

QUIC Working Group
Internet-Draft
Intended status: Informational
Expires: September 6, 2018

E. Rescorla
RTFM, Inc.
March 05, 2018

QUIC over DTLS
draft-rescorla-quic-over-dtls-00

Abstract

QUIC in-band cryptographic negotiation on stream 0 creates a number of odd edge cases. This document considers an alternative design in which QUIC transport is run directly over DTLS, thus separating the cryptographic negotiation from the transport piece.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of [BCP 78](#) and [BCP 79](#).

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

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

This Internet-Draft will expire on September 6, 2018.

Copyright Notice

Copyright (c) 2018 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Internet-Draft

QUIC/DTLS

March 2018

Table of Contents

1.	Introduction	2
2.	Design Overview	4
2.1.	Reliability	6
2.1.1.	DTLS Reliability and ACKs	6
2.1.2.	RTT Estimation	6
2.1.3.	QUIC Reliability and ACKs	6
2.2.	Version Negotiation	7
2.3.	Transport Parameters	8
2.4.	Key Changes and New Session Ticket	8
2.5.	Connection IDs	9
3.	Required Changes to DTLS	9
3.1.	Handshake Obfuscation	9
3.2.	Obfuscation Negotiation Packet	11
3.3.	Packet Header Encryption	11
3.4.	Stateless Reset	11
4.	Required Changes to QUIC Documents	12
4.1.	TLS Document	12
4.2.	Transport Document	12
4.3.	Recovery Document	13
4.4.	Invariants Document	13
5.	Potential Additional Benefits	13
5.1.	Record Coalescence	13
5.2.	More Straightforward Demuxing	13
6.	Potential Points of Concern	13
6.1.	Record Size/Overhead	14
6.2.	Status of DTLS 1.3 Maturity	14
6.3.	Epoch and ACKs	15
6.4.	Crypto Layer Agility	15
7.	WebRTC Binding	15
8.	Implementation Status	16
9.	Security Considerations	16
10.	References	16
10.1.	Normative References	16
10.2.	Informative References	17
	Author's Address	17

[1.](#) Introduction

QUIC [[I-D.ietf-quic-transport](#)] [[I-D.ietf-quic-tls](#)] as currently designed performs cryptographic negotiation by sending TLS 1.3 [[I-D.ietf-tls-tls13](#)] traffic directly in stream 0. This design was

the result of desiring to have tight coupling between the cryptographic handshake and the transport and has a number of advantages in that it allows the TLS 1.3 flow to take advantage of QUIC's transport services, in particular reliable in-order delivery,

RTT estimation, etc. However, it also results in several unpleasant corner cases:

- o The cryptographic handshake stream is subject to a variety of odd rules, such as:
 - * Stream 0 is unencrypted at the beginning of the connection, but encrypted after the handshake completes.
 - * Stream 0 is not subject to flow control; it can exceed limits and goes into negative credit after the handshake completes.
 - * Retransmission of stream 0 frames from lost packets needs special handling to avoid accidentally encrypting them.
 - * Stream 0 offsets are reset after the server sends a Retry packet. This creates special handling rules for the stream.
- o The QUIC stack needs tight coupling with the TLS stack to know, for instance, whether the TLS stack sent SH or HRR, or where the boundaries are between flights. See, Issue #1094 for an example of this kind of problem.
- o There are complicated rules about which packets can ACK other packets, as both cleartext and ciphertext ACKs are possible.
- o The interaction of the state machine advancing (which makes clear that packets have received) and ACKs is confusing. For instance, it is possible to respond to an Initial packet but not ACK it. The semantics of this are unclear.
- o There are complicated rules for how to handle 0-RTT (and especially 0-RTT failures).
- o The stack needs to continue to service stream 0 indefinitely in order to gather NewSessionTicket messages.

- o QUIC version negotiation isn't authenticated, so it is retroactively authenticated during the TLS handshake.

This document considers a design at the other end of the spectrum, which is to layer QUIC transport over DTLS 1.3 [[I-D.ietf-tls-dtls13](#)] (with some small pieces of coupling). This design addresses most of the corner cases described above, although it does introduce some new issues which must be considered.

2. Design Overview

The current QUIC/TLS integration treats TLS as a module which gets a lot of its services from QUIC. I.e., something like this:

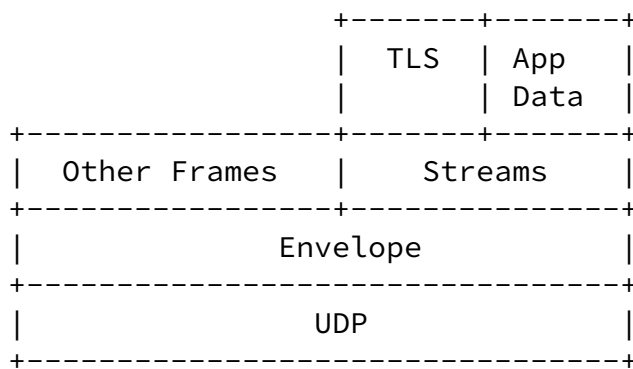
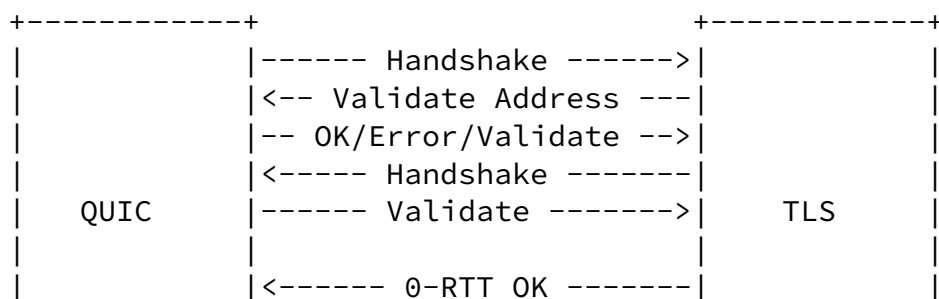


Figure 1: Current QUIC Architecture

This creates a relatively complicated interaction, as shown in the following diagram from [\[I-D.ietf-quic-tls\]](#).



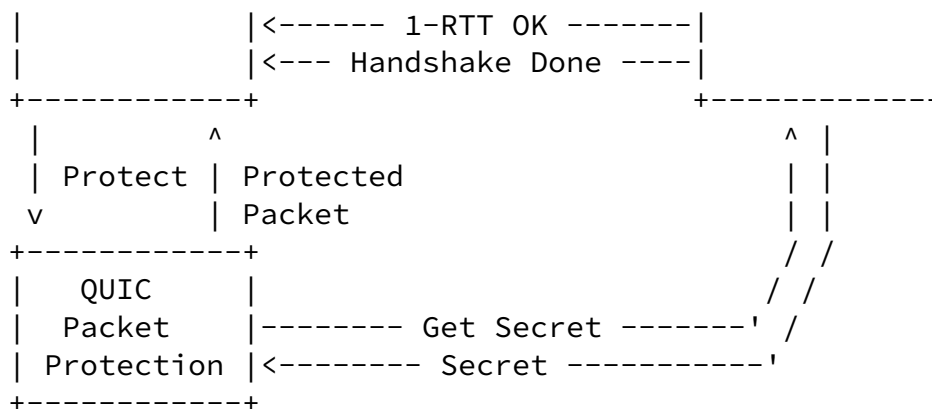


Figure 2: QUIC/TLS Interactions

In the design proposed by this document, we have a more natural layered structure, similar to that of HTTP2 over TLS. I.e.,

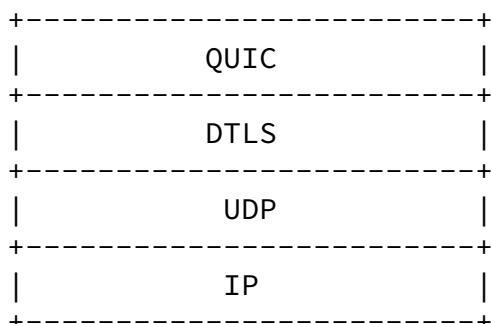


Figure 3: QUIC over DTLS architecture

Of course, you still need some signaling between QUIC and DTLS, but it's largely the conventional signaling that any application protocol over TLS needs. Specifically:

- o It's possible to send data (in 0-RTT or 1-RTT)
- o The handshake is complete
- o Here are the cryptographic parameters

- o The record number that corresponds to a given piece of data.

The only real special accommodation needed is to carry the QUIC transport parameters, which you already needed in the current design (although it's not shown). You may also want to expose DTLS's RTT estimates to QUIC (see [Section 2.1.2](#)) but this is not necessary to have a functional system.

Operationally, this is straightforward. You negotiate DTLS and then send QUIC frames over DTLS as type application data, as shown in Figure 4.

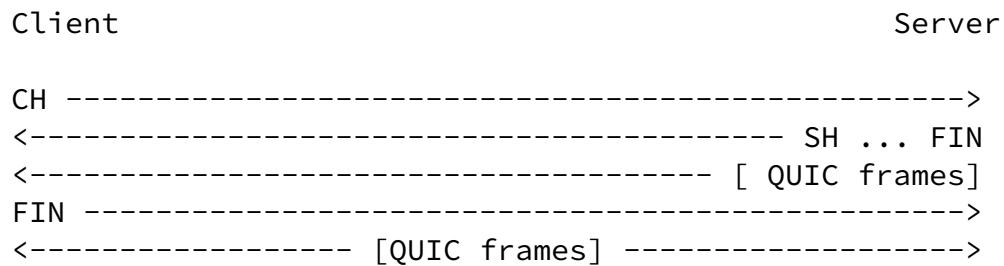
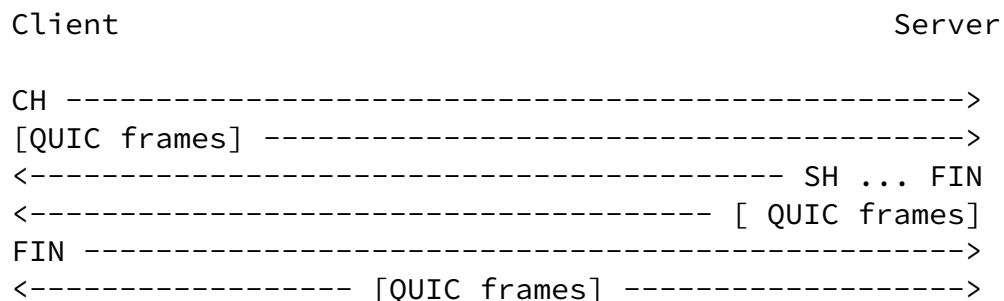


Figure 4: Simple QUIC over DTLS

The payload of each application data DTLS record consist of QUIC frames laid end to end, precisely as in current QUIC packets. Because DTLS application data is always encrypted, this means that the QUIC frames themselves are also encrypted.

When 0-RTT is used, things are as you expect, namely that QUIC frames can be sent in the early data, as shown in Figure 5. If 0-RTT is rejected, then the frames can just be treated as lost and retransmitted, though you probably want to do so immediately and some care must not be taken to let this modify your congestion state.



[2.1. Reliability](#)

[2.1.1. DTLS Reliability and ACKs](#)

In this design, DTLS takes care of its own reliability for the handshake (and for post-handshake messages) via a timeout and retransmission scheme, including ACKs. DTLS ACKs apply only to handshake records and so simply don't apply to QUIC frames at all. DTLS ACKs apply to both encrypted and unencrypted handshake records.

[2.1.2. RTT Estimation](#)

One virtue of the current design is that you get an RTT estimate from the QUIC congestion control machinery for packets that are sent during the handshake phase. Naively you would lose this, but DTLS actually gives you a primitive estimate of this via its own reliability mechanisms and ACKs. So the natural thing to do here is to have the DTLS implementation derive an RTT estimate from the DTLS handshake from the first round trip in each direction and then provide it to the QUIC part of the stack as an initial estimate.

[2.1.3. QUIC Reliability and ACKs](#)

By contrast, QUIC ACKs only apply to application data records (which contain encrypted QUIC frames) and are just sent with whatever the current DTLS epoch is.

One subtlety here is what the QUIC ACKs acknowledge, because we no longer have QUIC packets. One way of handling this would be to have packet numbers as a header inside each application data record, i.e.,:

[Packet Number] [Frame] [Frame]... [Frame]

However, this wastes space, so a better approach is to just refer to the DTLS record number. This is the primary piece of layer violation in this design in that the QUIC stack will need to be aware of:

- o Which DTLS record number a given frame went out on

- o Which DTLS record number a given frame came in on

This is a tiny bit ugly, but isn't complicated to implement.

[2.2.](#) Version Negotiation

In current QUIC, you negotiate two versions:

- o The QUIC version by having the client propose a version and then the server corrects it.
- o The TLS version which is negotiated by having the client propose all its versions and then the server selects one.

When we reorder QUIC and DTLS, a different design is appropriate. DTLS currently doesn't encrypt the ClientHello, so obfuscation does not immediately become an issue, and instead we can have both QUIC and DTLS versions negotiated in the customary way, namely: the client proposes two sets of versions:

- o DTLS versions in "supported_versions"
- o QUIC versions in "quic_versions" (new)

And then the server chooses its preference for both of them.

Assuming that we want to continue to have handshake obfuscation, we will need to modify DTLS to allow this (more details about this in [Section 3.1](#)), and then we can adopt the current scheme with a small set of modifications.

- o The records containing the ClientHello and ServerHello contain a DTLS version. This reflects the version of DTLS whose obfuscation constants are in use.
- o If the server recognizes that version, then it simply decrypts the handshake messages and does DTLS version negotiation mechanism (see Figure 6).

- o If the server doesn't recognize that version, it sends a VN packet

as with current QUIC (we would need to add this to DTLS). The client then re-sends the ClientHello except obfuscated with a different version. Note that you use the same ClientHello.supported_versions and so an attacker cannot impact DTLS version negotiation (see Figure 7).

Client	Server
CH version = A[supported_versions = A, B, C] ----->	
<----- SH version = B [supported_version = B]	

Figure 6: Version negotiation with compatible obfuscation

Client	Server
CH version = A[supported_versions = A, B, C] ----->	
<----- VN versions = [B, C]	
CH version = B[supported_versions = A, B, C] ----->	
<----- SH version = B [supported_version = B]	

Figure 7: Version negotiation with incompatible obfuscation

Note that this design is 1-RTT faster than the current design in cases where the server has changed it's preferred version. In current QUIC, that results in a VN, but here the server can just remember the old constants, decrypt the record, and proceed with negotiation as usual.

[2.3.](#) Transport Parameters

The QUIC transport parameters can be negotiated in DTLS extensions as they currently are.

[2.4.](#) Key Changes and New Session Ticket

Because the QUIC frames are carried over DTLS, the DTLS stack naturally reads and writes DTLS records that are not carrying application data (primarily handshake) as well. This means that post-handshake messages such as KeyUpdate and NewSessionTicket just work naturally: the peer sends them and the DTLS stack consumes them, transparently to the QUIC stack. Note that this is an improvement over the current design, in which the QUIC stack needs to continue to pass data back and forth to the TLS stack on stream 0. Key changes just take effect according to the DTLS schedule.

[2.5.](#) Connection IDs

DTLS 1.3 has no native support for connection IDs, but instead has farmed it out to a separate draft [[I-D.ietf-tls-dtls-connection-id](#)]. That draft takes a slightly different strategy from QUIC in which both sides provide the connection ID for the peer to use to send to them. However, we are currently considering precisely such a design in QUIC (see <https://github.com/quicwg/base-drafts/issues/1089>) in which case, much of the design would drop into place.

The remainder of this document assumes that we are using asymmetric connection IDs, as that seems like the direction we are going, but this part of DTLS is still under active design and so could presumably be modified to support QUIC's needs.

[3.](#) Required Changes to DTLS

Some modest changes would be needed for DTLS to ensure parity with the current QUIC design. These changes would reuse design work we have already done for QUIC.

In addition to the existing DTLS 1.3 capabilities and the connection ID work, the following changes are needed:

- o Handshake Obfuscation, [Section 3.1](#)
- o Obfuscation Negotiation, [Section 3.2](#)
- o Packet Number Encryption, [Section 3.3](#)
- o Stateless Reset, [Section 3.4](#)

These would each add a generic capability to DTLS that could be used by other protocols.

[3.1.](#) Handshake Obfuscation

As noted above, DTLS does not presently encrypt the ClientHello and ServerHello messages (the remainder of the messages are in the clear). However, there's an elegant way to handle this, as suggested by Martin Thomson, which is to define the format of DTLSPlaintext as being version specific. Recall the standard DTLSPlaintext:

Internet-Draft

QUIC/DTLS

March 2018

```
struct {
    ContentType type;
    ProtocolVersion version;
    uint16 epoch = 0 // DTLS field
    uint48 sequence_number; // DTLS field
    uint16 length;
    opaque fragment[DTLSPlaintext.length];
} DTLSPlaintext;
```

We can simply declare that when `DTLSPlaintext.version` is set to 1.3 or above, we instead have:

```
struct {
    ContentType opaque_type = handshake; // Real CT inside the encryption
    ProtocolVersion version;
    opaque dst_conn_id<0..255>
    opaque src_conn_id<0..255>;
    uint16 epoch = 0 // DTLS field
    uint48 sequence_number; // DTLS field
    uint16 length;
    opaque encrypted_record[length];
} DTLSHandshakeCiphertext;
```

This format would be used only for the initial handshake and any alerts or acks it generates (i.e., data in epochs 0, 1, and 2). For epoch 0 (what would be plaintext in TLS 1.3), the obfuscation is done essentially as with QUIC, i.e., with a KDF keyed with a per-connection function and the dst CID. For all other epochs, you encrypt just as you would with DTLS 1.3 today.

Note the difference here from TLS 1.3 in which these packets get `type=handshake`, which with the record version tells us that the CID variant is in use.

Note that this is essentially the same structure here as I recently proposed for asymmetric CIDs.

An advantage of this design is that it has a natural "backward compatible mode" for DTLS where you can use the old format if you

don't know what the peer supports (conventional D/TLS behavior) but if you have a minimum version that's 1.3 or above you can get obfuscation.

[[OPEN ISSUE: Grease the length bytes.]]

[3.2.](#) Obfuscation Negotiation Packet

In the case that the client chooses an obfuscation scheme that the server does not understand, we need a way for the server to request a different obfuscation scheme.

This is analogous to the QUIC Version Negotiation packet, but it only negotiates the choice of obfuscation. Because it is separate from the negotiation of the QUIC version, it can be much lighter weight.

We can more or less directly steal the QUIC Version Negotiation scheme. That is, the server sends a list of supported versions in place of the packet payload:

```
struct {
    ContentType opaque_type = handshake; // Real CT inside the encryption
    ProtocolVersion version = 0;
    opaque dst_conn_id<0..255>
    opaque src_conn_id<0..255>;
    uint16 epoch = 0 // DTLS field
    uint48 sequence_number; // DTLS field
    uint16 length;
    uint16 supported_versions<2..254>;
} DTLSObfuscationNegotiation;
```

Unlike the current design, where authenticating version negotiation is critical, a downgrade attack only causes the ClientHello and ServerHello to be protected with a different obfuscation scheme. Because obfuscation does not affect the integrity of the protocol negotiation there is no need to add separate authentication for this exchange.

[3.3.](#) Packet Header Encryption

DTLS does not encrypt its packet number and also does not presently have sequence number gaps, which makes changing connection IDs for privacy of modest value. There's no reason we couldn't adapt the same techniques we have discussed using for QUIC, when we get those nailed down. Pretty much all the same issues apply to DTLS and QUIC here.

[3.4.](#) Stateless Reset

As noted above, we would need to define stateless reset. The best way to do this is to define a new extension in Encrypted Extensions for this, so that it's usable for non-QUIC purposes. Otherwise the mechanics can be identical to the current QUIC design in Section 7.9.4 of [[I-D.ietf-quic-transport](#)].

Rescorla

Expires September 6, 2018

[Page 11]

Internet-Draft

QUIC/DTLS

March 2018

[4.](#) Required Changes to QUIC Documents

Obviously, we would require changes to the existing QUIC drafts to make this work. Primarily it's a matter of removal. This section covers the known changes.

[4.1.](#) TLS Document

This document is just removed entirely. We don't need it any more. This removes all or nearly all the crypto from QUIC, and in particular the somewhat complex key schedule, and leaves it in DTLS.

[4.2.](#) Transport Document

Much of this document also goes away. Specifically:

- o We are folding the entire QUIC header into the DTLS header (and stealing a lot of stuff in the process) so basically all of [Section 5](#) (Packet Types and Formats) away.
- o We carry frame types directly over DTLS, so [Section 6](#) (Frames and Frame Types) stays,
- o Essentially all of [Section 7](#) (Life of a Connection) disappears, and is replaced by the DTLS CID and new version negotiation scheme

carried in DTLS extensions (see [Section 2.2](#)). The connection migration design works without modifications, so that would be retained. Similarly, once the DTLS connection is established, QUIC style connection close should be used, with the exception of stateless reset and (optionally) if the DTLS layer decides to abort because of too many deprotection failures. The DTLS close_notify was intended as unreliable and is therefore less capable than QUIC's close.

- o As noted above, ACKs stay essentially the same, but we will need to change the encoding a bit to handle epoch changes, as described in [Section 2.1.3](#) and [Section 6.3](#).
- o [Section 9](#) (Packetization and Reliability), 10 (Streams), and 11 (Flow Control) also stay the same.
- o We'll want to trim down the error space ([Section 12](#)) a little bit because we'll want to use DTLS alerts for a lot of errors. This makes things easier because it avoids the confusion we have now about whether to send a QUIC error or a TLS alert.

[4.3](#). Recovery Document

Almost no changes, except for the rules about handshake recovery.

[4.4](#). Invariants Document

We will need to modify the invariants to match the DTLS invariants, but the principles are the same and the header/packet formats are reasonably similar, so this should be straightforward.

[5](#). Potential Additional Benefits

[5.1](#). Record Coalescence

One challenge in QUIC has been that it is not possible to place two QUIC packets in the same UDP datagram. This creates problems whenever you would like to have two packets encrypted under different keys sent together, as with the Initial packet and 0-RTT data, or

with 1-RTT data (which is encrypted) and the Finished message (which is not).

With DTLS, however, this is straightforward. DTLS has three record header formats:

- o DTLSPlaintext
- o DTLSCiphertext
- o DTLSShortCiphertext

The first two of these have built-in length field and can therefore be packed more than one to a UDP datagram. This allows for simple piggybacking of records of different types. Obviously, this header engineering could be done for QUIC, but its not something we have done.

[5.2.](#) More Straightforward Demuxing

We have had extended discussions about how to demux QUIC with other UDP protocols which might be on the same 5-tuple (principally for WebRTC). Because the existing cases already use DTLS, carrying QUIC over DTLS obviously makes this easier.

[6.](#) Potential Points of Concern

[6.1.](#) Record Size/Overhead

One concern we might have is about record overhead. It's a bit hard to make a straight-up comparison between QUIC and DTLS 1.3 because they make different assumptions about the length of the sequence number field. As noted above, DTLS 1.3 as currently proposed has two different header formats. The longer of the two has a 7 byte header with a 30 bit sequence number and a length field. The shorter is only two bytes with a 12 bit sequence number and no length field. In addition, DTLS 1.3 has a one byte internal content type field that is used to distinguish application data from handshake data and to

support padding.

DTLS has similar record overheads to QUIC, but with a larger increase in size if a longer sequence number is needed. If that turns out to be a problem, there are alternative designs that could be contemplated.

[6.2.](#) Status of DTLS 1.3 Maturity

DTLS 1.3 is a small delta off of TLS 1.3. In Singapore there were no remaining open issues, and a proposal to go to WGLC once we had more implementation experience, which we are accumulating now, so absent input from QUIC, I would expect DTLS to go to WGLC on London. Note that due to the lower level of inspection of DTLS and the preexisting need for hole-punching (e.g., WebRTC), the last minute middlebox interop issues that delayed TLS 1.3 should not be an issue for DTLS.

However, conveniently this status allows us to get the small changes described in [Section 3](#) in before we close the document. The TLS WG is very interested in coordinating with QUIC – and has many of the same key players – so it should be possible to close these quickly and proceed to DTLS 1.3 WGLC.

In general, DTLS 1.2 is reasonably widely implemented, though not as widely implemented as TLS. All the major browser stacks already have it to support WebRTC. DTLS 1.3 is less widely supported: NSS has an implementation and I am aware of several in progress implementations. Presumably, stacks will need to update to DTLS 1.3 in order to support WebRTC as well. Having done the DTLS implementations for NSS and Mingo, I can report that it's not a huge amount of work once you have TLS 1.3 support.

One important note here is that unlike the current design, which is very tightly coupled to TLS 1.3, the design proposed here does not depend on DTLS version. Thus, it is possible to have interop with two stacks as long as they support a common version, including DTLS

1.2, though of course 0-RTT will not be available unless you have TLS 1.3

[6.3.](#) Epoch and ACKs

DTLS record sequence numbers consist of a 16-bit epoch and 48-bit sequence number pair, with sequence numbers restarting at zero after each epoch change. This would put some stress on the ACK format, but it's easily addressed by having a separate set of ACK blocks for each epoch.

[6.4.](#) Crypto Layer Agility

One concern I've heard raised is how this affects the ability to swap out the crypto layer. There are pluses and minuses here. The architecture described here is a better fit for swapping in a conventional channel security protocol (e.g., IPsec, etc.) because the layering is very conventional. It is also probably better for swapping in a protocol which has totally out of band key management where you just carry a session identifier in the packets.

It is less good for swapping in a simple crypto core (e.g., OPTLS), because those cores don't have their own framing, reliability layers. It's not that onerous to invent one, but you do have to do that. On the other hand, those protocols often also don't have other protocol engineering pieces (e.g., cipher suite and curve negotiation, so they are hard to extend). Another possibility in that case is to move the crypto core into DTLS. For instance we have been looking at adding an OPTLS-style semi-static mode to D/TLS. This has the advantages that you don't need to reinvent the protocol engineering pieces of TLS while giving you crypto core flexibility. See [[I-D.putman-tls13-preshared-dh](#)] for an example here.

[7.](#) WebRTC Binding

There has been some initial discussion about doing WebRTC with QUIC. This actually slots in quite naturally with a DTLS design. There are two primary structures for WebRTC with QUIC:

- o Replace SCTP with QUIC but continue to carry media over RTP
- o Replace everything with QUIC

The former is a trivial change: you just do a DTLS handshake but signal (in ALPN or the SDP) that you are doing QUIC instead of SCTP, and carry QUIC over DTLS as expected, using the same DTLS Exporter to generate the SRTP keys.

In the second case, you just carry QUIC over DTLS, with whatever media over QUIC binding we invent, and then don't do a DTLS exporter.

[8.](#) Implementation Status

I have implemented the core of this proposal in Minq based on the in-progress implementation of DTLS 1.3 for Mint. Handshakes succeed and I can exchange data. I have not yet implemented 0-RTT or resumption (though Mint supports both) and I expect them to be straightforward. The one thing that seems like it might be slightly is exposing the DTLS packet numbers to Minq, but that seems like it's just plumbing. The whole effort took me less than 12 hours and resulted in a net shrinkage of Minq by about 10% (700 lines of code) just from code which was obviously unneeded and in the way, without any attempt to do a real scrub.

[9.](#) Security Considerations

No doubt plenty.

[10.](#) References

[10.1.](#) Normative References

[I-D.ietf-quic-tls]

Thomson, M. and S. Turner, "Using Transport Layer Security (TLS) to Secure QUIC", [draft-ietf-quic-tls-10](#) (work in progress), March 2018.

[I-D.ietf-quic-transport]

Iyengar, J. and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport", [draft-ietf-quic-transport-10](#) (work in progress), March 2018.

[I-D.ietf-tls-dtls-connection-id]

Rescorla, E., Tschofenig, H., Fossati, T., and T. Gondrom, "The Datagram Transport Layer Security (DTLS) Connection Identifier", [draft-ietf-tls-dtls-connection-id-00](#) (work in progress), December 2017.

[I-D.ietf-tls-dtls13]

Rescorla, E., Tschofenig, H., and N. Modadugu, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3", [draft-ietf-tls-dtls13-26](#) (work in progress), March 2018.

Internet-Draft

QUIC/DTLS

March 2018

[I-D.ietf-tls-tls13]

Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", [draft-ietf-tls-tls13-26](#) (work in progress), March 2018.

[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>>.

[10.2](#). Informative References

[I-D.putman-tls13-preshared-dh]

Putman, T., "Authenticated Key Agreement using Pre-Shared Asymmetric Keypairs for (Datagram) Transport Layer Security ((D)TLS) Protocol version 1.3", [draft-putman-tls13-preshared-dh-00](#) (work in progress), January 2018.

Author's Address

Eric Rescorla
RTFM, Inc.

Email: ekr@rtfm.com

Rescorla

Expires September 6, 2018

[Page 17]