using the APN Framework

With such an identifier in the network, we can easily solve the two issues above-mentioned. It is not necessary to resolve the 5-tuple and perform the deep inspection in every node along the path. This structured value is encapsulated in the IP layer and can be easily read by the routers and service functions. If the data plane is SRv6, for example, such an identifier can be encapsulated in an SRH TLV where it represents the policy corresponding to the application requirements.

Since this identifier is taken as an object to the network, the network operators will simply place the policies in the nodes/service functions where this indicated traffic will go through, and the corresponding node/service function will just apply policies for this object. This can be easily done by utilizing this structured value, which is not possible with any current existing mechanism.

Such structured value will also bring other benefits, for example,

- Improve the forwarding performance since it will only use 1 field in the IP layer instead of resolving 5 tuples, which will also improve the scalability.

- Very flexible policy enforcement in various nodes and service functions along the network path.

Furthermore, with such structured value, more new services could be enabled, for example,

- Even more fine-granularity performance measurement could be achieved and the granularity to be monitored and visualized can be controllable, which is able to relieve the processing pressure on the controller when it is facing the massive monitoring data.

- The policy execution on the service function can be based only on this value and not based on 5-tuple, which can eliminate the need of deep packet inspection.

- The underlay performance guarantee could be achieved for SD-WAN overlay services, such as explicit traffic engineering path satisfying SLA and selective visualized accurate performance measurement.

6. Solution Gap Analysis

There are already some solutions specified in IETF, which use identifier to perform traffic steering and service provisioning. However, none of them is the same as APN and able to achieve the same effects.

6.1. IPv6/MPLS Flow Label

[RFC6437] specifies the IPv6 flow label which enables the IPv6 flow classification. However, the IPv6 flow label is mainly used for Equal Cost Multipath Routing (ECMP) and Link Aggregation [RFC6438].

Similarly, [RFC6391] describes a method of adding an additional Label Stack Entry (LSE) at the bottom of the stack in order to facilitate the load balancing of the flows within a pseudowire (PW) over the available ECMPs. A similar design for general MPLS use has also been proposed in [RFC6790] using the concept of Entropy Label.

6.2. SFC ServiceID

Subscriber Identifier and Performance Policy Identifier are specified in [I-D.ietf-sfc-serviceid-header]. These identifiers are carried only in the Network Service Header (NSH) [RFC8300] Context Header, as shown in Figure 3, while the APN identifier can be carried in various data plane encapsulations.

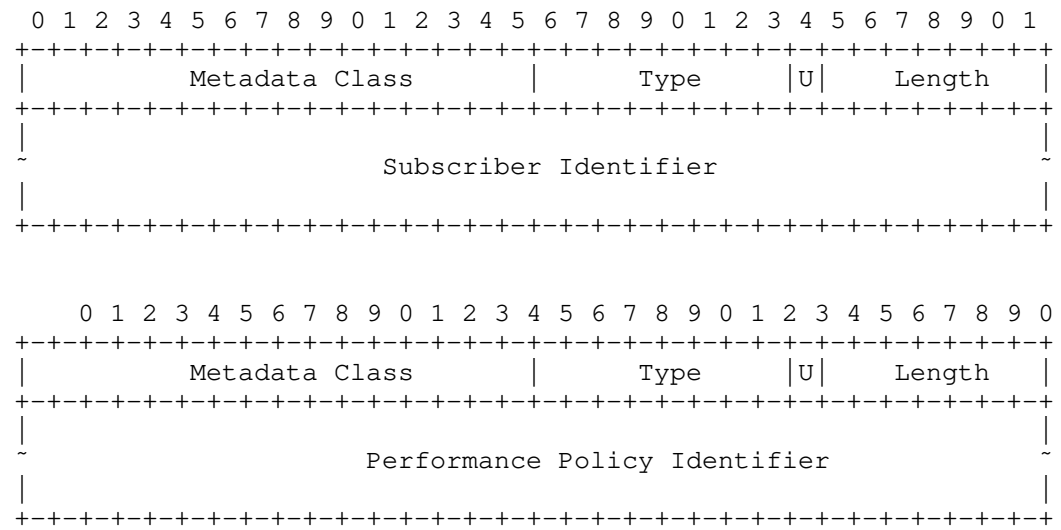


Figure 3. Subscriber Identifier and Performance Policy Identifier

In this draft [I-D.ietf-sfc-serviceid-header], the Subscriber Identifier carries an opaque local identifier that is assigned to a subscriber by a network operator, and the Performance Policy Identifier represents an opaque value pointing to specific performance policy to be enforced. In this way, in order to apply various policies in different nodes along the network path onto a traffic flow altogether, e.g., at the headend to steer into corresponding path, at the midpoint to collect corresponding performance measurement data, and at the service function to execute particular policies, those various policies would have to be stacked in a list of TLVs at the headend, introducing great complexities and big challenges on the hardware processing and forwarding.

The APN identifier, which is a structured value, is treated as an opaque object in the network, to which the network operator applies policies in various nodes/service functions along the path and provide corresponding services. The identifier may represent the application traffic of a particular user but does not identify the actual user nor the actual application for network operators.

6.3. IOAM Flow ID

A 32-bit Flow ID is specified in [I-D.ietf-ippm-ioam-direct-export], which is used to correlate the exported data of the same flow from multiple nodes and from multiple packets, while the APN identifier can serve more various purposes.

6.4. Binding SID

The Binding SID (BSID) [RFC8402] is bound to an SR Policy, instantiation of which may involve a list of SIDs. Any packets received with an active segment equal to BSID are steered onto the bound SR Policy. A BSID may be either a local or a global SID. While the APN identifier is not bound to SRv6 only, and it can be carried in various data plane encapsulations.

6.5. FlowSpec Label

The flow specification (FlowSpec) [RFC5575] is actually an n-tuple consisting of several matching criteria that can be applied to IP traffic, which include elements such as source and destination address prefixes, IP protocol, and transport protocol port numbers. In BGP VPN/MPLS networks, BGP FlowSpec can be extended to identify and change (push/swap/pop) the label(s) for traffic that matches a particular FlowSpec rule in [I-D.ietf-idr-flowspec-mpls-match] and [I-D.ietf-idr-bgp-flowspec-label]. In [I-D.liang-idr-bgp-flowspec-route], BGP is used to distribute the FlowSpec rule bound with label(s). While the APN identifier is not bound to MPLS only, and it can be carried in various data plane encapsulations.

6.6. Summary

As driven by ever-emerging new 5G services, fine-granularity service provisioning becomes urgent. The existing solutions are either specific to a particular scenario or data plane. While APN aims to define a generalized identifier used for fine-granularity service provisioning, and can be carried in various data plane encapsulations.

7. IANA Considerations

There are no IANA considerations in this document.

8. Acknowledgements

The authors would like to acknowledge Martin Vigoureux, Alvaro Retana, Barry Leiba, Stefano Previdi, Adrian Farrel, and Daniel King for their valuable review and comments.

9. Informative References

[I-D.ietf-idr-bgp-flowspec-label]
liangqiandeng, l., Hares, S., You, J., Raszuk, R., and d. danma@cisco.com, "Carrying Label Information for BGP FlowSpec", draft-ietf-idr-bgp-flowspec-label-01 (work in progress), December 2016.

[I-D.ietf-idr-flowspec-mpls-match]
Yong, L., Hares, S., liangqiandeng, l., and J. You, "BGP Flow Specification Filter for MPLS Label", draft-ietf-idr-flowspec-mpls-match-01 (work in progress), December 2016.

[I-D.ietf-ippm-ioam-direct-export]
Song, H., Gafni, B., Zhou, T., Li, Z., Brockners, F., Bhandari, S., Sivakolundu, R., and T. Mizrahi, "In-situ OAM Direct Exporting", draft-ietf-ippm-ioam-direct-export-02 (work in progress), November 2020.

[I-D.ietf-sfc-serviceid-header]
Sarikaya, B., Hugo, D., and M. Boucadair, "Service Function Chaining: Subscriber and Performance Policy Identification Variable-Length Network Service Header (NSH) Context Headers", draft-ietf-sfc-serviceid-header-14 (work in progress), December 2020.

[I-D.li-6man-app-aware-ipv6-network]
Li, Z., Peng, S., Li, C., Xie, C., Voyer, D., Li, X., Liu, P., Liu, C., and K. Ebisawa, "Application-aware IPv6 Networking (APN6) Encapsulation", draft-li-6man-app-aware-ipv6-network-02 (work in progress), July 2020.

[I-D.li-apn-framework]
Li, Z., Peng, S., Voyer, D., Li, C., Geng, L., Cao, C., Ebisawa, K., Previdi, S., and J. Guichard, "Application-aware Networking (APN) Framework", draft-li-apn-framework-01 (work in progress), September 2020.

[I-D.li-apn-problem-statement-usecases]
Li, Z., Peng, S., Voyer, D., Xie, C., Liu, P., Qin, Z., Ebisawa, K., Previdi, S., and J. Guichard, "Problem Statement and Use Cases of Application-aware Networking (APN)", draft-li-apn-problem-statement-usecases-01 (work in progress), September 2020.

[I-D.liang-idr-bgp-flowspec-route]
Liang, Q. and J. You, "BGP FlowSpec based Multi-dimensional Route Distribution", draft-liang-idr-bgp-flowspec-route-00 (work in progress), October 2014.

[I-D.peng-apn-security-privacy-consideration]
Peng, S., Li, Z., Voyer, D., Li, C., Liu, P., and C. Cao, "APN Security and Privacy Considerations", draft-peng-apn-security-privacy-consideration-00 (work in progress), September 2020.

[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>.

[RFC5575] Marques, P., Sheth, N., Raszuk, R., Greene, B., Mauch, J., and D. McPherson, "Dissemination of Flow Specification Rules", RFC 5575, DOI 10.17487/RFC5575, August 2009, <https://www.rfc-editor.org/info/rfc5575>.

[RFC6391] Bryant, S., Ed., Filsfils, C., Drafz, U., Kompella, V., Regan, J., and S. Amante, "Flow-Aware Transport of Pseudowires over an MPLS Packet Switched Network", RFC 6391, DOI 10.17487/RFC6391, November 2011, <https://www.rfc-editor.org/info/rfc6391>.

[RFC6437] Amante, S., Carpenter, B., Jiang, S., and J. Rajahalme, "IPv6 Flow Label Specification", RFC 6437, DOI 10.17487/RFC6437, November 2011, <https://www.rfc-editor.org/info/rfc6437>.

[RFC6438] Carpenter, B. and S. Amante, "Using the IPv6 Flow Label for Equal Cost Multipath Routing and Link Aggregation in Tunnels", RFC 6438, DOI 10.17487/RFC6438, November 2011, <https://www.rfc-editor.org/info/rfc6438>.

[RFC6790] Kompella, K., Drake, J., Amante, S., Henderickx, W., and L. Yong, "The Use of Entropy Labels in MPLS Forwarding", RFC 6790, DOI 10.17487/RFC6790, November 2012, <https://www.rfc-editor.org/info/rfc6790>.

[RFC8402] Filsfils, C., Ed., Previdi, S., Ed., Ginsberg, L., Decraene, B., Litkowski, S., and R. Shakir, "Segment Routing Architecture", RFC 8402, DOI 10.17487/RFC8402, July 2018, <https://www.rfc-editor.org/info/rfc8402>.

[SD-WAN] MEF 70.1 Draft (R1), available at https://www.mef.net/wp-content/uploads/2020/08/MEF-70-1-Draft-R1.pdf/, "SD-WAN Service Attributes and Service Framework", August 2020.

Authors' Addresses

Shuping Peng
Huawei Technologies
Beijing
China

Email: pengshuping@huawei.com

Zhenbin Li
Huawei Technologies
Beijing
China

Email: lizhenbin@huawei.com

Network Working Group
Internet-Draft
Expires: August 25, 2021

J. Rosenberg
Five9
C. Jennings
Cisco
T. Asveren
Ribbon Communications

February 21, 2021

SIP Extensions for High Availability and Load Balancing for Public Cloud
draft-rosenberg-dispatch-cloudsip-00

Abstract

Software making use of the Session Initiation Protocol (SIP) faces challenges in achieving high availability, especially for call stateful applications like softswitches, Session Border Controllers (SBCs), and IP-based call centers applications. The state maintained in the SIP, SDP and SRTP layers changes frequently, and is difficult to replicate. For this reason, commercial systems have often relied on complex active-standby configurations making use of IP address takeover. These solutions are also ill-suited for usage in modern public cloud environments. This document defines a SIP extension facilitating HA, including keeping calls active, which is optimized for server-to-server communication where one or both sides are in public cloud.

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 RFC 2119, BCP 14

[RFC2119] and indicate requirement levels for compliant CoAP implementations.

Software making use of the Session Initiation Protocol (SIP) [RFC3261] faces challenges in achieving high availability, especially for call stateful applications like softswitches, Session Border Controllers (SBCs), and IP-based call centers applications. The state maintained in the SIP, Session Description Protocol (SDP) Offer/Answer [RFC3264] and Secure Real Time Transport Protocol (SRTP) [RFC3711] layers changes frequently, and is difficult to replicate. For this reason, commercial systems have often relied on complex active-standby configurations making use of IP address takeover. These solutions are also ill-suited for usage in modern public cloud environments. SIP assumed server-side components would not maintain call state, and thus it never had built-in mechanisms to facilitate server side HA. In practice, the vast majority of server deployments are B2BUAs and maintain call state.

Besides the challenges in replicating call state, SIP also struggles in achieving HA in modern cloud deployments making use of elastic compute. In these environments, the underlying cloud platform (such as kubernetes), can automatically add and remove instances to a cluster based on usage. Similarly, they will remove elements from the cluster which fail health checks. This information needs to propagate quickly to upstream elements, in order to avoid sending calls to failed or overloaded instances. SIP envisioned that a DNS-based solution using SRV records, [RFC3263] would be sufficient. However, DNS changes are slow to propagate and unpredictable. Commercial implementations have made use of SIP OPTIONS probing to assess liveliness, without standardized behavior. There is also no standardized way to communicate or update the IP addresses used in a cluster of servers.

This specification seeks to remedy these gaps. It defines a simple SIP extension, which is largely a definition of mandatory behaviors for SIP elements, that enable rapid detection and recovery from a failed instance while ensuring that calls do not drop. It also defines a small protocol for retrieving and pushing the set of instances in a cluster so support elastic expansion and contraction of a cluster in a fully automated fashion.

2. Applicability

This extension is focused on server-to-server use cases, where one or both sides are a cluster of servers deployed in a public cloud environment. Examples of these situations include SIP trunks between a PSTN carrier and an enterprise, a PSTN carrier and a VOIP provider (such as a cloud contact center), or between VOIP providers providing

peering. The extension also assumes usage in bilateral peering arrangements, and as such, provides no mechanism for discovery. Rather, it assumes both sides have agreed to use this extension as part of configuration provided through techniques outside the scope of this specification.

3. Requirements

- The solution must enable a call to be recovered in less than 2 seconds. This time represents the amount of time before which a user would hangup because they cannot hear the other party.
- A recovered call means that media continues to flow, and future signaling for features or call hangup, can be performed
- The HA technique must not require servers in the cluster to replicate any SIP/SDP/RTP state beyond the dialog identifiers for calls
- The solution should minimize the changes required to the SIP and RTP protocols and their respective implementations
- The solution must support the case where the telco is using traditional SBCs and is not deploying kubernetes or using public cloud
- The solution must enable fully automated elastic expansion and contraction of clusters
- The solution must support availability, so that when an instance in a cluster fails, new calls are distributed across the remaining N instances
- The solution must support availability, so that when an instance of a cluster fails, all of the active calls that were being handled by that instance are spread across the remaining nodes in the cluster, within 2 seconds
- The solution must support clusters wherein each instance of a cluster has a differing amount of capacity for call handling
- The solution must support the ability for instances of a cluster to gracefully shut down without dropping calls

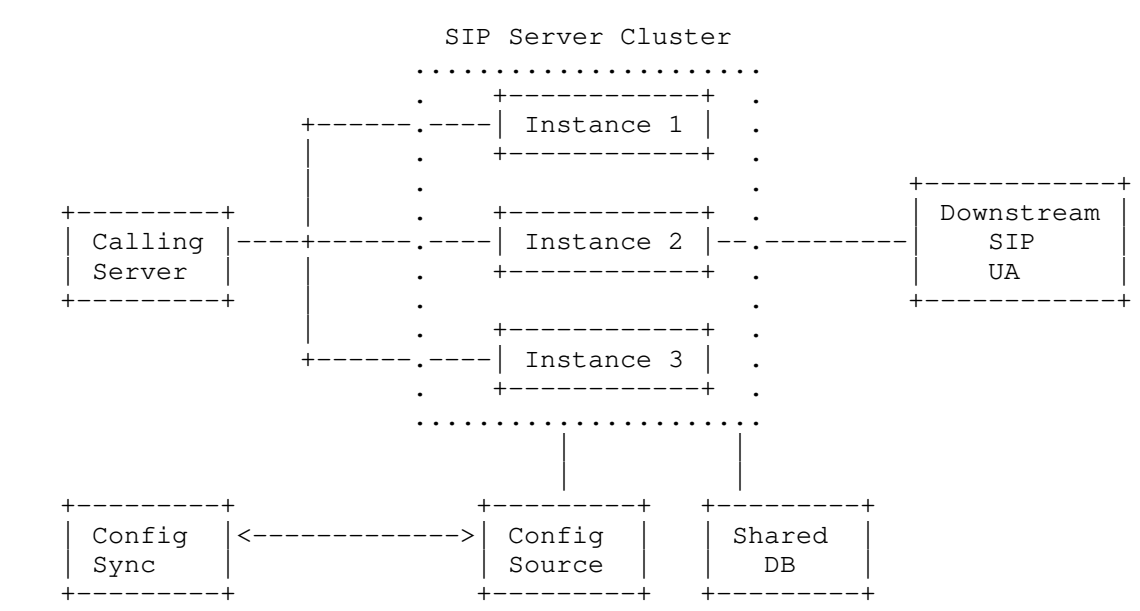
4. Relationship to RIPT

This protocol is similar in goals to RIPT - enabling SIP servers to run in public cloud environments, and achieve HA through techniques employed by web applications. RIPT attempted to solve this problem by utilizing HTTP/3 and fully redefining SIP, repairing many of its problems in the process. This specification is less ambitious, focusing on the minimum changes to SIP required to facilitate HA.

As such, this specification does not alleviate the value in a full-fledged replacement for SIP.

5. Reference Architecture

Cloud SIP uses an assymetric relationship between peers. One side acts as the caller, and the other as the call recipient. New SIP calls can only be placed by the caller, not by the call recipient. If a deployment requires calls to flow in both directions, each side acts as both caller and call recipient.



The calling server wishes to send calls to a cluster, which has a set of instances. The calling server is a B2BUA, and is capable of initiating calls, typically in response to an upstream INVITE it receives. The calling server itself may be a member of a cluster.

When the calling server wishes to generate a new INVITE for a new call, it load balances them amongst the instances in the cluster. Consider a specific call that was sent to instance 2, and was then forwarded through zero or more SIP proxies (not shown) before landing at a UA, referred to here as the downstream SIP UA. The downstream SIP UA may itself be another B2BUA that is a member of a cluster, or even be an end user client. This specification requires the downstream UA to implement the SIP Replaces header field [RFC3891].

When instance 2 fails, we wish to have the call taken over by one of the other instances in the cluster, which can then re-establish media with the downstream SIP UA using INVITE/Replaces.

There is a logical function associated with the cluster, called the config source, which is aware of the configuration of the cluster. Specifically, it knows the IP/port of each instance, and whether that instance is healthy. This config source learns this information through non-standardized means, unique to the cloud environment in which the cluster resides. The config source communicates that information to a config sync associated with the upstream calling server. This communication is bidirectional, using HTTP requests. The config sync distributes this information to the calling server (and any other calling servers should they themselves be a cluster).

There is a shared database of some sorts, accessible by all instances in the cluster. This is used to store the dialog state needed for operation of this extension.

6. Solution Applicability

This specification is applicable in two scenarios:

1. The calling server (and other members in its cluster) and the config sync service are controlled by one entity, and the cluster and the downstream UA are controlled by a second. A common example of this is where the calling server and config sync are part of a telecom carrier, and the cluster and downstream UA are part of an enterprise or SaaS provider that has purchased SIP trunking services from the carrier.

2. The calling server, cluster, and downstream UA are all controlled by a single administrative entity.

3. The instances which make up the cluster are assumed to be provided by the same vendor. This allows for vendor-specific solutions to replicate state and messaging as required by this specification.

This specification also requires that the calling server be a UA (including B2BUAs), and that the instances in the cluster are B2BUAs and the downstream UA is under the administrative control of the same entity that operates the cluster.

7. Overview of Solution

The solution is pretty straightforward.

The calling server will maintain, through the HTTP-based protocol described below, a list of instances in the cluster. These instances are identified by both IP and port. The inclusion of a port allows the instances to share a common IP but vary by port. Such a configuration is useful inside of public cloud environments which can be fronted by a network load balancer which allows each instance to actually have the same IP, but utilize different ports.

The calling server continuously validates that each instance in the cluster is alive, every 250ms. New calls are delivered only to instances which are healthy based on the algorithm defined here. It can ascertain health via reverse RTP traffic, rapid RTCP receiver reports, or via SIP OPTIONS. If SIP OPTIONS are used, these are performed at a rate of a new transaction every 250ms. This is very fast, but it is critical for rapid detection of failures.

If the calling server is itself a member of a cluster, the work of ascertaining the health of each instance can be distributed across the calling servers, in order to avoid a full-mesh of OPTION probing, and then the resulting state distributed through means outside of this specification.

When a call is initially established - to instance 2 in this case - instance 2 will place an entry into the database which contains three pieces of information - (1) the dialogID of the SIP leg from the caller to itself, (2) the dialog ID of the downstream SIP leg from itself to the downstream SIP UA, (3) the IP address and port of the downstream SIP UA.

If an instance transitions from healthy to unhealthy, the calling server 'moves' the existing instance 2 calls uniformly across to the remaining healthy instances in the cluster. To avoid a flood of instant traffic, it moves these calls over a window of at least 200ms and at most one second. To move the calls, the calling server sends an INVITE w. Replaces header field for each such call. Because this is a fresh SIP dialog, a new SDP offer/answer and SRTP is established. This is what avoids the need for replication of RTP state, SDP state and other lower-layer states across the instances in the cluster. When the INVITE/Replaces arrives at one of the

instances in the cluster (say, instance 3), the instance takes the dialogID in the Replaces header field, and looks it up in the shared DB. It will find that there is a matching dialog, and it will retrieve the outbound dialogID and downstream SIP UA. Instance 3 sends an INVITE/Replaces to the downstream UA, using the dialogID it retrieved from the database.

Establishment of a new SIP dialog between the calling server and instance 3 can take place in parallel with the establishment of the new dialog between instance 3 and the downstream UA. Thus the time required to failover the live call is equal to the time to detect instance failure, plus the time to establish a new SIP call.

8. Configuration

A cloud SIP "trunk" is configured in the config sync service through means outside of the scope of this specification. Each such trunk is defined by an HTTPS URI - the trunk config URI - which points to the config source service representing that cluster. This is the only configuration required to establish a cloud SIP trunk.

Once configured with this URI, the config sync MUST perform a GET against this URI. The config source MUST return a JSON document conformant to the schema defined in this specification. This document MUST provide the config sync with a list of instances, each with IP address and port. The JSON document MUST also contain a cluster name, formatted as a hostname, and a webhook registration URI.

Once retrieved, the config sync MUST perform a POST against the webhook registration URI. The POST MUST contain a JSON document conformant to the schema defined in this specification. That document MUST contain an HTTPS webhook URI used by the config sync to receive webhook callbacks that push an updated cluster configuration from the config source.

The config sync MUST refresh its webhook registration at least once a day to ensure that an up to date value for the webhook URI exists.

The config source MUST perform a POST against the webhook URI whenever the cluster configuration changes, including when it detects, on its own, that an instance is unhealthy, removing it from the list.

9. SIP Behavioral Requirements

9.1. Calling Server

9.1.1. Health Probing

The calling server MUST be capable of detecting the failure of any instance of the cluster within 1.5 + RTT seconds. The specific means for doing this detection can vary by implementation. It is also expected that some implementations may have failure detection computed from one instance of a calling server, and the resulting state shared with other instances of the calling server through some means outside the scope of this specification.

One suggested technique for detecting failure is to utilize a SIP OPTIONS probe. The OPTIONS request can be sent every 250ms, directed to each instance of the cluster. To facilitate high scale and determination of RTT, a single OPTIONS request can be sent for each transaction (since retransmits are largely useless due to the short timeout defined for this use case). With such an interval, the calling server can consider an instance unhealthy at time T if, at time T, zero OPTIONS responses have been received for a time equal to the RTT to the instance plus 6 * 250ms = 1.5s. The calling server can maintain the RTT in any fashion it desires. If the OPTIONS requests for a specific transaction are not retransmitted, the time between transmission of the request and receipt of the response can be used to measure RTT.

A MUST strength for 1.5s + RTT is specified to ensure that the cluster can count on consistent and predictable behavior from the upstream calling server. An instance is considered healthy if it is not unhealthy.

OPEN ISSUE: SHould we put a Require header field in the OPTIONS? Should we specify any other behaviors in the OPTIONS?

9.1.2. Utilization Measurement

The instances can place a SIP header, Instance-Utilization, into all responses sent to the calling server. These values indicate the utilization of that instance, as an integral value from 0 to 100. They are used by the calling server to weight the traffic in proportion to utilization.

The calling server MUST remove this header field before propagating it in any upstream responses, as they only have significance on the link between the calling server and cluster.

If this header field is present in a response, the calling server MUST remember the most recent value received from that instance (ordered by the wall clock time at which the response is received). The calling server MUST NOT utilize the source IP of the response to identify the instance. Instead, it MUST correlate the response to a request, and remember the instance to which the request was sent.

If no value has been received for 5 seconds, or no value was ever received, the default value of 50 MUST be used as the utilization.

9.1.3. New Call Initiation

The calling server MUST NOT place a new SIP call to an instance in the cluster which is unhealthy at the time the call is to be placed.

The calling server MUST select an instance for the call using a random function across the instances which are healthy. The calling server MUST weight the probability of selecting that instance in proportion to (100 - the utilization of that instance) . It MUST then direct the call to this instance, by sending the SIP INVITE to the IP address and port of this instance.

As an example, if a cluster has three instances with utilizations at 50, 75 and 100, and all three instances are healthy, no INVITEs are sent to the third instance, 66% are sent to instance one, and 33% are sent to instance 2. Note that, in this case, since instance 3 is not handling new calls, further utilization values can only be learned via responses to OPTION pings, which the calling server MUST send for instances with over 90% utilization.

9.1.4. Instance Failure

When the calling server detects the failure of an instance, it MUST identify all calls which are still active, which were sent to that instance. For each such call, it MUST select a new instance for that call, by choosing one using a uniformly distributed random function amongst the healthy instances. The calling server MUST generate a new INVITE (not a re-INVITE), establishing a new SIP dialog. This INVITE MUST contain a Replaces header field. The Replaces header field MUST contain the dialogID of the call which is being failed over. The INVITE requests MUST be sent uniformly across a 500ms window of time.

9.1.5. Instance to Inactive

If the config sync receives an updated configuration file, and one of the instances from the cluster has been marked as inactive, the calling server MUST NOT send new calls to that instance. However, it

MUST keep existing calls up, and MUST continue to send OPTIONS probes to that instance.

9.1.6. Receiving a REFER

If the calling server receives a REFER request, and the Refer-To URI has a domain portion equal to the IP address of a cluster instance or the FQDN of the cluster, and the Refer-To URI contains an embedded Replaces header field containing a dialogID of a call managed by the calling server, then this REFER is meant to trigger a movement of the call.

The calling server MUST authenticate that this request came from an instance in the cluster. The request is authorized if the domain portion of the Refer-To URI contains an IP address of a cluster instance, or the FQDN of the instance. Furthermore the dialogID in the embedded Refer-To header field matches a dialog that is in progress to that cluster.

If the domain portion of the URI contains an IP address, the calling server MUST perform the requested INVITE/Replaces to that cluster instance. If the domain portion contains the FQDN of the cluster, the calling server MUST send the INVITE/Replacs to one of the other cluster instances, besides the one to which the dialog is currently connected. It MUST select amongst the other instances as if the currently connected instance were inactive, and then round robin using the utilization measures for the remaining instances.

TODO: better explanation, more details

9.2. Cluster Instances

9.2.1. Sending Utilization Values

It is RECOMMENDED that if any one instance of a cluster send values for Cluster-Utilization, all instances do. If none send it, calls will be uniformly balanced across the cluster. Thus, the usage of this header field is only meant for cases where uniform load balancing will not produce uniform utilization.

If an instance is configured to send utilization, it MUST place an Instance-Utilization header field in all responses it sends to all transactions, and include its current measure of utilization. The utilization measure MUST be an integer between 0 and 100 inclusive. Since absolute ordering of responses cannot be guaranteed, the measure SHOULD NOT change more frequently than once a second.

9.2.2. Receiving INVITE w. Replaces

If an instance in the cluster receives an INVITE for a call, and that call has a Replaces header field containing a dialogID for a call that the instance knows is in progress within the cluster, it will know that this is a failover call. It may happen that the failover call is one being handled by the instance receiving the INVITE with Replaces. This is a race condition, but in this case the instance MUST still follow the procedures defined here.

If this is a failover call, the instance MUST authenticate that the INVITE came from the upstream calling server.

There may be cases where the cluster instance receives an INVITE with Replaces header field, but the dialogID does not match a dialog known to the cluster. In such a case, the INVITE MUST be treated as a normal INVITE with a Replaces header field as defined by [RFC3891]. In many cases this may be propagated downstream, or challenged for credentials, neither of which are done if the dialogID is a match for a dialog known to the cluster.

Any downstream SIP dialogs associated with the call MUST be sent an INVITE with Replaces, moving the call to this instance. This will necessarily require the cluster to store the dialogIDs for all dialogs in and out of the cluster, along with any application state needed to reconstruct the dialogs at a new instance.

9.2.3. Graceful Shutdown with Migration

In cases where an instance in the cluster wishes to shut down quickly (perhaps to facilitate a rolling upgrade across the cluster), it can do so by ceasing to respond to OPTIONS requests targeted to itself. The upstream caller will see this as a failure, and move all of the calls off of the instance, onto the remaining instances in the cluster. When the instance reboots, it will begin responding to the OPTIONS probes, enabling it to begin to receive new calls.

9.2.4. Graceful Shutdown without Migration

Another common use case for graceful restart is to cease accepting new calls, but to allow the calls in progress to complete. Once all of the calls have completed, the instance can shut down and restart if desired.

To accomplish this, the cluster config service will mark the instance as inactive in the config file, and pass the updated file to the config sync via webhook. This will cause the calling server to stop

sending new calls to the instance. However, calls in progress will not be dropped.

9.3. Moving a Dialog

Another common case is that an instance is overloaded and wishes to shed a few calls. To facilitate this, a cluster instance MAY send a REFER to the calling server, requesting it to send an INVITE with a Replaces header field. The Refer-To header field embedded in the Refer-To URI MUST contain the dialogID of the call from the calling server to that instance, which is to be moved. To move the call to a specific other instance in the cluster, the domain portion of the URI is set to be equal to the IP address of that instance. Note that the calling server will validate that this IP address is another member of the cluster before authorizing the REFER. Alternatively, the REFER can request the calling server to send the call to any one of the other instances in the cluster, not including itself. To do that, it sets the domain portion of the SIP URI equal to the cluster FQDN.

TODO: Probably need examples and some more details on in or out of dialog REFER

10. Cloud SIP Trunk Configuration File

Something like:

```
{
    "cloud-sip-trunk-name" : "trunk32.acme.com",
    "uri" : "https://configs.sip.acme.com/trunk32",
    "version": 23,
    "webhook-registration" : "https://webhooks.sip.acme.com/trunk32",

    "instances" : [
        {
            "IP" : "1.2.3.4",
            "port" : "5061",
            "status" : "active"
        },
        {
            "IP" : "1.2.3.7",
            "port" : "5061",
            "status" : "inactive"
        }

    ]
}
```

11. Webhook Registration Object

Something like:

```
{
    "webhook" : "https://webhook-receipt.sip.acme.com"
}
```

12. Instance-Utilization Header Field

Something like:

```
{
Instance-Utilization: 34
}
```

IANA registration and formal syntax TBD.

13. Why not DNS

The usage of DNS - and specifically [RFC3263] - might appear to be an alternative to the mechanism in this specification for communicating the IP addresses for the instances of the cluster. However, DNS does not meet the requirements outlined above.

Firstly, DNS is not fast enough to be responsive to the need to add or remove an instance from the cluster. Changes in DNS can take time to propagate. At the time [RFC3263] was conceived, the notion of elastic (and automated) expansion and contraction of clusters did not exist. Cluster instance IPs were extremely static and therefore DNS was sufficient. This is no longer the case.

Secondly, DNS cannot convey state - in particular, information about whether the cluster instances are active or inactive. This is needed to facilitate graceful shutdown of instances. [RFC3263] did not have to concern itself with this problem, because at the time it was believed SIP servers would not contain call state, and therefore, we would not need to worry about this problem.

In addition, because we need to failover extremely quickly - in under two seconds - the calling server needs to perform rapid health probing against all instances in the cluster. This requires the calling server to know all of the IP addresses of the all the instances in the cluster. Typically, DNS queries for an FQDN return one or perhaps a handful of A records, and not every single A record. We expect this specification to be used with clusters that have

instances counts in the hundreds, which is wholly inappropriate to convey via DNS.

14. TODO

Reconcile this with draft-kinamdar-dispatch-sip-audo-peer.

15. Informative 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>.

[RFC3261] Rosenberg, J., Schulzrinne, H., Camarillo, G., Johnston, A., Peterson, J., Sparks, R., Handley, M., and E. Schooler, "SIP: Session Initiation Protocol", RFC 3261, DOI 10.17487/RFC3261, June 2002, <https://www.rfc-editor.org/info/rfc3261>.

[RFC3263] Rosenberg, J. and H. Schulzrinne, "Session Initiation Protocol (SIP): Locating SIP Servers", RFC 3263, DOI 10.17487/RFC3263, June 2002, <https://www.rfc-editor.org/info/rfc3263>.

[RFC3264] Rosenberg, J. and H. Schulzrinne, "An Offer/Answer Model with Session Description Protocol (SDP)", RFC 3264, DOI 10.17487/RFC3264, June 2002, <https://www.rfc-editor.org/info/rfc3264>.

[RFC3711] Baugher, M., McGrew, D., Naslund, M., Carrara, E., and K. Norrman, "The Secure Real-time Transport Protocol (SRTP)", RFC 3711, DOI 10.17487/RFC3711, March 2004, <https://www.rfc-editor.org/info/rfc3711>.

[RFC3891] Mahy, R., Biggs, B., and R. Dean, "The Session Initiation Protocol (SIP) "Replaces" Header", RFC 3891, DOI 10.17487/RFC3891, September 2004, <https://www.rfc-editor.org/info/rfc3891>.

Authors' Addresses

Jonathan Rosenberg
Five9

Email: jdrosen@jdrosen.net

Cullen Jennings
Cisco

Email: fluffy@cisco.com

Tolga Asveren
Ribbon Communications

Email: tasveren@rbbn.com

Calendaring Extensions Working Group
Internet-Draft
Obsoletes: 3339 (if approved)
Intended status: Standards Track
Expires: 26 July 2021

U. Sharma
Igalia, S.L.

22 January 2021

Date and Time on the Internet: Timestamps with additional information
draft-ryzokuken-datetime-extended-01

Abstract

This document defines a date and time format for use in Internet protocols for representation of dates and times using the proleptic Gregorian calendar, with optional extensions representing additional information including a time zone.

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 26 July 2021.

Copyright Notice

Table of Contents

1. Introduction

Date and time formats cause a lot of confusion and interoperability problems on the Internet. This document addresses many of the problems encountered and makes recommendations to improve consistency and interoperability when representing and using date and time in Internet protocols.

This document includes an extension to an Internet profile of the [ISO8601] standard for representation of dates and times using the proleptic Gregorian calendar alongside any additional information.

There are many ways in which date and time values might appear in Internet protocols: this document focuses on just one common usage, viz. timestamps for Internet protocol events. This limited consideration has the following consequences:

* All dates and times are assumed to be in the "current era", somewhere between 0000AD and 9999AD.

* All times expressed have a stated relationship (offset) to Coordinated Universal Time (UTC). Certain applications require the presence of a time zone in order to perform scheduling as well as handle Daylight Savings Time transitions properly. In that case, an optional time zone ID may be included.

* Timestamps can express times that occurred before the introduction of UTC. Such timestamps are expressed relative to universal time, using the best available practice at the stated time.

* Date and time expressions indicate an instant in time. Description of time periods, or intervals, is not covered here.

2. Definitions

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].

UTC Coordinated Universal Time as maintained by the Bureau International des Poids et Mesures (BIPM).

second A basic unit of measurement of time in the International System of Units. It is defined as the duration of 9,192,631,770 cycles of microwave light absorbed or emitted by the hyperfine transition of cesium-133 atoms in their ground state undisturbed by external fields.

minute A period of time of 60 seconds. However, see also the restrictions in section Section 5.9 and Appendix C for how leap seconds are denoted within minutes.

hour A period of time of 60 minutes.

day A period of time of 24 hours.

leap year In the proleptic Gregorian calendar, a year which has 366 days. A leap year is a year whose number is divisible by four an integral number of times, except that if it is a centennial year (i.e. divisible by one hundred) it shall also be divisible by four hundred an integral number of times.

ABNF Augmented Backus-Naur Form, a format used to represent permissible strings in a protocol or language, as defined in [RFC2234].

Email Date/Time Format The date/time format used by Internet Mail as defined by [RFC2822].

Internet Date/Time Format The date/time format defined in section 5 of this document.

Timestamp This term is used in this document to refer to an unambiguous representation of some instant in time.

Z A suffix which, when applied to a time, denotes a UTC offset of 00:00; often spoken "Zulu" from the ICAO phonetic alphabet representation of the letter "Z".

Time Zone A time zone that is a included in the Time Zone Database (often called "tz" or "zoneinfo") maintained by IANA.

For more information about time scales, see Appendix E of [RFC1305], Section 3 of [ISO8601], and the appropriate ITU documents (ITU-R-TF).

3. Two Digit Years

The use of 2 (and 3) digit years was allowed but deprecated in [RFC3339], the predecessor of this document.

The use of such a format is no longer allowed, and implementations should use either a standard 4-digit year or the extended 6-digit value with a sign.

4. Local Time

4.1. Coordinated Universal Time (UTC)

Because the daylight saving rules for local time zones are so convoluted and can change based on local law at unpredictable times, true interoperability is best achieved by using Coordinated Universal Time (UTC). This specification by itself does not cater to local time zone rules. However, certain implementations may be expected to. For these situations, a timestamp may additionally include a local time zone that the implementations can take into account.

4.2. Local Offsets

The offset between local time and UTC is often useful information. For example, in electronic mail (RFC2822, [RFC2822]) the local offset provides a useful heuristic to determine the probability of a prompt response. Attempts to label local offsets with alphabetic strings have resulted in poor interoperability in the past [RFC1123]. As a result, RFC2822 [RFC2822] has made numeric offsets mandatory.

Numeric offsets are calculated as "local time minus UTC". So the equivalent time in UTC can be determined by subtracting the offset from the local time. For example, "18:50:00-04:00" is the same time as "22:50:00Z". (This example shows negative offsets handled by adding the absolute value of the offset.)

Numeric offsets may differ from UTC by any number of seconds, or even a fraction of seconds. This can be easily represented by including an optional seconds value in the offset, which may further optionally include a fraction of seconds behind a decimal point, for example "+12:34:56.789". This is especially useful in the case of certain historical time zones.

4.3. Unknown Local Offset Convention

If the time in UTC is known, but the offset to local time is unknown, this can be represented with an offset of "-00:00". This differs semantically from an offset of "Z" or "+00:00", which imply that UTC is the preferred reference point for the specified time. RFC2822 [RFC2822] describes a similar convention for email.

4.4. Unqualified Local Time

A number of devices currently connected to the Internet run their internal clocks in local time and are unaware of UTC. While the Internet does have a tradition of accepting reality when creating specifications, this should not be done at the expense of interoperability. Since interpretation of an unqualified local time zone will fail in approximately 23/24 of the globe, the interoperability problems of unqualified local time are deemed unacceptable for the Internet. Systems that are configured with a local time, are unaware of the corresponding UTC offset, and depend on time synchronization with other Internet systems, MUST use a mechanism that ensures correct synchronization with UTC. Some suitable mechanisms are:

* Use Network Time Protocol [RFC1305] to obtain the time in UTC.

* Use another host in the same local time zone as a gateway to the Internet. This host MUST correct unqualified local times that are transmitted to other hosts.

* Prompt the user for the local time zone and daylight saving rule settings.

5. Date and Time format

This section discusses desirable qualities of date and time formats and defines a format that extends the profile of ISO 8601 for use in Internet protocols.

5.1. Ordering

If date and time components are ordered from least precise to most precise, then a useful property is achieved. Assuming that the time zones of the dates and times are the same (e.g., all in UTC), expressed using the same string (e.g., all "Z" or all "+00:00"), all times have the same number of fractional second digits, and they all have the same suffix (or none), then the date and time strings may be sorted as strings (e.g., using the "strcmp()" function in C) and a time-ordered sequence will result. The presence of optional punctuation would violate this characteristic.

5.2. Human Readability

Human readability has proved to be a valuable feature of Internet protocols. Human readable protocols greatly reduce the costs of debugging since telnet often suffices as a test client and network analyzers need not be modified with knowledge of the protocol. On the other hand, human readability sometimes results in interoperability problems. For example, the date format "10/11/1996" is completely unsuitable for global interchange because it is interpreted differently in different countries. In addition, the date format in (RFC822) has resulted in interoperability problems when people assumed any text string was permitted and translated the three letter abbreviations to other languages or substituted date formats which were easier to generate (e.g. the format used by the C function "ctime"). For this reason, a balance must be struck between human readability and interoperability.

Because no date and time format is readable according to the conventions of all countries, Internet clients SHOULD be prepared to transform dates into a display format suitable for the locality. This may include translating UTC to local time as well as converting from the Gregorian calendar to the viewer’s preferred calendar.

5.3. Rarely Used Options

A format which includes rarely used options is likely to cause interoperability problems. This is because rarely used options are less likely to be used in alpha or beta testing, so bugs in parsing are less likely to be discovered. Rarely used options should be made mandatory or omitted for the sake of interoperability whenever possible.

5.4. Redundant Information

If a date/time format includes redundant information, that introduces the possibility that the redundant information will not correlate. For example, including the day of the week in a date/time format introduces the possibility that the day of week is incorrect but the date is correct, or vice versa. Since it is not difficult to compute the day of week from a date (see Appendix A), the day of week should not be included in a date/time format.

5.5. Simplicity

The complete set of date and time formats specified in ISO 8601 [ISO8601] is quite complex in an attempt to provide multiple representations and partial representations. Internet protocols have somewhat different requirements and simplicity has proved to be an important characteristic. In addition, Internet protocols usually need complete specification of data in order to achieve true interoperability. Therefore, the complete grammar for ISO 8601 is deemed too complex for most Internet protocols.

The following section defines a format that in an extension of a profile of ISO 8601 for use on the Internet. It is a conformant subset of the ISO 8601 extended format with additional information optionally suffixed. Simplicity is achieved by making most fields and punctuation mandatory.

5.6. Informative

The format should allow implementations to specify additional important information in addition to the bare timestamp. This is done by allowing implementations to include an informative suffix at the end with as many tags as required, each with a hyphen separated key and value. The value can be a hyphen delimited list of multiple values.

In case a key is repeated or conflicted, the implementations should give precedence to whichever value is positioned first.

5.7. Namespaced

Since the suffix can include all sorts of additional information, different standards bodies/organizations need a way to identify which part adheres to their standards. For this, all information needs to be namespaced. Each key is therefore divided into two hyphen-separated sections: the namespace and the key. For example, the calendar as defined by the Unicode consortium could be included as "u-ca-<value>".

All single-character namespaces are reserved for BCP47 extensions recorded in the BCP47 extensions registry. For these namespaces:

* Case differences are ignored.

* The namespace is restricted to single alphanum, corresponding to extension singletons ('x' can be used for a private use extension).

* In addition, for CLDR extensions:

  - There must be a "namespace-key" and it is restricted to 2 "alphanum" characters.

  - A "suffix-value" is limited to "3*8alphanum".

Multi-character namespaces can be registered specifically for use in this format. They are assigned by IANA using the "IETF Review" policy defined by [RFC5226]. This policy requires the development of an RFC, which SHALL define the name, purpose, processes, and procedures for maintaining the subtags. The maintaining or registering authority, including name, contact email, discussion list email, and URL location of the registry, MUST be indicated clearly in the RFC. The RFC MUST specify or include each of the following:

* The specification MUST reference the specific version or revision of this document that governs its creation and MUST reference this section of this document.

* The specification and all keys defined by the specification MUST follow the ABNF and other rules for the formation of keys as defined in this document. In particular, it MUST specify that case is not significant and that keys MUST NOT exceed eight characters in length.

* The specification MUST specify a canonical representation.

* The specification of valid keys MUST be available over the Internet and at no cost.

* The specification MUST be in the public domain or available via a royalty-free license acceptable to the IETF and specified in the RFC.

* The specification MUST be versioned, and each version of the specification MUST be numbered, dated, and stable.

* The specification MUST be stable. That is, namespace keys, once defined by a specification, MUST NOT be retracted or change in meaning in any substantial way.

* The specification MUST include, in a separate section, the registration form reproduced in this section (below) to be used in registering the namespace upon publication as an RFC.

* IANA MUST be informed of changes to the contact information and URL for the specification.

IANA will maintain a registry of allocated multi-character namespaces. This registry MUST use the record-jar format described by the ABNF in [RFC5646]. Upon publication of a namespace as an RFC, the maintaining authority defined in the RFC MUST forward this registration form to <mailto:iesg@ietf.org>, who MUST forward the request to <mailto:iana@iana.org>. The maintaining authority of the namespace MUST maintain the accuracy of the record by sending an updated full copy of the record to <mailto:iana@iana.org> with the subject line "TIMESTAMP FORMAT NAMESPACE UPDATE" whenever content changes. Only the 'Comments', 'Contact_Email', 'Mailing_List', and 'URL' fields MAY be modified in these updates.

Failure to maintain this record, maintain the corresponding registry, or meet other conditions imposed by this section of this document MAY be appealed to the IESG [RFC2028] under the same rules as other IETF decisions (see [RFC2026]) and MAY result in the authority to maintain the extension being withdrawn or reassigned by the IESG.

```
%%
Identifier:
Description:
Comments:
Added:
RFC:
Authority:
Contact_Email:
Mailing_List:
URL:
%%
```

Figure 1: Format of Records in the Timestamp Format Namespace Registry

'Identifier' contains the multi-character sequence assigned to the namespace. The Internet-Draft submitted to define the namespace SHOULD specify which sequence to use, although the IESG MAY change the assignment when approving the RFC.

'Description' contains the name and description of the namespace.

'Comments' is an OPTIONAL field and MAY contain a broader description of the namespace.

'Added' contains the date the namespace's RFC was published in the "date-full" format specified in Figure 2. For example: 2004-06-28 represents June 28, 2004, in the Gregorian calendar.

'RFC' contains the RFC number assigned to the namespace.

'Authority' contains the name of the maintaining authority for the namespace.

'Contact_Email' contains the email address used to contact the maintaining authority.

'Mailing_List' contains the URL or subscription email address of the mailing list used by the maintaining authority.

'URL' contains the URL of the registry for this namespace.

The determination of whether an Internet-Draft meets the above conditions and the decision to grant or withhold such authority rests solely with the IESG and is subject to the normal review and appeals process associated with the RFC process.

5.8. Internet Date/Time Format

The following extension of a profile of [ISO8601] dates SHOULD be used in new protocols on the Internet. This is specified using the syntax description notation defined in [RFC2234].

```
alphanum       = ALPHA / DIGIT

date-year      = 4DIGIT / ("+" / "-") 6DIGIT
date-month     = 2DIGIT ; 01-12
date-mday      = 2DIGIT ; 01-28, 01-29, 01-30, 01-31 based on month/year
date-full      = date-year "-" date-month "-" date-mday

time-hour      = 2DIGIT ; 00-23
time-minute    = 2DIGIT ; 00-59
time-second    = 2DIGIT ; 00-58, 00-59, 00-60 based on leap second rules
time-secfrac   = "." 1*DIGIT
time-partial   = time-hour ":" time-minute ":" time-second [time-secfrac]
time-numoffset = ("+" / "-") time-partial
time-offset    = "Z" / time-numoffset
time-full      = time-partial time-offset

time-zone-char = ALPHA / "." / "_"
time-zone-part = time-zone-char *13(time-zone-char / DIGIT / "-" / "+") ; but
not "." or ".."
time-zone-id   = time-zone-part *("/" time-zone-part)
time-zone      = "[" time-zone-id "]"

namespace      = 1*alphanum
namespace-key  = 1*alphanum
suffix-key     = namespace ["-" namespace-key]

suffix-value   = 1*alphanum
suffix-values  = suffix-value *("-" suffix-value)
suffix-tag     = "[" suffix-key "-" suffix-values "]"
suffix         = [timezone] *suffix-tag

date-time      = date-full "T" time-full suffix
```

Figure 2

> NOTE 1: Per [RFC2234] and ISO8601, the "T" and "Z" characters in this syntax may alternatively be lower case "t" or "z" respectively.

This date/time format may be used in some environments or contexts that distinguish between the upper- and lower-case letters 'A'-'Z' and 'a'-'z' (e.g. XML). Specifications that use this format in such environments MAY further limit the date/time syntax so that the

letters 'T' and 'Z' used in the date/time syntax must always be upper case. Applications that generate this format SHOULD use upper case letters.

> NOTE 2: ISO 8601 defines date and time separated by "T". Applications using this syntax may choose, for the sake of readability, to specify a full-date and full-time separated by (say) a space character.

5.9. Restrictions

The grammar element date-mday represents the day number within the current month. The maximum value varies based on the month and year as follows:

Month Number	Month/Year	Maximum value of date-mday
01	January	31
02	February, normal	28
02	February, leap year	29
03	March	31
04	April	30
05	May	31
06	June	30
07	July	31
08	August	31
09	September	30
10	October	31
11	November	30
12	December	31

Table 1: Days in each month

Appendix B contains sample C code to determine if a year is a leap year.

The grammar element time-second may have the value "60" at the end of months in which a leap second occurs - to date: June (XXXX-06-30T23:59:60Z) or December (XXXX-12-31T23:59:60Z); see Appendix C for a table of leap seconds. It is also possible for a leap second to be subtracted, at which times the maximum value of time-second is "58". At all other times the maximum value of time-second is "59". Further, in time zones other than "Z", the leap second point is shifted by the zone offset (so it happens at the same instant around the globe).

Leap seconds cannot be predicted far into the future. The International Earth Rotation Service publishes bulletins (IERS) that announce leap seconds with a few weeks' warning. Applications should not generate timestamps involving inserted leap seconds until after the leap seconds are announced.

Although ISO 8601 permits the hour to be "24", this extension of a profile of ISO 8601 only allows values between "00" and "23" for the hour in order to reduce confusion.

5.10. Examples

Here are some examples of Internet date/time format.

```
1985-04-12T23:20:50.52Z
```

Figure 3

This represents 20 minutes and 50.52 seconds after the 23rd hour of April 12th, 1985 in UTC.

```
+001985-04-12T23:20:50.52Z
```

Figure 4

This represents the same instant as the previous example but with the expanded 6-digit year format.

```
1996-12-19T16:39:57-08:00
```

Figure 5

This represents 39 minutes and 57 seconds after the 16th hour of December 19th, 1996 with an offset of -08:00 from UTC (Pacific Standard Time). Note that this is equivalent to 1996-12-20T00:39:57Z in UTC.

```
1996-12-19T16:39:57-08:00[America/Los_Angeles]
```

Figure 6

This represents the exact same instant as the previous example but additionally specifies the human time zone associated with it for time zone aware implementations to take into account.

```
1996-12-19T16:39:57-08:00[America/Los_Angeles][u-ca-hebrew]
```

Figure 7

This represents the exact same instant but it informs calendar-aware implementations that they should project it to the Hebrew calendar.

```
1990-12-31T23:59:60Z
```

Figure 8

This represents the leap second inserted at the end of 1990.

```
1990-12-31T15:59:60-08:00
```

Figure 9

This represents the same leap second in Pacific Standard Time, 8 hours behind UTC.

```
1937-01-01T12:00:27.87+00:19:32.130
```

Figure 10

This represents the same instant of time as noon, January 1, 1937, Netherlands time. Standard time in the Netherlands was exactly 19 minutes and 32.13 seconds ahead of UTC by law from 1909-05-01 through 1937-06-30.

```
1937-01-01T12:00:27.87+00:19:32.130[u-ca-japanese]
```

Figure 11

This represents the exact same instant as the previous example but additionally specifies the human calendar associated with it for calendar aware implementations to take into account.

```
1937-01-01T12:00:27.87+00:19:32.130[u-ca-islamic-civil]
```

Figure 12

Since there's not a single agreed upon way to deal with dates in the Islamic calendar, it provides another value to disambiguate between the different interpretations.

```
1937-01-01T12:00:27.87+00:19:32.130[x-foo-bar][x-baz-bat]
```

Figure 13

This timestamp utilizes the private use namespace to declare two additional pieces of information in the suffix that can be interpreted by any compatible implementations and ignored otherwise.

6. Security Considerations

Since the local time zone of a site may be useful for determining a time when systems are less likely to be monitored and might be more susceptible to a security probe, some sites may wish to emit times in UTC only. Others might consider this to be loss of useful functionality at the hands of paranoia.

7. Normative references

[RFC2822] Resnick, P., Ed., "Internet Message Format", IETF RFC 2822, IETF RFC 2822, DOI 10.17487/RFC2822, April 2001, <https://www.rfc-editor.org/info/rfc2822>.

[RFC2234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", IETF RFC 2234, IETF RFC 2234, DOI 10.17487/RFC2234, November 1997, <https://www.rfc-editor.org/info/rfc2234>.

[RFC1123] Braden, R., Ed., "Requirements for Internet HostsApplication and Support", IETF RFC 1123, IETF RFC 1123, DOI 10.17487/RFC1123, October 1989, <https://www.rfc-editor.org/info/rfc1123>.

[RFC1305] Mills, D., "Network Time Protocol (Version 3) Specification, Implementation and Analysis", IETF RFC 1305, IETF RFC 1305, DOI 10.17487/RFC1305, March 1992, <https://www.rfc-editor.org/info/rfc1305>.

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

[RFC5646] Phillips, A., Ed. and M. Davis, Ed., "Tags for Identifying Languages", IETF RFC 5646, IETF RFC 5646, DOI 10.17487/RFC5646, September 2009, <https://www.rfc-editor.org/info/rfc5646>.

[RFC2026] Bradner, S., "The Internet Standards ProcessRevision 3", IETF RFC 2026, IETF RFC 2026, DOI 10.17487/RFC2026, October 1996, <https://www.rfc-editor.org/info/rfc2026>.

[RFC2028] Hovey, R. and S. Bradner, "The Organizations Involved in the IETF Standards Process", IETF RFC 2028, IETF RFC 2028, DOI 10.17487/RFC2028, October 1996, <https://www.rfc-editor.org/info/rfc2028>.

8. Bibliography

[ISO8601] International Organization for Standardization, "Data elements and interchange formats", ISO 8601:1988, June 1988, <https://www.iso.org/standard/15903.html>.

[RFC3339] Klyne, G. and C. Newman, "Date and Time on the Internet: Timestamps", IETF RFC 3339, IETF RFC 3339, DOI 10.17487/RFC3339, July 2002, <https://www.rfc-editor.org/info/rfc3339>.

Appendix A. Day of the Week

The following is a sample C subroutine loosely based on Zeller's Congruence (ZELLER) which may be used to obtain the day of the week for dates on or after 0000-03-01:

```
char *day_of_week(int day, int month, int year)
{
    int cent;
    char *dayofweek[] = {
        "Sunday", "Monday", "Tuesday", "Wednesday",
        "Thursday", "Friday", "Saturday"
    };

    /* adjust months so February is the last one */
    month -= 2;
    if (month < 1) {
        month += 12;
        --year;
    }
    /* split by century */
    cent = year / 100;
    year %= 100;
    return (dayofweek[((26 * month - 2) / 10 + day + year
                    + year / 4 + cent / 4 + 5 * cent) % 7]);
}
```

Figure 14

Appendix B. Leap Years

Here is a sample C subroutine to calculate if a year is a leap year:

```
/* This returns non-zero if year is a leap year.  Must use 4 digit
    year.
*/
int leap_year(int year)
{
    return (year % 4 == 0 && (year % 100 != 0 || year % 400 == 0));
}
```

Figure 15

Appendix C. Leap Seconds

Information about leap seconds can be found at the US Navy Oceanography Portal (https://www.usno.navy.mil/USNO/time/master-clock/leap-seconds). In particular, it notes that:

> The decision to introduce a leap second in UTC is the responsibility of the International Earth Rotation Service (IERS). According to the CCIR Recommendation, first preference is given to the opportunities at the end of December and June, and second preference to those at the end of March and September.

When required, insertion of a leap second occurs as an extra second at the end of a day in UTC, represented by a timestamp of the form YYYY-MM-DDT23:59:60Z. A leap second occurs simultaneously in all time zones, so that time zone relationships are not affected. See section Section 5.10 for some examples of leap second times.

The following table is an excerpt from the table maintained by the United States Naval Observatory. The source data is located at the US Navy Oceanography Portal (ftp://maia.usno.navy.mil/ser7/tai-utc.dat).

This table shows the date of the leap second, and the difference between the time standard TAI (which isn't adjusted by leap seconds) and UTC after that leap second.

UTC Date	TAI - UTC After Leap Second
1972-06-30	11
1972-12-31	12
1973-12-31	13
1974-12-31	14
1975-12-31	15
1976-12-31	16
1977-12-31	17
1978-12-31	18
1979-12-31	19
1981-06-30	20
1982-06-30	21
1983-06-30	22
1985-06-30	23
1987-12-31	24
1989-12-31	25

1990-12-31	26
1992-06-30	27
1993-06-30	28
1994-06-30	29
1995-12-31	30
1997-06-30	31
1998-12-31	32

Table 2: Historic leap seconds

Author's Address

Ujjwal Sharma
Igalia, S.L.

Email: ryzokuken@igalia.com

Internet Engineering Task Force
Internet-Draft
Intended status: Informational
Expires: September 10, 2021

L. Svensson
R. Verborgh
Ghent University - imec
H. Van de Sompel
Data Archiving and Networked Services
March 9, 2021

Indicating, Discovering, Negotiating, and Writing Profiled Representations
draft-svensson-profiled-representations-01

Abstract

This document details approaches for enriching HTTP interactions with information pertaining to the profiles to which resource representations conform. It surveys approaches that were recently introduced to indicate the profile of a resource representation, and to make representations that conform to a profile discoverable. It introduces a generally applicable approach to negotiate for a resource representation that conforms to a profile preferred by a user agent. That approach leverages the existing content negotiation mechanism but applies it to the profile dimension to which it was previously not applied. The document also shows how a server can convey which profiled representations it is able to accept from a user agent.

Table of Contents

1. Introduction

Any given web resource can typically be represented in a variety of ways. For example, the same information could be rendered according to different media types, say, as XML or JSON. But in many cases, variations in representation other than those inherent to a given media type are also possible. For example, the same structured data could be rendered in XML according to different XML Schema [W3C.REC-xmlschema11-1-20120405]. Or the same RDF graph could be expressed on the basis of different vocabularies.

This dimension of variability regarding representations that goes beyond media types has been acknowledged for quite some time. For example, in 2000, the Dublin Core Metadata Initiative (DCMI) introduced "Application Profiles" to explicitly acknowledge that the same metadata can be represented in various ways and defined them as "schemas which consist of data elements drawn from one or more namespaces, combined together by implementers, and optimised for a particular local application" [HeeryAndPatel].

1.1. Terminology

Inspired by the term used in [RFC6906] to refer to this extra dimension of variability, this document uses the term "profile" to mean a description of structural and/or semantic constraints on representations of resources that apply in addition to the constraints inherently indicated by their MIME type:

- Profiles can be dependent on a media type. For example, this is the case for XML, with constraints being expressed using an XML Schema. This is also the case for JSON that offers a wide range of options regarding the use of tree structure, keys, and value types. In these cases, the meaning of the term "profile" intended by this document coincides with the use of the same term in [RFC6906].

- Profiles can be independent of media type. For example, this is the case for RDF graphs that can be rendered according to various media types, while constraints can be expressed in a manner that is independent of media type, among others, using SHACL [W3C.REC-shacl-20170720]. In these cases, the meaning of the term "profile" intended by this document is a slight extension of the one intended by [RFC6906].

1.2. Purpose

When it comes to HTTP interactions, profiles have received little attention despite their de facto existence and the added-value they can bring for building rich applications. Such applications benefit from knowledge regarding the nature of a representation that a client obtains from a server, that a client sends to a server, and that a server is willing to accept from a client, beyond what is conveyed by the representation’s MIME type. These applications are also be helped by an ability to discover representations rendered according to a profile they can handle, or, optimally, an ability to explicitly request a rendering according to a preferred profile.

A common approach to handle profiles is to register them as a media type, dedicated to the combination of an actual media type and a

profile of it. Media types that illustrate this approach include "application/activity+json", "application/calendar+json", and "application/calendar+xml. This approach allows conveying all necessary profile information in HTTP interactions, e.g. using the "Accept" and "Content-Encoding" HTTP headers and the "type" attribute for web links. As such it supports indicating, discovering, and content negotiating (in the media type dimension) for profiled representations. This registration-based approach may be feasible for profiles that are expected to be very widely used but is not practical in case support for many different profiles is required. Also, the "calendar" examples illustrate that the registration-based approach is not ideal when a profile applies to multiple media types. And, the "activity" example illustrates that the approach supports indicating what the major ingredient of a profiled representation is (i.e. the ActivityStreams Vocabulary) but becomes problematic when indications are also needed regarding additional vocabularies used in representations.

Another approach to handle profiles leverages the ability provided by [RFC6838] to register parameters when registering a media type. Some media types have used this capability to register an attribute dedicated to conveying profiles of the media type. For example, for "application/ld+json" the "profile" parameter has been registered for this purpose. The approach provides flexibility for handling many profiles, including ones that are not yet known when registering the media type. It also supports indicating, discovering, and content negotiating (in the media type dimension) for profiled representations using common approaches. But the approach remains problematic because it ties profile information to a media type, depends on registering a parameter to convey profile information when registering a new media type, and, realistically, on the registration of the same parameter name (i.e. "profile" as suggested in [RFC6906]) for all media types for which registrants deem that conveying profile information is important. Additionaly, [RFC6838] discourages registering parameters for previously registered media types, making it highly questionable that a uniform attribute to convey profile information across all media types could retroactively be defined.

Recognizing the importance of profiles and the problems with the aforementioned approaches to handle them, specifications have started to introduce alternative approaches to express information about resource representation profiles in HTTP interactions:

- [RFC6906] introduces the "profile" link relation type that is generally applicable for indicating the profile of a resource representation that is sent by a client to a server or by a server to a client.

- [I-D.nottingham-link-hint] introduces a capability to make profiled representations discoverable via web links by using the "formats" attribute to express the profile of a linked resource.

Section 2 and Section 3 provide a concise overview of the approaches introduced by [RFC6906] and [I-D.nottingham-link-hint], to respectively indicate and discover the profile of resource representations. Section 4 specifies a generally applicable approach to negotiate for representations that conform to a profile preferred by a user agent. Section 5 shows how servers can convey which profiled representations they are able to accept from user agents.

1.3. Notational Conventions

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 RFC 2119 [RFC2119].

This specification uses the terms "link context" and "link target" as defined in [RFC8288]. These terms respectively correspond with "Context IRI" and "Target IRI" as used in [RFC5988]. Although defined as IRIs, in common scenarios they are also URIs.

In the examples provided in this document, links in the HTTP "Link" header are shown on separate lines in order to improve readability. Note, however, that as per Section 3.2 of [RFC7230], line breaks are not allowed in values for HTTP headers; only whitespaces and tabs are supported as seperators.

2. Indicating Profiled Representations

As per [RFC6906], a web link with a "profile" link relation type can be used to indicate the profile of a representation that is exchanged in HTTP interactions. Figure 1 shows a client requesting a representation from a server and Figure 2 shows the server responding. The response includes a "Link" header ([RFC8288]) that contains a link with the "profile" relation type. The link target <http://purl.org/dc/terms/> indicates the profile of the response body (not shown).

Figure 3 shows a client submitting a representation to a server, using the same approach to express the profile to which that representation complies. Section 5 describes how a server can convey the profiles it supports for representations that are submitted by a user agent.

```
GET /some/resource HTTP/1.1
Host: example.org
Connection: close
```

Figure 1: Client requests a representation

```
HTTP/1.1 200 OK
Content-Type: application/xml ; charset=utf-8
Link: <http://purl.org/dc/terms/>; rel="profile"
Content-Length: 23364
Connection: close

...
```

Figure 2: Server indicates the profile of the returned representation

```
PUT /some/resource HTTP/1.1
Host: example.org
Content-Type: application/xml ; charset=utf-8
Link: <http://purl.org/dc/terms/>; rel="profile"
Content-Length: 23364
Connection: close

...
```

Figure 3: Client indicates the profile of a representation submitted to a server

In some cases organisations use separate servers to perform content negotiation and to deliver resources, e. g. when static content is served through content delivery networks. The servers that perform the content negotiation interact with the client requesting the resource and then typically refer to the correct representation using a 303 redirect. In those cases the servers that deliver the representations are not profile aware and thus cannot add the appropriate "Link" headers to the response. Instead the server performing the negotiation will have to supply that information. This is done by adding an "anchor" attribute pointing to the representation the link header refers to. Figure 4 shows the response to a Figure 1 where the server performing the negotiation redirects the client to the resource specified in the "Location" header and by using the same URI in the "anchor" attribute of the "Link" header indicates that the information in the "Link" header does not apply to this response but to the resource redirected to.

```
HTTP/1.1 300 See other
Location: https://static.example.org/other/resource
Link: <http://purl.org/dc/terms/>; rel="profile"
        ; anchor="https://static.example.org/other/resource"

...
```

Figure 4: Server indicates the profile of the representation referred to in the 'Location' header

3. Discovering Profiled Representations

The link hints capability introduced in [I-D.nottingham-link-hint], can be leveraged by a server to make profiled representations discoverable by including a "formats" attribute on web links. Figure 5 shows a response to the client request of Figure 1 in which the server uses this technique. The "Link" header indicates the profile of the returned representation but also points at two alternative representations, each of which conforms to another profile.

```
HTTP/1.1 200 OK
Content-Type: application/xml
Link: <http://purl.org/dc/terms/> ; rel="profile" ,
      </some/other_resource_1>
         ; rel="alternate" ; type="application/xml"
         ; formats="http://example.org/our_internal_xml_profile" ,
      </some/other_resource_2>
         ; rel="alternate"
         ; formats="http://example.org/our_community_profile"
Content-Length: 23364
Connection: close

...
```

Figure 5: Server makes profiled representations discoverable

4. Negotiating for Profiled Representations

Section 1.2 describes two approaches for conveying profile information in HTTP interactions that make it possible to negotiate for profiled representations by applying content negotiation in the media type dimension. It also indicates the restricted applicability of these approaches, in both cases a result of their direct dependence on the media type registration process.

This section describes an approach that applies content negotiation in a dimension that it was previously not applied to. The profile negotiation approach introduced here is generally applicable for resource representations, irrespective of media type. These are its core aspects:

- In order to allow a user agent to inform a server about its preferences regarding profiles for resource representations, the "Accept-Profile" HTTP header introduced here is used. A user agent can specify several profiles and use quality indicators (q-values) to indicate preferences.

- In order to allow a server to express its support for profile negotiation the "Vary" HTTP header is used, in this case, with the "Accept-Profile" value.

- In order to allow a server to convey the profile of a representation delivered to a user agent, rather than introducing a server-side counterpart to the client-side "Accept-Profile" header, the existing "profile" link approach introduced by [RFC6906] is used. If a representation conforms to multiple profiles, a distinct "profile" link is used per profile; the order in which these links are provided has no relevance.

- In order to allow a server to convey the profiles it supports, web links with the "formats" link hint introduced in [I-D.nottingham-link-hint] are used to convey the profiles of the link target resources. This approach uniformely applies to reponses to HTTP HEAD/GET/PUT/POST/PATCH.

- Throughout the profile negotiation approach, a profile MUST be referred to by a URI. This is the case for the content of the "Accept-Profile" HTTP header, for the target of web links with the "profile" link relation, and for the content of the "formats" link hint for web links.

4.1. Profile Negotiation Details

Profile negotiation uses the content negotiation processes described in Section 3.4 of RFC 7231 [RFC7231] but applies them to the profile dimension. Both proactive negotiation and reactive negotiation for profiles can be supported by servers. Both are described in more detail in the remainder of this section.

In profile negotiation, a profile MUST be referred to by a URI that, from here onwards, is named a profile URI. If the profile URI is dereferencable it SHOULD lead to a document that details the profile. If the profile URI is not dereferencable (e.g. a URN [RFC8141] or an

info-URI [RFC4452]) facilities SHOULD be available to allow user agents and servers to understand their meaning, e.g. community registries of profiles.

4.1.1. Proactive Profile Negotiation

In proactive profile negotiation, the user agent uses the "Accept-Profile" HTTP header to inform the server about the agent's preference regarding profiles to be used for representing a resource in the server's response. In case a user agent wants to express a preference for a single profile, the value of the header is that profile's URI. In case a user agent wants to express a preference for multiple profiles, the value of the header is a list containing each profile's URI, seperated by commas. Alternatively, multiple "Accept-Profile" HTTP headers can be used, each conveying a single profile URI. Quality indicators (q-values) MAY be used to rank profile preferences. The order in which profile URIs are conveyed or the duplicate mentioning of a same profile URI MUST NOT be interpreted as significant.

A server that supports proactive profile negotiation for the resource that a user agent interacts with:

- MUST include a "Vary" HTTP header containing the value "accept-profile" in its response to the user agent.
- MUST include a "Link" HTTP header containing a link with the "profile" relation type that has as link target the profile URI of the resource representation returned to the user agent. In case the representation conforms to additional profiles known to the server, such a "profile" link SHOULD be included for each.
- SHOULD convey the availability of alternate profiled representations of the resource by using the link hint approach described in Section 4.1.2.

These requirements for servers that support proactive profile negotiation also apply when:

- The user agent expressed a profile preference in its request by using an "Accept-Profile" header but the server cannot return a representation that conforms to a preferred profile.
- The user agent did not express a profile preference using an "Accept-Profile" header in its request.

A user agent SHOULD interpret the abscence of a "Vary" HTTP header with an "accept-profile" value in a response from a server as the

lack of support for profile negotiation for the resource the user agent interacts with.

A server SHOULD consider representations that do not conform to any of the profiles listed by a user agent in an "Accept-Profile" header as non-interpretable by the agent. As such, honoring the user agent preferences in the profile dimension SHOULD take precedence over honoring content negotiation in other dimensions.

In Figure 6 a user agent requests an RDF serialization from a server and expresses preference for two media types using the "Accept" header and two profiles using the "Accept-Profile" header. It uses q-values to express a preference for the profile with profile URI <http://example.org/shapes/shape-1> over the one with profile URI <http://example.org/shapes/shape-2>.

```
GET /document HTTP/1.1
Host: example.org
Accept: text/turtle, application/rdf+xml
Accept-Profile: "http://example.org/shapes/shape-1" ; q=0.8 ,
                "http://example.org/shapes/shape-2" ; q=0.5
Connection: close
```

Figure 6: Client expresses a preference for two profiles

Figure 7 shows the server's response to the request of Figure 6. By means of the "Vary" header, the server expresses support for negotiation in both the media type and profile dimensions. The "profile" link with link target <http://example.org/shapes/shape-2> indicates that the server was able to honor the user agent's second profile preference. Another "profile" link shows that the delivered representation also conforms to a profile with profile URI <http://example.org/shapes/shape-3>. Furthermore, using an "alternate" link, the server indicates support for another profile with <http://example.org/shapes/shape-4> as profile URI. Note that, even if the user agent does not express profile preferences using the "Accept-Profile" header and the server's "Vary" header would be the same, the "Link" header would still include a "profile" link to indicate the profile of the representation returned by the server.

```
HTTP/1.1 200 OK
Content-Type: text/turtle
Vary: Accept, Accept-Profile
Link: <http://example.org/shapes/shape-2>
      ; rel="profile" ,
      <http://example.org/shapes/shape-3>
      ; rel="profile" ,
      <http://example.org/document>
      ; rel="alternate"
      ; type="text/turtle"
      ; formats="http://example.org/shapes/shape-4"
Content-Length: 8724
Connection: close

...
```

Figure 7: Response honors a user agent’s preference

Figure 8 shows the response to the request of Figure 6 in case the server supports profile negotiation for the resource at hand but can not return a representation that conforms to a profile preferred by the user agent. The server has chosen to nevertheless return a representation that conforms to profile <http://example.org/shapes/shape-4>, which is not among the ones preferred by the user agent. The server also reveals the existence of a representation that conforms to profile <http://example.org/shapes/shape-5>. The server could also choose not to return a default representation in which case it would return a "406 Not Acceptable" HTTP response code and no response body. It would not provide any "profile" links but might use "alternate" links with a "formats" attribute to indicate the existence of supported profiles.

```
HTTP/1.1 200 OK
Content-Type: text/turtle
Vary: Accept, Accept-Profile
Link: <http://example.org/shapes/shape-4>
      ; rel="profile" ,
      <http://example.org/document>
      ; rel="alternate"
      ; type="text/turtle"
      ; formats="http://example.org/shapes/shape-5"
Content-Length: 6333
Connection: close

...
```

Figure 8: Response does not honor a user agent's preference but includes default representation

Figure 9 shows the response to the request of Figure 6 in case the server does not support profile negotiation for the resource <http://example.org/document>. It does support negotiation in the media type dimension and has honoured one of the user agent's preferences with that regard, as can be seen by the "Vary" and "Content-Type" headers.

```
HTTP/1.1 200 OK
Content-Type: text/turtle
Vary: Accept
Content-Length: 8724
Connection: close

...
```

Figure 9: Response indicates lack of support for proactive profile negotiation

4.1.2. Reactive Profile Negotiation

In reactive profile negotiation, the user agent selects the profiled representation that best meets its preferences on the basis of a list of possible representations it obtains from the server. A server that supports reactive profile negotiation MUST provide such a list of supported profiled representation as a set of links in the "Link" header. Each of these links:

- SHOULD have the "alternate" relation type.

- MUST use the "formats" link hint to convey the profile URI of the profile to which the resource that is the link target conforms.

- SHOULD use the "allow" link hint to convey the HTTP methods that are supported by the resource that is the link target.

Figure 10 shows a user agent issuing a HTTP HEAD on a resource in order to determine whether profiled representations are available for it. Figure 11 shows the response of a server that supports reactive profile negotiation. By means of "alternate" links in the "Link" header, the server indicates support for two profiled representations for the resource at hand, and, for each, indicates the URI at which they can be accessed, as well as their respective profile URIs, media types, and supported HTTP methods. On the basis of this response, the client can decide whether any of the linked resources conform to a preferred profile, and, if so, access the respective link target. Figure 12 shows the response to an HTTP HEAD issued on the link target <http://example.org/bibrecord/1/DC> of the first "alternate" link.

```
HEAD /bibrecord/1 HTTP/1.1
Host: example.org
Accept: application/xml
Connection: close
```

Figure 10: Client determines support for profiles

```
HTTP/1.1 200 OK
Content-Type: text/plain
Link: <http://example.org/bibrecord/1/DC>
      ; rel="alternate"
      ; type="application/xml"
      ; formats="http://purl.org/dc/terms/"
      ; allow="HEAD,GET,PATCH" ,
      <http://example.org/bibrecord/1/BIBFRAME>
      ; rel="alternate"
      ; formats="http://id.loc.gov/ontologies/bibframe/"
      ; allow="HEAD,GET"
Content-Length: 200
Connection: close
```

Figure 11: Server supports two profiles

```
HTTP/1.1 200 OK
Content-Type: application/xml
Link: <http://purl.org/dc/terms/>
      ; rel="profile" ,
      <http://example.org/bibrecord/1/BIBFRAME>
      ; rel="alternate"
      ; formats="http://id.loc.gov/ontologies/bibframe/"
      ; allow="HEAD,GET"
Allow: HEAD, GET, PATCH
Accept-Patch: application/xml-patch+xml
Content-Length: 458
Connection: close
```

Figure 12: Response to a client accessesing a profiled representation

4.2. Accept-Profile HTTP Header Syntax

Figure 13 describes the syntax of the "Accept-Profile" HTTP header, using the grammar defined in RFC 5234 [RFC5234] and the rules defined in Section 3.2 of RFC 7230 [RFC7230]. The definitions of "URI-reference" and "weight" are imported from RFC 7230 [RFC7230] and RFC 7231 [RFC7231], respectively.

```
Accept-Profile = "Accept-Profile" ":"
OWS (accept-value) *(OWS "," OWS accept-value) OWS
accept-value = "<" URI-reference ">" [weight] | accept-value-ext
```

Figure 13: ABNF for the "Accept-Profile" HTTP header

5. Writing Profiled Representations

A user agent that wants to submit a profiled representation to a server can use the reactive negotiation approach to determine the nature of a server's support with this regard.

A server that allows user agents to submit profiled representations SHOULD follow the directions for reactive negotiation described in Section 4.1.2.

A client that submits a representation that complies to a profile that was not advertised by the server by means of the reactive negotiation approach, SHOULD assume that the server is not able to process it.

A server that fails a submission request due to receiving a payload with a profile that it does not support MUST respond with a "422

Unprocessable Entity" HTTP status code and SHOULD use the approach described in Section 4.1.2 to convey profiles that are supported.

Continuing from Figure 12, Figure 14 shows a user agent issuing a HTTP PATCH against resource <http://example.org/bibrecord/1/DC> in order to update it. It uses the "Content-Encoding" header to convey the XML Patch media type of its message body and a link with a "profile" relation type in the "Link" header to indicate the profile to which it conforms. Figure 15 shows the server's response, indicating that the patch was applied successfully; the server returns the updated representation as message body.

```
PATCH /bibrecord/1/DC HTTP/1.1
Host: example.org
Content-Encoding: application/xml-patch+xml
Link: <http://purl.org/dc/terms/>
      ; rel="profile"
Accept: application/xml
Content-Length: 321
Connection: close

...
```

Figure 14: Client submits profiled representation

```
HTTP/1.1 200 OK
Content-Type: application/xml
Content-Location: http://example.org/bibrecord/1/DC
Link: <http://example.org/bibrecord/1/DC>
      ; rel=profile" ,
      <http://example.org/bibrecord/1/BIBFRAME>
      ; rel="alternate"
      ; formats="http://id.loc.gov/ontologies/bibframe/"
      ; allow="HEAD,GET"
Allow: HEAD, GET, PATCH
Accept-Patch: application/xml-patch+xml
Content-Length: 592
Connection: close

...
```

Figure 15: Server indicates successful submission of profiled representation by client

If, in Figure 14, the user agent would have issued an HTTP PATCH on resource <http://example.org/bibrecord/1/DC> indicating that the

profile URI of the patch was <http://id.loc.gov/ontologies/bibframe/>, the response would have a 422 HTTP status code to express that the profile is not supported by the resource at hand. Such a response is shown in Figure 16.

```
HTTP/1.1 422 Unprocessable Entity
Content-Type: text/plain
Link: <http://example.org/bibrecord/1/DC>
      ; rel="alternate"
      ; type="application/xml"
      ; formats="http://purl.org/dc/terms/"
      ; allow="HEAD,GET,PATCH"
Content-Length: 110
Connection: close

...
```

Figure 16: Server indicates unsuccessful submission of profiled representation by client

6. IANA Considerations

This memo requires IANA to register the Accept-Profile HTTP header defined in Section 4.2 in the appropriate IANA registry:

- Header Field Name: Accept-Profile
- Applicable Protocol: Hypertext Transfer Protocol (HTTP)
- Status: Informational
- Author/Change controller: IETF
- Specification document(s): this document

7. Security Considerations

8. Acknowledgments

Many thanks to our colleagues for feedback: Enno Meijers, Michael L. Nelson

9. References

9.1. Normative References

[I-D.nottingham-link-hint]
Nottingham, M., "HTTP Link Hints", draft-nottingham-link-hint-02 (work in progress), March 2020.

[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>.

[RFC5234] Crocker, D., Ed. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, DOI 10.17487/RFC5234, January 2008, <https://www.rfc-editor.org/info/rfc5234>.

[RFC6838] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, DOI 10.17487/RFC6838, January 2013, <https://www.rfc-editor.org/info/rfc6838>.

[RFC6906] Wilde, E., "The 'profile' Link Relation Type", RFC 6906, DOI 10.17487/RFC6906, March 2013, <https://www.rfc-editor.org/info/rfc6906>.

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <https://www.rfc-editor.org/info/rfc7230>.

[RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <https://www.rfc-editor.org/info/rfc7231>.

9.2. Informative References

[HeeryAndPatel]
Heery, R. and M. Patel, "Application Profiles: Mixing and Matching Metadata Schemas", 2000, <http://www.ariadne.ac.uk/issue25/app-profiles>.

[RFC4452] Van de Sompel, H., Hammond, T., Neylon, E., and S. Weibel, "The "info" URI Scheme for Information Assets with Identifiers in Public Namespaces", RFC 4452, DOI 10.17487/RFC4452, April 2006, <https://www.rfc-editor.org/info/rfc4452>.

[RFC5988] Nottingham, M., "Web Linking", RFC 5988, DOI 10.17487/RFC5988, October 2010, <https://www.rfc-editor.org/info/rfc5988>.

[RFC8141] Saint-Andre, P. and J. Klensin, "Uniform Resource Names (URNs)", RFC 8141, DOI 10.17487/RFC8141, April 2017, <https://www.rfc-editor.org/info/rfc8141>.

[RFC8288] Nottingham, M., "Web Linking", RFC 8288, DOI 10.17487/RFC8288, October 2017, <https://www.rfc-editor.org/info/rfc8288>.

[W3C.REC-shacl-20170720] Knublauch, H. and D. Kontokostas, "Shapes Constraint Language (SHACL)", World Wide Web Consortium Recommendation REC-shacl-20170720, July 2017, <https://www.w3.org/TR/2017/REC-shacl-20170720>.

[W3C.REC-xmlschema11-1-20120405] Gao, S., Sperberg-McQueen, M., Thompson, H., Mendelsohn, N., Beech, D., and M. Maloney, "W3C XML Schema Definition Language (XSD) 1.1 Part 1: Structures", World Wide Web Consortium Recommendation REC-xmlschema11-1-20120405, April 2012, <https://www.w3.org/TR/2012/REC-xmlschema11-1-20120405>.

Authors' Addresses

Lars G. Svensson

Email: lars.svensson@web.de
URI: https://orcid.org/0000-0002-8714-9718

Ruben Verborgh
Ghent University - imec
Sint-Pietersnieuwstraat 41
Ghent 9000
Belgium

Phone: +32 9 331 49 10
Email: ruben.verborgh@ugent.be
URI: https://ruben.verborgh.org/

Herbert Van de Sompel
Data Archiving and Networked Services
Anna van Saksenlaan 51
The Hague 2593 HW
Netherlands

Email: herbert.van.de.sompel@dans.knaw.nl
URI: https://orcid.org/0000-0002-0715-6126