QUICWG
Internet-Draft
Intended status: Standards Track
Expires: 5 November 2026

F. Rochet
UNamur

4 May 2026

Reverso for the QUIC protocol
draft-frochet-quicwg-reverso-for-quic-01

Abstract

This document describes a QUIC version re-designing the layout of the QUIC protocol to avoid memory fragmentation at the receiver and allows implementers seeking a more efficient implementation to have the option to implement contiguous zero-copy at the receiver. This document describes the change from QUIC v1 required in packet formats, variable-length integers and frame formats. Everything else from QUIC v1 described in [RFC9000] is untouched.

About This Document

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

Status information for this document may be found at https://datatracker.ietf.org/doc/draft-frochet-quicwg-reverso-for-quic/.

Source for this draft and an issue tracker can be found at https://github.com/https://github.com/frochet/draft-rochet-reverso-for-quic.

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 5 November 2026.

Copyright Notice

Table of Contents

1. Introduction

QUIC is a general-purpose transport protocol with mandatory encryption leveraged from a TLS 1.3 key exchange. QUIC is specified in [RFC9000], and is the result of several years of efforts from several major companies, independent individuals and academics. One of the main benefits of QUIC is to resist ossification thanks to a two-level encryption design (header and payload), supporting extensions and modifications of internal QUIC information to resist friction from independent lower layers at deployment time.

However, it is of notoriety that the QUIC design is CPU costly. The root cause of QUIC's high CPU cost isn't unique, and this document addresses one of them: a misalignment between QUIC's protocol specification and encryption integration. Indeed, the QUIC design in [RFC9000] unavoidably fragments Application Data and forces any implementation to perform at least a memory copy to provide a contiguous bytestream abstraction to the upper layer, at the receiver.

This document suggests another QUIC Version demanding the Stream frame to always be the first frame if any, and reversing the wire representation of the QUIC protocol. These two changes offer the opportunity for implementers to provide a contiguous zero-copy abstraction at the receiver side for each stream using the decryption internal copy for data reassembly. With this version, QUIC frames are encoded in reverse ordering and would be written and processed from right to left, instead of the usual left to right as in any protocol. The stream frame may be followed by any number of control frames up to the packet boundary. Other stream frames may be packed within the same packet, although receiver implementations would not be able to process them in contiguous zero-copy.

2. Goals

We aim to change how the QUIC protocol specifies its frames to support a stream abstraction with the option to offer a contiguous zero-copy interface to the upper layer. A few more bytes also have to be added within the protected short header. Those changes are, however, engineered with goals to:

* Minimizing added control overheads.

* Not requiring a different frame encoding/decoding code logic.

Despite the change of the wire format, the code for writing and processing QUIC frames does not need adaptation as long as a protocol-independent buffer abstraction to write and read from right to left exists.

* Not mandating existing QUIC implementations to support this version. Can fallback to QUIC v1 (by the QUIC protocol negotiation design).

* Not modifying any of the QUIC's transport properties (i.e., HoL blocking avoidance, multiplexing, extensibility, ...) and not conflicting with the goals of any ongoing work on QUIC extensions (e.g., MPQUIC) otherwise than by requiring them to change their wire representation as well.

* Not impacting QUIC's security/safety assuming implementers follow added guidance to Reverso.

* Keeping Encryption/decryption compatible with the current usage of existing crypto libraries.

3. Conventions and Definitions

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.

4. Streams

Stream ID values start at 1. We reserve the value 0 to indicate within the new short header (see Section 6.1) that no stream frame is packed within the encrypted payload.

5. Frame Formats

Frames' structure written on the wire is altered in this QUIC version to support a backward processing of QUIC packets. All the other frames are straightforward to adapt from [RFC9000]. Essentially, on [RFC9000], each frame begins with a Frame Type followed by additional type-dependent fields, and is represented as this:

```
Frame {
  Frame Type (i),
  Type-Dependent Fields (..),
}
```

This representation follows the implicit rule that what we specify, from top to bottom, is written and read from left to right on the wire.

If QUIC Reverso is used, frames are reversed. Type-dependent fields appear first (from left to right on the wire), and the frame terminates with the Frame Type. We represent those frames by reversing their representation in specifications:

```
Frame {
  Type-Dependent Fields (..),
  Frame Type (i),
}
```

The choice of order of Type-Dependent Fields only matters to ease the transition and adaptation of existing code handling [RFC9000]'s frame format. Reversing the existing ordering and not making other changes within the relative order of elements may support straightforward

adaptation of existing code. For example, in [RFC9000], the MAX_STREAM_DATA Frame is defined as:

```
MAX_STREAM_DATA Frame {
  Type (i) = 0x11,
  Stream ID (i),
  Maximum Stream Data (i),
}
```

Which would translate to:

```
MAX_STREAM_DATA Frame {
  Maximum Stream Data (i),
  Stream ID (i),
  Type (i) = 0x11,
}
```

6. Packet Formats

For implementers to take advantage of Reverso and use the decryption internal copy for data reassembly, we require to know the Stream ID of any stream frame within the payload and the data offset. These two integers are added to the QUIC short header and protected with the mask using a XOR. In QUIC v1, 5 out of 16 bytes available are being used. In Reverso, we would use 13 out of 16 bytes.

6.1. Header Protection

The header of 1-RTT short header packets is extended to add at most 8 bytes of information, requiring a 13-byte mask. Application of the mask follows the same procedure as specified in [RFC9001], as a minimum of 16 bytes are currently available from the current header protection algorithms.

```
1-RTT Packet {
  Header Form (1) = 0,
  Fixed Bit (1) = 1,
  Spin Bit (1),
  Reserved Bits (2),         # Protected
  Key Phase (1),             # Protected
  Packet Number Length (2),  # Protected
  Destination Connection ID (0..160),
  Packet Number (8..32),     # Protected
  Stream ID (8..32)          # Protected
  Offset (8..32)             # Protected
  Protected Payload (0..72), # Skipped Part
  Protected Payload (128),   # Sampled Part
  Protected Payload (..),    # Remainder
}
```

The 1-RTT packets has the following modifications from QUIC v1:

* Packet Number: The Packet Number field is 1 to 4 bytes long, with the least two significant bits of the last byte containing the length of the Stream ID. This length is encoded as an unsigned two-bit integer that is one less than the length of the Stream ID field in bytes. This field is protected using [RFC9001]'s mask, which can consume a maximum of 5 bytes (including the first header byte) from the guaranteed 16 bytes.

* Stream ID: The Stream ID field is 1 to 4 bytes long, with the least two significant bits of the last byte containing the length of the Offset. This length is encoded as an unsigned two-bit integer that is one less than the length of the Offset field in bytes. A 1-byte value of 0 for this field is reserved to indicate that the encrypted payload does not contain any Stream frame.

This field is protected using [RFC9001]’s mask, up to consume 9 bytes from the guaranteed 16 bytes in total.

* Offset: The Offset field is 1 to 4 bytes long, and encodes a value based on the knowledge of the maximum acknowledged offset, similar to the Packet Number field but encoding a value based on the highest acknowledged offset. On the receiver, the decoding procedure is similar to decoding packet numbers. This field is protected using [RFC9001]’s mask, up to consume 13 bytes from the guaranteed 16 bytes in total.

* Protected Payload Skipped Part’s length: 72 bits are skipped instead of 24. 24 bits are skipped in QUIC v1 in order to account for the maximum (yet unknown) length of the Packet Number when sampling the encrypted payload for header decryption. Since we add variable integers, we need sampling further away to guarantee always falling into the AEAD encryption (and/or tag). We need skipping 72 bits to account for the maximum combined (yet unknown) lengths of Packet Number, Stream ID and offset. This affects the minimum payload length for preparing a QUIC packet at the sender, which was following the relation in QUIC v1:

pn_len + min_payload_len + tag_len = 4 + sample_len

=> min_payload_len := 4 + sample_len - tag_len - pn_len => min_payload_len := 20 - tag_len - pn_len

as defined in [RFC9001], where a safe static value can be set to 3 bytes for min_payload_len (i.e., it is the max value of the upper relation). In VReverso, the relation becomes:

pn_len + stream_id_len + offset_len + min_payload_len + tag_len = 12 + sample_len

=> min_payload_len := 12 + sample_len - tag_len - pn_len - stream_id_len - offset_len => min_payload_len := 28 - tag_len - pn_len - stream_id_len - offset_len

A safe static value for min_payload_len can be set to 9 bytes in implementations.

6.2. Stream ID encoding (to debate)

The QUIC v1 protocol supports up to 2^{60} maximum streams. A QUIC Reverso implementation must encode a Stream ID within at most 30 bits in its header. Due to the monotonic increasing nature of Stream IDs, we can work out a solution that still permits up to 2^{60} maximum streams, but constraints endpoints to fire at most 2^{30} new streams at any time. We consider (up to debate) this constraint to exceed any reasonable usage of the QUIC protocol given the memory requirement to hold up to 2^{30} opened streams. Different solutions are possible. An approach could be to reuse packet number encoding/ decoding, but based on acknowledged new streams.

6.3. Frame ordering

In Reverso, a Stream Frame, if any, MUST be the first frame within the payload. The Stream frame can be followed by any number of control frames up to the packet boundary. Any other Stream frame SHOULD NOT be added within the same QUIC packet, unless in scenarios where multiplexing may bring more benefits than contiguous zero-copy (e.g., multiplexed HTTP queries within a single packet).

7. Variable-Length Integer Encoding

QUIC v1 uses variable-length encoding for non-negative integer values

to encode fewer bytes on the wire than usual host representations. In QUIC v1 the encoding reserves the two most significant bits of the first byte, and encodes the integer in the remaining bits in the network byte order. In this version, we encode the length in the two least significant bits of the last byte to accommodate processing the information by rewinding the buffer. The remaining bits encode the integer value in the network byte order.

2LSB	Length	Usable Bits	Range
00	1	6	0-63
01	2	14	0-16383
10	4	30	0-1073741823
11	8	62	0-4611686018427387903

Table 1: Summary of Integer Encodings with Reverso

8. Security, Safety and Liveness Considerations

The goal of this section is to discuss careful considerations which a QUIC Reverso implementation must consider while implementing a contiguous zero-copy receiver interface.

8.1. Avoiding Data Corruption

Contiguous zero-copy with Reverso is obtained by exploiting the added information in the short header and the decryption's internal copy to reassemble data fragments. For payload decryption, the Stream ID contained within the short header should be used as a buffer selection mechanism, and the offset is used to locate where to decrypt the packet content within the buffer.

AEAD implementations may write at the destination address specified by the caller even if the decryption fails. Therefore, receivers must track the highest contiguous received authenticated offset for each stream and always decrypt in place any packet containing an offset below or equal to the tracked value. Furthermore, implementers must be careful with data gaps within a stream buffer created due to out-of-order packets, where the decryption of a late out-of-order packet may override part of the existing buffered data.

Different implementation solutions are possible to deal with this issue. In all cases it involves a copy of the decrypted data. A possible solution is to apply the following logic:

if the packet's data is to be decrypted at a location higher than the highest received contiguous offset + 1, and if the AEAD ciphertext is of size N and aimed at location L in the stream buffer, check whether the range L..L+N does not contain any previously decrypted data. Decrypt in place if the answer is yes to avoid data corruption, and safely copy to location L in the stream buffer.

8.2. Manipulating Short Header bits may cause hitting Stream Limits

A QUIC Reverso implementation may allocate a new stream context before a packet containing a new Stream is decrypted. If an on-path adversary flips bits in the encrypted header it would result in a flipped bit in the decrypted header as per [RFC9001]'s XOR properties used for header protection. Such an event would be detected in the AEAD decryption phase, since the AEAD decryption would fail. In the meantime, any memory allocated related to a new Stream context

resulting from the adversarial manipulation would need to be released, and any stream limit modification would need to be credited back.

9. References

9.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <https://www.rfc-editor.org/rfc/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/rfc/rfc8174>.

9.2. Informative References

[RFC9000] Iyengar, J., Ed. and M. Thomson, Ed., "QUIC: A UDP-Based Multiplexed and Secure Transport", RFC 9000, DOI 10.17487/RFC9000, May 2021, <https://www.rfc-editor.org/rfc/rfc9000>.

[RFC9001] Thomson, M., Ed. and S. Turner, Ed., "Using TLS to Secure QUIC", RFC 9001, DOI 10.17487/RFC9001, May 2021, <https://www.rfc-editor.org/rfc/rfc9001>.

Appendix A. Acknowledgments {:numbered="false"}

TODO acknowledge.

Author’s Address

Florentin Rochet
UNamur
Email: florentin.rochet@unamur.be