Stream:	Internet Engineering Task Force (IETF)
RFC:	9147
Obsoletes:	6347
Category:	Standards Track
Published:	April 2022
ISSN:	2070-1721
Authors:	E. Rescorla	H. Tschofenig	N. Modadugu
---	---	---	---
	Mozilla	*Arm Limited*	*Google, Inc.*

RFC 9147
The Datagram Transport Layer Security (DTLS) Protocol Version 1.3

Abstract

This document specifies version 1.3 of the Datagram Transport Layer Security (DTLS) protocol. DTLS 1.3 allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.

The DTLS 1.3 protocol is based on the Transport Layer Security (TLS) 1.3 protocol and provides equivalent security guarantees with the exception of order protection / non-replayability. Datagram semantics of the underlying transport are preserved by the DTLS protocol.

This document obsoletes RFC 6347.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc9147.

Table of Contents

1. Introduction 4
2. Conventions and Terminology 5
3. DTLS Design Rationale and Overview 6
   3.1. Packet Loss 6
   3.2. Reordering 7
   3.3. Fragmentation 7
   3.4. Replay Detection 8
4. The DTLS Record Layer 8
   4.1. Demultiplexing DTLS Records 12
   4.2. Sequence Number and Epoch 13
      4.2.1. Processing Guidelines 13
      4.2.2. Reconstructing the Sequence Number and Epoch 14
      4.2.3. Record Number Encryption 14
   4.3. Transport Layer Mapping 15
   4.4. PMTU Issues 16
   4.5. Record Payload Protection 17
      4.5.1. Anti-Replay 17
      4.5.2. Handling Invalid Records 18

4.5.3. AEAD Limits 18
5. The DTLS Handshake Protocol 19
5.1. Denial-of-Service Countermeasures 20
5.2. DTLS Handshake Message Format 22
5.3. ClientHello Message 24
5.4. ServerHello Message 25
5.5. Handshake Message Fragmentation and Reassembly 25
5.6. EndOfEarlyData Message 26
5.7. DTLS Handshake Flights 26
5.8. Timeout and Retransmission 29
5.8.1. State Machine 29
5.8.2. Timer Values 32
5.8.3. Large Flight Sizes 32
5.8.4. State Machine Duplication for Post-Handshake Messages 33
5.9. Cryptographic Label Prefix 33
5.10. Alert Messages 34
5.11. Establishing New Associations with Existing Parameters 34
6. Example of Handshake with Timeout and Retransmission 34
6.1. Epoch Values and Rekeying 36
7. ACK Message 38
7.1. Sending ACKs 39
7.2. Receiving ACKs 40
7.3. Design Rationale 40
8. Key Updates 41
9. Connection ID Updates 43
9.1. Connection ID Example 44
10. Application Data Protocol 45
11. Security Considerations 46
12. Changes since DTLS 1.2 48
13. Updates Affecting DTLS 1.2 48

14. IANA Considerations 48
15. References 49
15.1. Normative References 49
15.2. Informative References 50
Appendix A. Protocol Data Structures and Constant Values 52
A.1. Record Layer 53
A.2. Handshake Protocol 53
A.3. ACKs 55
A.4. Connection ID Management 55
Appendix B. Analysis of Limits on CCM Usage 55
B.1. Confidentiality Limits 56
B.2. Integrity Limits 56
B.3. Limits for AEAD_AES_128_CCM_8 57
Appendix C. Implementation Pitfalls 57
Contributors 58
Authors' Addresses 59

1. Introduction

The primary goal of the TLS protocol is to establish an authenticated, confidentiality- and integrity-protected channel between two communicating peers. The TLS protocol is composed of two layers: the TLS record protocol and the TLS handshake protocol. However, TLS must run over a reliable transport channel -- typically TCP [RFC0793].

There are applications that use UDP [RFC0768] as a transport and the Datagram Transport Layer Security (DTLS) protocol has been developed to offer communication security protection for those applications. DTLS is deliberately designed to be as similar to TLS as possible, both to minimize new security invention and to maximize the amount of code and infrastructure reuse.

DTLS 1.0 [RFC4347] was originally defined as a delta from TLS 1.1 [RFC4346], and DTLS 1.2 [RFC6347] was defined as a series of deltas to TLS 1.2 [RFC5246]. There is no DTLS 1.1; that version number was skipped in order to harmonize version numbers with TLS. This specification describes the most current version of the DTLS protocol as a delta from TLS 1.3 [TLS13]. It obsoletes DTLS 1.2.

Implementations that speak both DTLS 1.2 and DTLS 1.3 can interoperate with those that speak only DTLS 1.2 (using DTLS 1.2 of course), just as TLS 1.3 implementations can interoperate with TLS 1.2 (see Appendix D of [TLS13] for details). While backwards compatibility with DTLS 1.0 is possible, the use of DTLS 1.0 is not recommended, as explained in Section 3.1.2 of [RFC7525]. [DEPRECATE] forbids the use of DTLS 1.0.

2. Conventions and Terminology

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.

The following terms are used:

client: The endpoint initiating the DTLS connection.

association: Shared state between two endpoints established with a DTLS handshake.

connection: Synonym for association.

endpoint: Either the client or server of the connection.

epoch: One set of cryptographic keys used for encryption and decryption.

handshake: An initial negotiation between client and server that establishes the parameters of the connection.

peer: An endpoint. When discussing a particular endpoint, "peer" refers to the endpoint that is remote to the primary subject of discussion.

receiver: An endpoint that is receiving records.

sender: An endpoint that is transmitting records.

server: The endpoint that did not initiate the DTLS connection.

CID: Connection ID.

MSL: Maximum Segment Lifetime.

The reader is assumed to be familiar with [TLS13]. As in TLS 1.3, the HelloRetryRequest has the same format as a ServerHello message, but for convenience we use the term HelloRetryRequest throughout this document as if it were a distinct message.

DTLS 1.3 uses network byte order (big-endian) format for encoding messages based on the encoding format defined in [TLS13] and earlier (D)TLS specifications.

The reader is also assumed to be familiar with [RFC9146], as this document applies the CID functionality to DTLS 1.3.

Figures in this document illustrate various combinations of the DTLS protocol exchanges, and the symbols have the following meaning:

'+' indicates noteworthy extensions sent in the previously noted message.

'*' indicates optional or situation-dependent messages/extensions that are not always sent.

'{}' indicates messages protected using keys derived from a [sender]_handshake_traffic_secret.

'[]' indicates messages protected using keys derived from traffic_secret_N.

3. DTLS Design Rationale and Overview

The basic design philosophy of DTLS is to construct "TLS over datagram transport". Datagram transport neither requires nor provides reliable or in-order delivery of data. The DTLS protocol preserves this property for application data. Applications such as media streaming, Internet telephony, and online gaming use datagram transport for communication due to the delay-sensitive nature of transported data. The behavior of such applications is unchanged when the DTLS protocol is used to secure communication, since the DTLS protocol does not compensate for lost or reordered data traffic. Note that while low-latency streaming and gaming use DTLS to protect data (e.g., for protection of a WebRTC data channel), telephony utilizes DTLS for key establishment and the Secure Real-time Transport Protocol (SRTP) for protection of data [RFC5763].

TLS cannot be used directly over datagram transports for the following four reasons:

1. TLS relies on an implicit sequence number on records. If a record is not received, then the recipient will use the wrong sequence number when attempting to remove record protection from subsequent records. DTLS solves this problem by adding sequence numbers to records.
2. The TLS handshake is a lock-step cryptographic protocol. Messages must be transmitted and received in a defined order; any other order is an error. The DTLS handshake includes message sequence numbers to enable fragmented message reassembly and in-order delivery in case datagrams are lost or reordered.
3. Handshake messages are potentially larger than can be contained in a single datagram. DTLS adds fields to handshake messages to support fragmentation and reassembly.
4. Datagram transport protocols are susceptible to abusive behavior effecting denial-of-service (DoS) attacks against nonparticipants. DTLS adds a return-routability check and DTLS 1.3 uses the TLS 1.3 HelloRetryRequest message (see Section 5.1 for details).

3.1. Packet Loss

DTLS uses a simple retransmission timer to handle packet loss. Figure 1 demonstrates the basic concept, using the first phase of the DTLS handshake:

Figure 1: DTLS Retransmission Example

Once the client has transmitted the ClientHello message, it expects to see a HelloRetryRequest or a ServerHello from the server. However, if the timer expires, the client knows that either the ClientHello or the response from the server has been lost, which causes the client to retransmit the ClientHello. When the server receives the retransmission, it knows to retransmit its HelloRetryRequest or ServerHello.

The server also maintains a retransmission timer for messages it sends (other than HelloRetryRequest) and retransmits when that timer expires. Not applying retransmissions to the HelloRetryRequest avoids the need to create state on the server. The HelloRetryRequest is designed to be small enough that it will not itself be fragmented, thus avoiding concerns about interleaving multiple HelloRetryRequests.

For more detail on timeouts and retransmission, see Section 5.8.

3.2. Reordering

In DTLS, each handshake message is assigned a specific sequence number. When a peer receives a handshake message, it can quickly determine whether that message is the next message it expects. If it is, then it processes it. If not, it queues it for future handling once all previous messages have been received.

3.3. Fragmentation

TLS and DTLS handshake messages can be quite large (in theory up to 2^24-1 bytes, in practice many kilobytes). By contrast, UDP datagrams are often limited to less than 1500 bytes if IP fragmentation is not desired. In order to compensate for this limitation, each DTLS handshake message may be fragmented over several DTLS records, each of which is intended to fit in a single UDP datagram (see Section 4.4 for guidance). Each DTLS handshake message contains both a fragment offset and a fragment length. Thus, a recipient in possession of all bytes of a handshake message can reassemble the original unfragmented message.

3.4. Replay Detection

DTLS optionally supports record replay detection. The technique used is the same as in IPsec AH/ESP, by maintaining a bitmap window of received records. Records that are too old to fit in the window and records that have previously been received are silently discarded. The replay detection feature is optional, since packet duplication is not always malicious but can also occur due to routing errors. Applications may conceivably detect duplicate packets and accordingly modify their data transmission strategy.

4. The DTLS Record Layer

The DTLS 1.3 record layer is different from the TLS 1.3 record layer and also different from the DTLS 1.2 record layer.

1. The DTLSCiphertext structure omits the superfluous version number and type fields.
2. DTLS adds an epoch and sequence number to the TLS record header. This sequence number allows the recipient to correctly decrypt and verify DTLS records. However, the number of bits used for the epoch and sequence number fields in the DTLSCiphertext structure has been reduced from those in previous versions.
3. The DTLS epoch serialized in DTLSPlaintext is 2 octets long for compatibility with DTLS 1.2. However, this value is set as the least significant 2 octets of the connection epoch, which is an 8 octet counter incremented on every KeyUpdate. See Section 4.2 for details. The sequence number is set to be the low order 48 bits of the 64 bit sequence number. Plaintext records **MUST NOT** be sent with sequence numbers that would exceed 2^48-1, so the upper 16 bits will always be 0.
4. The DTLSCiphertext structure has a variable-length header.

DTLSPlaintext records are used to send unprotected records and DTLSCiphertext records are used to send protected records.

The DTLS record formats are shown below. Unless explicitly stated the meaning of the fields is unchanged from previous TLS/DTLS versions.

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

struct {
     opaque content[DTLSPlaintext.length];
     ContentType type;
     uint8 zeros[length_of_padding];
} DTLSInnerPlaintext;

struct {
    opaque unified_hdr[variable];
    opaque encrypted_record[length];
} DTLSCiphertext;
```

Figure 2: DTLS 1.3 Record Formats

legacy_record_version: This value **MUST** be set to {254, 253} for all records other than the initial ClientHello (i.e., one not generated after a HelloRetryRequest), where it may also be {254, 255} for compatibility purposes. It **MUST** be ignored for all purposes. See [TLS13], Appendix D.1 for the rationale for this.

epoch: The least significant 2 bytes of the connection epoch value.

unified_hdr: The unified header (unified_hdr) is a structure of variable length, shown in Figure 3.

encrypted_record: The encrypted form of the serialized DTLSInnerPlaintext structure.

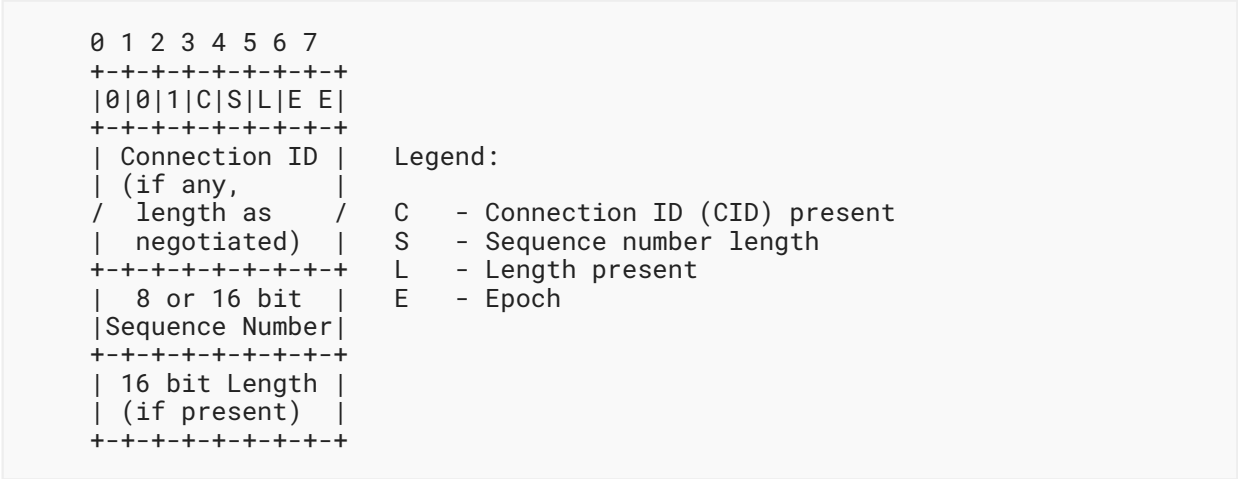


Figure 3: DTLS 1.3 Unified Header

Fixed Bits: The three high bits of the first byte of the unified header are set to 001. This ensures that the value will fit within the DTLS region when multiplexing is performed as described in [RFC7983]. It also ensures that distinguishing encrypted DTLS 1.3 records from encrypted DTLS 1.2 records is possible when they are carried on the same host/port quartet; such multiplexing is only possible when CIDs [RFC9146] are in use, in which case DTLS 1.2 records will have the content type tls12_cid (25).

C: The C bit (0x10) is set if the Connection ID is present.

S: The S bit (0x08) indicates the size of the sequence number. 0 means an 8-bit sequence number, 1 means 16-bit. Implementations **MAY** mix sequence numbers of different lengths on the same connection.

L: The L bit (0x04) is set if the length is present.

E: The two low bits (0x03) include the low-order two bits of the epoch.

Connection ID: Variable-length CID. The CID functionality is described in [RFC9146]. An example can be found in Section 9.1.

Sequence Number: The low-order 8 or 16 bits of the record sequence number. This value is 16 bits if the S bit is set to 1, and 8 bits if the S bit is 0.

Length: Identical to the length field in a TLS 1.3 record.

As with previous versions of DTLS, multiple DTLSPlaintext and DTLSCiphertext records can be included in the same underlying transport datagram.

Figure 4 illustrates different record headers.

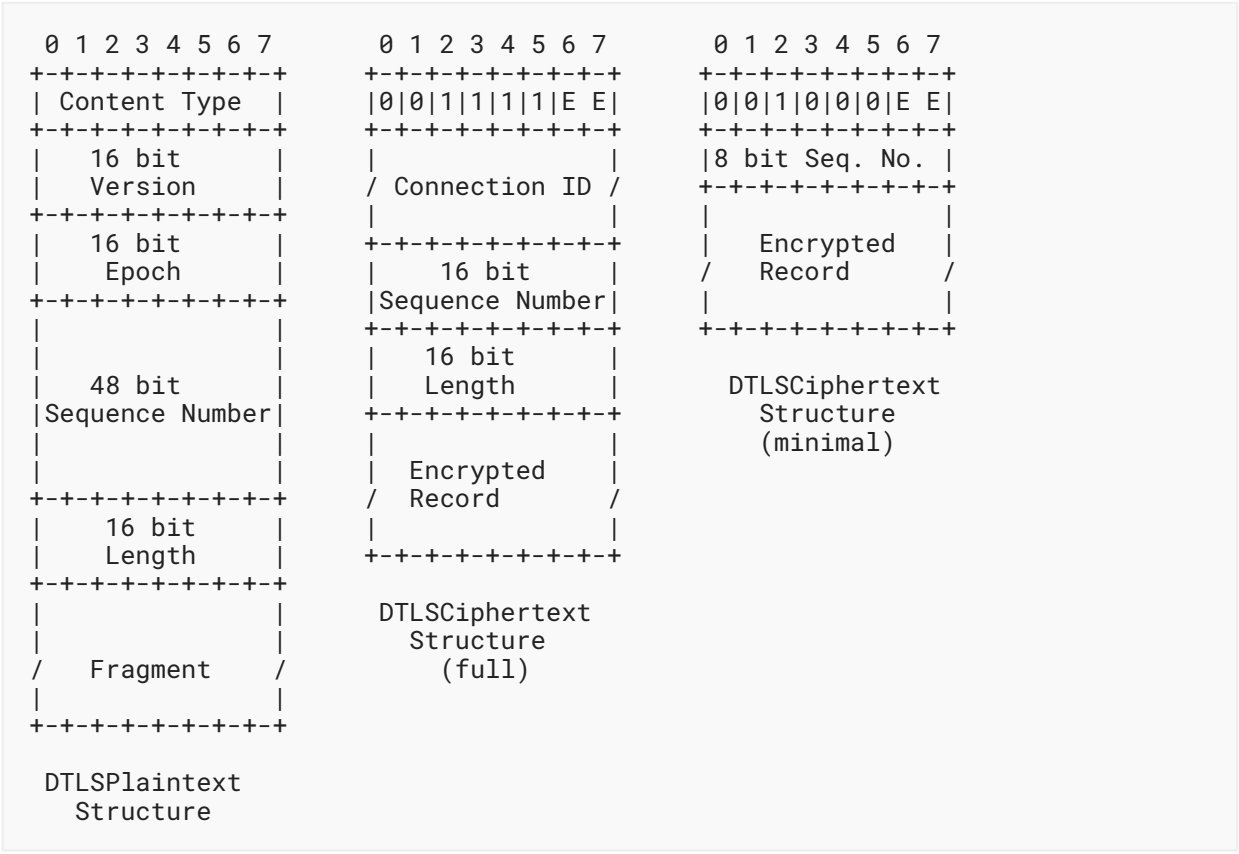


Figure 4: DTLS 1.3 Header Examples

The length field **MAY** be omitted by clearing the L bit, which means that the record consumes the entire rest of the datagram in the lower level transport. In this case, it is not possible to have multiple DTLSCiphertext format records without length fields in the same datagram. Omitting the length field **MUST** only be used for the last record in a datagram. Implementations **MAY** mix records with and without length fields on the same connection.

If a Connection ID is negotiated, then it **MUST** be contained in all datagrams. Sending implementations **MUST NOT** mix records from multiple DTLS associations in the same datagram. If the second or later record has a connection ID which does not correspond to the same association used for previous records, the rest of the datagram **MUST** be discarded.

When expanded, the epoch and sequence number can be combined into an unpacked RecordNumber structure, as shown below:

```
struct {
    uint64 epoch;
    uint64 sequence_number;
} RecordNumber;
```

This 128-bit value is used in the ACK message as well as in the "record_sequence_number" input to the Authenticated Encryption with Associated Data (AEAD) function. The entire header value shown in Figure 4 (but prior to record number encryption; see Section 4.2.3) is used as the additional data value for the AEAD function. For instance, if the minimal variant is used, the Associated Data (AD) is 2 octets long. Note that this design is different from the additional data calculation for DTLS 1.2 and for DTLS 1.2 with Connection IDs. In DTLS 1.3 the 64-bit sequence_number is used as the sequence number for the AEAD computation; unlike DTLS 1.2, the epoch is not included.

4.1. Demultiplexing DTLS Records

DTLS 1.3's header format is more complicated to demux than DTLS 1.2, which always carried the content type as the first byte. As described in Figure 5, the first byte determines how an incoming DTLS record is demultiplexed. The first 3 bits of the first byte distinguish a DTLS 1.3 encrypted record from record types used in previous DTLS versions and plaintext DTLS 1.3 record types. Hence, the range 32 (0b0010 0000) to 63 (0b0011 1111) needs to be excluded from future allocations by IANA to avoid problems while demultiplexing; see Section 14. Implementations can demultiplex DTLS 1.3 records by examining the first byte as follows:

- If the first byte is alert(21), handshake(22), or ack(proposed, 26), the record **MUST** be interpreted as a DTLSPlaintext record.
- If the first byte is any other value, then receivers **MUST** check to see if the leading bits of the first byte are 001. If so, the implementation **MUST** process the record as DTLSCiphertext; the true content type will be inside the protected portion.
- Otherwise, the record **MUST** be rejected as if it had failed deprotection, as described in Section 4.5.2.

Figure 5 shows this demultiplexing procedure graphically, taking DTLS 1.3 and earlier versions of DTLS into account.

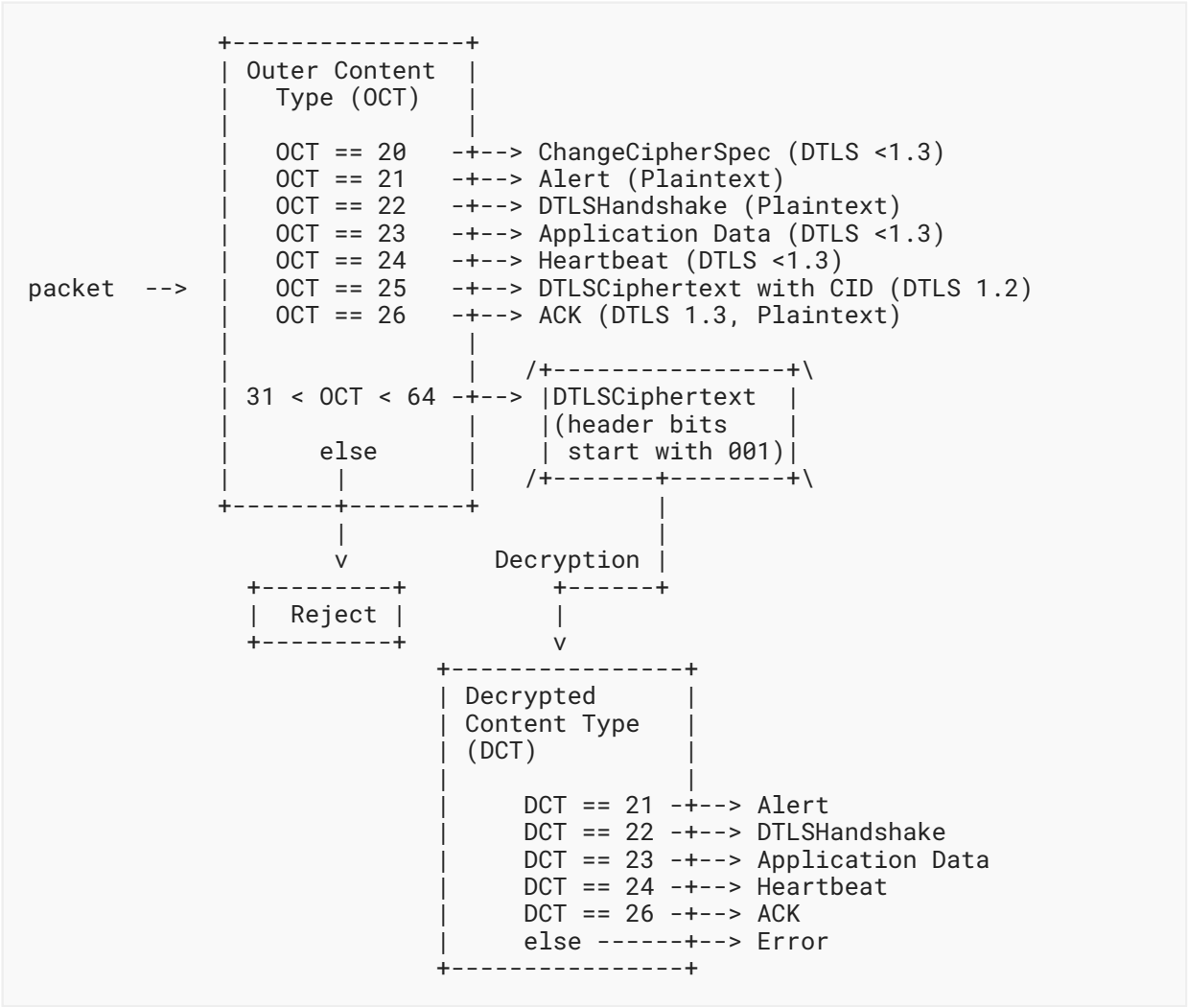


Figure 5: Demultiplexing DTLS 1.2 and DTLS 1.3 Records

4.2. Sequence Number and Epoch

DTLS uses an explicit or partly explicit sequence number, rather than an implicit one, carried in the sequence_number field of the record. Sequence numbers are maintained separately for each epoch, with each sequence_number initially being 0 for each epoch.

The epoch number is initially zero and is incremented each time keying material changes and a sender aims to rekey. More details are provided in Section 6.1.

4.2.1. Processing Guidelines

Because DTLS records could be reordered, a record from epoch M may be received after epoch N (where N > M) has begun. Implementations **SHOULD** discard records from earlier epochs but **MAY** choose to retain keying material from previous epochs for up to the default MSL specified for

TCP [RFC0793] to allow for packet reordering. (Note that the intention here is that implementers use the current guidance from the IETF for MSL, as specified in [RFC0793] or successors, not that they attempt to interrogate the MSL that the system TCP stack is using.)

Conversely, it is possible for records that are protected with the new epoch to be received prior to the completion of a handshake. For instance, the server may send its Finished message and then start transmitting data. Implementations **MAY** either buffer or discard such records, though when DTLS is used over reliable transports (e.g., SCTP [RFC4960]), they **SHOULD** be buffered and processed once the handshake completes. Note that TLS's restrictions on when records may be sent still apply, and the receiver treats the records as if they were sent in the right order.

Implementations **MUST** send retransmissions of lost messages using the same epoch and keying material as the original transmission.

Implementations **MUST** either abandon an association or rekey prior to allowing the sequence number to wrap.

Implementations **MUST NOT** allow the epoch to wrap, but instead **MUST** establish a new association, terminating the old association.

4.2.2. Reconstructing the Sequence Number and Epoch

When receiving protected DTLS records, the recipient does not have a full epoch or sequence number value in the record and so there is some opportunity for ambiguity. Because the full sequence number is used to compute the per-record nonce and the epoch determines the keys, failure to reconstruct these values leads to failure to deprotect the record, and so implementations **MAY** use a mechanism of their choice to determine the full values. This section provides an algorithm which is comparatively simple and which implementations are **RECOMMENDED** to follow.

If the epoch bits match those of the current epoch, then implementations **SHOULD** reconstruct the sequence number by computing the full sequence number which is numerically closest to one plus the sequence number of the highest successfully deprotected record in the current epoch.

During the handshake phase, the epoch bits unambiguously indicate the correct key to use. After the handshake is complete, if the epoch bits do not match those from the current epoch, implementations **SHOULD** use the most recent past epoch which has matching bits, and then reconstruct the sequence number for that epoch as described above.

4.2.3. Record Number Encryption

In DTLS 1.3, when records are encrypted, record sequence numbers are also encrypted. The basic pattern is that the underlying encryption algorithm used with the AEAD algorithm is used to generate a mask which is then XORed with the sequence number.

When the AEAD is based on AES, then the mask is generated by computing AES-ECB on the first 16 bytes of the ciphertext:

```
Mask = AES-ECB(sn_key, Ciphertext[0..15])
```

When the AEAD is based on ChaCha20, then the mask is generated by treating the first 4 bytes of the ciphertext as the block counter and the next 12 bytes as the nonce, passing them to the ChaCha20 block function (Section 2.3 of [CHACHA]):

```
Mask = ChaCha20(sn_key, Ciphertext[0..3], Ciphertext[4..15])
```

The sn_key is computed as follows:

```
[sender]_sn_key = HKDF-Expand-Label(Secret, "sn", "", key_length)
```

[sender] denotes the sending side. The per-epoch Secret value to be used is described in Section 7.3 of [TLS13]. Note that a new key is used for each epoch: because the epoch is sent in the clear, this does not result in ambiguity.

The encrypted sequence number is computed by XORing the leading bytes of the mask with the on-the-wire representation of the sequence number. Decryption is accomplished by the same process.

This procedure requires the ciphertext length to be at least 16 bytes. Receivers **MUST** reject shorter records as if they had failed deprotection, as described in Section 4.5.2. Senders **MUST** pad short plaintexts out (using the conventional record padding mechanism) in order to make a suitable-length ciphertext. Note that most of the DTLS AEAD algorithms have a 16 byte authentication tag and need no padding. However, some algorithms, such as TLS_AES_128_CCM_8_SHA256, have a shorter authentication tag and may require padding for short inputs.

Future cipher suites, which are not based on AES or ChaCha20, **MUST** define their own record sequence number encryption in order to be used with DTLS.

Note that sequence number encryption is only applied to the DTLSCiphertext structure and not to the DTLSPlaintext structure, even though it also contains a sequence number.

4.3. Transport Layer Mapping

DTLS messages **MAY** be fragmented into multiple DTLS records. Each DTLS record **MUST** fit within a single datagram. In order to avoid IP fragmentation, clients of the DTLS record layer **SHOULD** attempt to size records so that they fit within any Path MTU (PMTU) estimates obtained from the record layer. For more information about PMTU issues, see Section 4.4.

Multiple DTLS records **MAY** be placed in a single datagram. Records are encoded consecutively. The length field from DTLS records containing that field can be used to determine the boundaries between records. The final record in a datagram can omit the length field. The first byte of the datagram payload **MUST** be the beginning of a record. Records **MUST NOT** span datagrams.

DTLS records without CIDs do not contain any association identifiers, and applications must arrange to multiplex between associations. With UDP, the host/port number is used to look up the appropriate security association for incoming records without CIDs.

Some transports, such as DCCP [RFC4340], provide their own sequence numbers. When carried over those transports, both the DTLS and the transport sequence numbers will be present. Although this introduces a small amount of inefficiency, the transport layer and DTLS sequence numbers serve different purposes; therefore, for conceptual simplicity, it is superior to use both sequence numbers.

Some transports provide congestion control for traffic carried over them. If the congestion window is sufficiently narrow, DTLS handshake retransmissions may be held rather than transmitted immediately, potentially leading to timeouts and spurious retransmission. When DTLS is used over such transports, care should be taken not to overrun the likely congestion window. [RFC5238] defines a mapping of DTLS to DCCP that takes these issues into account.

4.4. PMTU Issues

In general, DTLS's philosophy is to leave PMTU discovery to the application. However, DTLS cannot completely ignore the PMTU for three reasons:

- The DTLS record framing expands the datagram size, thus lowering the effective PMTU from the application's perspective.
- In some implementations, the application may not directly talk to the network, in which case the DTLS stack may absorb ICMP "Datagram Too Big" indications [RFC1191] or ICMPv6 "Packet Too Big" indications [RFC4443].
- The DTLS handshake messages can exceed the PMTU.

In order to deal with the first two issues, the DTLS record layer **SHOULD** behave as described below.

If PMTU estimates are available from the underlying transport protocol, they should be made available to upper layer protocols. In particular:

- For DTLS over UDP, the upper layer protocol **SHOULD** be allowed to obtain the PMTU estimate maintained in the IP layer.
- For DTLS over DCCP, the upper layer protocol **SHOULD** be allowed to obtain the current estimate of the PMTU.
- For DTLS over TCP or SCTP, which automatically fragment and reassemble datagrams, there is no PMTU limitation. However, the upper layer protocol **MUST NOT** write any record that exceeds the maximum record size of 2^14 bytes.

The DTLS record layer **SHOULD** also allow the upper layer protocol to discover the amount of record expansion expected by the DTLS processing; alternately, it **MAY** report PMTU estimates minus the estimated expansion from the transport layer and DTLS record framing.

Note that DTLS does not defend against spoofed ICMP messages; implementations **SHOULD** ignore any such messages that indicate PMTUs below the IPv4 and IPv6 minimums of 576 and 1280 bytes, respectively.

If there is a transport protocol indication that the PMTU was exceeded (either via ICMP or via a refusal to send the datagram as in Section 14 of [RFC4340]), then the DTLS record layer **MUST** inform the upper layer protocol of the error.

The DTLS record layer **SHOULD NOT** interfere with upper layer protocols performing PMTU discovery, whether via [RFC1191] and [RFC4821] for IPv4 or via [RFC8201] for IPv6. In particular:

- Where allowed by the underlying transport protocol, the upper layer protocol **SHOULD** be allowed to set the state of the Don't Fragment (DF) bit (in IPv4) or prohibit local fragmentation (in IPv6).
- If the underlying transport protocol allows the application to request PMTU probing (e.g., DCCP), the DTLS record layer **SHOULD** honor this request.

The final issue is the DTLS handshake protocol. From the perspective of the DTLS record layer, this is merely another upper layer protocol. However, DTLS handshakes occur infrequently and involve only a few round trips; therefore, the handshake protocol PMTU handling places a premium on rapid completion over accurate PMTU discovery. In order to allow connections under these circumstances, DTLS implementations **SHOULD** follow the following rules:

- If the DTLS record layer informs the DTLS handshake layer that a message is too big, the handshake layer **SHOULD** immediately attempt to fragment the message, using any existing information about the PMTU.
- If repeated retransmissions do not result in a response, and the PMTU is unknown, subsequent retransmissions **SHOULD** back off to a smaller record size, fragmenting the handshake message as appropriate. This specification does not specify an exact number of retransmits to attempt before backing off, but 2-3 seems appropriate.

4.5. Record Payload Protection

Like TLS, DTLS transmits data as a series of protected records. The rest of this section describes the details of that format.

4.5.1. Anti-Replay

Each DTLS record contains a sequence number to provide replay protection. Sequence number verification **SHOULD** be performed using the following sliding window procedure, borrowed from Section 3.4.3 of [RFC4303]. Because each epoch resets the sequence number space, a separate sliding window is needed for each epoch.

The received record counter for an epoch **MUST** be initialized to zero when that epoch is first used. For each received record, the receiver **MUST** verify that the record contains a sequence number that does not duplicate the sequence number of any other record received in that epoch during the lifetime of the association. This check **SHOULD** happen after deprotecting the record;

otherwise, the record discard might itself serve as a timing channel for the record number. Note that computing the full record number from the partial is still a potential timing channel for the record number, though a less powerful one than whether the record was deprotected.

Duplicates are rejected through the use of a sliding receive window. (How the window is implemented is a local matter, but the following text describes the functionality that the implementation must exhibit.) The receiver **SHOULD** pick a window large enough to handle any plausible reordering, which depends on the data rate. (The receiver does not notify the sender of the window size.)

The "right" edge of the window represents the highest validated sequence number value received in the epoch. Records that contain sequence numbers lower than the "left" edge of the window are rejected. Records falling within the window are checked against a list of received records within the window. An efficient means for performing this check, based on the use of a bit mask, is described in Section 3.4.3 of [RFC4303]. If the received record falls within the window and is new, or if the record is to the right of the window, then the record is new.

The window **MUST NOT** be updated due to a received record until that record has been deprotected successfully.

4.5.2. Handling Invalid Records

Unlike TLS, DTLS is resilient in the face of invalid records (e.g., invalid formatting, length, MAC, etc.). In general, invalid records **SHOULD** be silently discarded, thus preserving the association; however, an error **MAY** be logged for diagnostic purposes. Implementations which choose to generate an alert instead **MUST** generate fatal alerts to avoid attacks where the attacker repeatedly probes the implementation to see how it responds to various types of error. Note that if DTLS is run over UDP, then any implementation which does this will be extremely susceptible to DoS attacks because UDP forgery is so easy. Thus, generating fatal alerts is **NOT RECOMMENDED** for such transports, both to increase the reliability of DTLS service and to avoid the risk of spoofing attacks sending traffic to unrelated third parties.

If DTLS is being carried over a transport that is resistant to forgery (e.g., SCTP with SCTP-AUTH), then it is safer to send alerts because an attacker will have difficulty forging a datagram that will not be rejected by the transport layer.

Note that because invalid records are rejected at a layer lower than the handshake state machine, they do not affect pending retransmission timers.

4.5.3. AEAD Limits

Section 5.5 of [TLS13] defines limits on the number of records that can be protected using the same keys. These limits are specific to an AEAD algorithm and apply equally to DTLS. Implementations **SHOULD NOT** protect more records than allowed by the limit specified for the negotiated AEAD. Implementations **SHOULD** initiate a key update before reaching this limit.

[TLS13] does not specify a limit for AEAD_AES_128_CCM, but the analysis in Appendix B shows that a limit of 2^23 packets can be used to obtain the same confidentiality protection as the limits specified in TLS.

The usage limits defined in TLS 1.3 exist for protection against attacks on confidentiality and apply to successful applications of AEAD protection. The integrity protections in authenticated encryption also depend on limiting the number of attempts to forge packets. TLS achieves this by closing connections after any record fails an authentication check. In comparison, DTLS ignores any packet that cannot be authenticated, allowing multiple forgery attempts.

Implementations **MUST** count the number of received packets that fail authentication with each key. If the number of packets that fail authentication exceeds a limit that is specific to the AEAD in use, an implementation **SHOULD** immediately close the connection. Implementations **SHOULD** initiate a key update with update_requested before reaching this limit. Once a key update has been initiated, the previous keys can be dropped when the limit is reached rather than closing the connection. Applying a limit reduces the probability that an attacker is able to successfully forge a packet; see [AEBounds] and [ROBUST].

For AEAD_AES_128_GCM, AEAD_AES_256_GCM, and AEAD_CHACHA20_POLY1305, the limit on the number of records that fail authentication is 2^36. Note that the analysis in [AEBounds] supports a higher limit for AEAD_AES_128_GCM and AEAD_AES_256_GCM, but this specification recommends a lower limit. For AEAD_AES_128_CCM, the limit on the number of records that fail authentication is 2^23.5; see Appendix B.

The AEAD_AES_128_CCM_8 AEAD, as used in TLS_AES_128_CCM_8_SHA256, does not have a limit on the number of records that fail authentication that both limits the probability of forgery by the same amount and does not expose implementations to the risk of denial of service; see Appendix B.3. Therefore, TLS_AES_128_CCM_8_SHA256 **MUST NOT** be used in DTLS without additional safeguards against forgery. Implementations **MUST** set usage limits for AEAD_AES_128_CCM_8 based on an understanding of any additional forgery protections that are used.

Any TLS cipher suite that is specified for use with DTLS **MUST** define limits on the use of the associated AEAD function that preserves margins for both confidentiality and integrity. That is, limits **MUST** be specified for the number of packets that can be authenticated and for the number of packets that can fail authentication before a key update is required. Providing a reference to any analysis upon which values are based -- and any assumptions used in that analysis -- allows limits to be adapted to varying usage conditions.

5. The DTLS Handshake Protocol

DTLS 1.3 reuses the TLS 1.3 handshake messages and flows, with the following changes:

1. To handle message loss, reordering, and fragmentation, modifications to the handshake header are necessary.
2. Retransmission timers are introduced to handle message loss.
3. A new ACK content type has been added for reliable message delivery of handshake messages.

In addition, DTLS reuses TLS 1.3's "cookie" extension to provide a return-routability check as part of connection establishment. This is an important DoS prevention mechanism for UDP-based protocols, unlike TCP-based protocols, for which TCP establishes return-routability as part of the connection establishment.

DTLS implementations do not use the TLS 1.3 "compatibility mode" described in Appendix D.4 of [TLS13]. DTLS servers **MUST NOT** echo the "legacy_session_id" value from the client and endpoints **MUST NOT** send ChangeCipherSpec messages.

With these exceptions, the DTLS message formats, flows, and logic are the same as those of TLS 1.3.

5.1. Denial-of-Service Countermeasures

Datagram security protocols are extremely susceptible to a variety of DoS attacks. Two attacks are of particular concern:

1. An attacker can consume excessive resources on the server by transmitting a series of handshake initiation requests, causing the server to allocate state and potentially to perform expensive cryptographic operations.
2. An attacker can use the server as an amplifier by sending connection initiation messages with a forged source address that belongs to a victim. The server then sends its response to the victim machine, thus flooding it. Depending on the selected parameters, this response message can be quite large, as is the case for a Certificate message.

In order to counter both of these attacks, DTLS borrows the stateless cookie technique used by Photuris [RFC2522] and IKE [RFC7296]. When the client sends its ClientHello message to the server, the server **MAY** respond with a HelloRetryRequest message. The HelloRetryRequest message, as well as the "cookie" extension, is defined in TLS 1.3. The HelloRetryRequest message contains a stateless cookie (see [TLS13], Section 4.2.2). The client **MUST** send a new ClientHello with the cookie added as an extension. The server then verifies the cookie and proceeds with the handshake only if it is valid. This mechanism forces the attacker/client to be able to receive the cookie, which makes DoS attacks with spoofed IP addresses difficult. This mechanism does not provide any defense against DoS attacks mounted from valid IP addresses.

The DTLS 1.3 specification changes how cookies are exchanged compared to DTLS 1.2. DTLS 1.3 reuses the HelloRetryRequest message and conveys the cookie to the client via an extension. The client receiving the cookie uses the same extension to place the cookie subsequently into a ClientHello message. DTLS 1.2, on the other hand, used a separate message, namely the HelloVerifyRequest, to pass a cookie to the client and did not utilize the extension mechanism. For backwards compatibility reasons, the cookie field in the ClientHello is present in DTLS 1.3 but is ignored by a DTLS 1.3-compliant server implementation.

The exchange is shown in Figure 6. Note that the figure focuses on the cookie exchange; all other extensions are omitted.

Figure 6: DTLS Exchange with HelloRetryRequest Containing the "cookie" Extension

The "cookie" extension is defined in Section 4.2.2 of [TLS13]. When sending the initial ClientHello, the client does not have a cookie yet. In this case, the "cookie" extension is omitted and the legacy_cookie field in the ClientHello message **MUST** be set to a zero-length vector (i.e., a zero-valued single byte length field).

When responding to a HelloRetryRequest, the client **MUST** create a new ClientHello message following the description in Section 4.1.2 of [TLS13].

If the HelloRetryRequest message is used, the initial ClientHello and the HelloRetryRequest are included in the calculation of the transcript hash. The computation of the message hash for the HelloRetryRequest is done according to the description in Section 4.4.1 of [TLS13].

The handshake transcript is not reset with the second ClientHello, and a stateless server-cookie implementation requires the content or hash of the initial ClientHello (and HelloRetryRequest) to be stored in the cookie. The initial ClientHello is included in the handshake transcript as a synthetic "message_hash" message, so only the hash value is needed for the handshake to complete, though the complete HelloRetryRequest contents are needed.

When the second ClientHello is received, the server can verify that the cookie is valid and that the client can receive packets at the given IP address. If the client's apparent IP address is embedded in the cookie, this prevents an attacker from generating an acceptable ClientHello apparently from another user.

One potential attack on this scheme is for the attacker to collect a number of cookies from different addresses where it controls endpoints and then reuse them to attack the server. The server can defend against this attack by changing the secret value frequently, thus invalidating those cookies. If the server wishes to allow legitimate clients to handshake through the transition (e.g., a client received a cookie with Secret 1 and then sent the second ClientHello after the server has changed to Secret 2), the server can have a limited window during which it accepts both secrets. [RFC7296] suggests adding a key identifier to cookies to detect this case. An alternative approach is simply to try verifying with both secrets. It is **RECOMMENDED** that servers implement a key rotation scheme that allows the server to manage keys with overlapping lifetimes.

Alternatively, the server can store timestamps in the cookie and reject cookies that were generated outside a certain interval of time.

DTLS servers **SHOULD** perform a cookie exchange whenever a new handshake is being performed. If the server is being operated in an environment where amplification is not a problem, e.g., where ICE [RFC8445] has been used to establish bidirectional connectivity, the server **MAY** be configured not to perform a cookie exchange. The default **SHOULD** be that the exchange is performed, however. In addition, the server **MAY** choose not to do a cookie exchange when a session is resumed or, more generically, when the DTLS handshake uses a PSK-based key exchange and the IP address matches one associated with the PSK. Servers which process 0-RTT requests and send 0.5-RTT responses without a cookie exchange risk being used in an amplification attack if the size of outgoing messages greatly exceeds the size of those that are received. A server **SHOULD** limit the amount of data it sends toward a client address to three times the amount of data sent by the client before it verifies that the client is able to receive data at that address. A client address is valid after a cookie exchange or handshake completion. Clients **MUST** be prepared to do a cookie exchange with every handshake. Note that cookies are only valid for the existing handshake and cannot be stored for future handshakes.

If a server receives a ClientHello with an invalid cookie, it **MUST** terminate the handshake with an "illegal_parameter" alert. This allows the client to restart the connection from scratch without a cookie.

As described in Section 4.1.4 of [TLS13], clients **MUST** abort the handshake with an "unexpected_message" alert in response to any second HelloRetryRequest which was sent in the same connection (i.e., where the ClientHello was itself in response to a HelloRetryRequest).

DTLS clients which do not want to receive a Connection ID **SHOULD** still offer the "connection_id" extension [RFC9146] unless there is an application profile to the contrary. This permits a server which wants to receive a CID to negotiate one.

5.2. DTLS Handshake Message Format

DTLS uses the same Handshake messages as TLS 1.3. However, prior to transmission they are converted to DTLSHandshake messages, which contain extra data needed to support message loss, reordering, and message fragmentation.

```
    enum {
        client_hello(1),
        server_hello(2),
        new_session_ticket(4),
        end_of_early_data(5),
        encrypted_extensions(8),
        request_connection_id(9),           /* New */
        new_connection_id(10),              /* New */
        certificate(11),
        certificate_request(13),
        certificate_verify(15),
        finished(20),
        key_update(24),
        message_hash(254),
        (255)
    } HandshakeType;
```

```
    struct {
        HandshakeType msg_type;    /* handshake type */
        uint24 length;             /* bytes in message */
        uint16 message_seq;        /* DTLS-required field */
        uint24 fragment_offset;    /* DTLS-required field */
        uint24 fragment_length;    /* DTLS-required field */
        select (msg_type) {
            case client_hello:          ClientHello;
            case server_hello:          ServerHello;
            case end_of_early_data:     EndOfEarlyData;
            case encrypted_extensions:  EncryptedExtensions;
            case certificate_request:   CertificateRequest;
            case certificate:           Certificate;
            case certificate_verify:    CertificateVerify;
            case finished:              Finished;
            case new_session_ticket:    NewSessionTicket;
            case key_update:            KeyUpdate;
            case request_connection_id: RequestConnectionId;
            case new_connection_id:     NewConnectionId;
        } body;
    } DTLSHandshake;
```

In DTLS 1.3, the message transcript is computed over the original TLS 1.3-style Handshake messages without the message_seq, fragment_offset, and fragment_length values. Note that this is a change from DTLS 1.2 where those values were included in the transcript.

The first message each side transmits in each association always has message_seq = 0. Whenever a new message is generated, the message_seq value is incremented by one. When a message is retransmitted, the old message_seq value is reused, i.e., not incremented. From the perspective of the DTLS record layer, the retransmission is a new record. This record will have a new DTLSPlaintext.sequence_number value.

Note: In DTLS 1.2, the message_seq was reset to zero in case of a rehandshake (i.e., renegotiation). On the surface, a rehandshake in DTLS 1.2 shares similarities with a post-handshake message exchange in DTLS 1.3. However, in DTLS 1.3 the message_seq is not reset, to allow distinguishing a retransmission from a previously sent post-handshake message from a newly sent post-handshake message.

DTLS implementations maintain (at least notionally) a next_receive_seq counter. This counter is initially set to zero. When a handshake message is received, if its message_seq value matches next_receive_seq, next_receive_seq is incremented and the message is processed. If the sequence number is less than next_receive_seq, the message **MUST** be discarded. If the sequence number is greater than next_receive_seq, the implementation **SHOULD** queue the message but **MAY** discard it. (This is a simple space/bandwidth trade-off).

In addition to the handshake messages that are deprecated by the TLS 1.3 specification, DTLS 1.3 furthermore deprecates the HelloVerifyRequest message originally defined in DTLS 1.0. DTLS 1.3-compliant implementations **MUST NOT** use the HelloVerifyRequest to execute a return-routability check. A dual-stack DTLS 1.2 / DTLS 1.3 client **MUST**, however, be prepared to interact with a DTLS 1.2 server.

5.3. ClientHello Message

The format of the ClientHello used by a DTLS 1.3 client differs from the TLS 1.3 ClientHello format, as shown below.

```
uint16 ProtocolVersion;
opaque Random[32];

uint8 CipherSuite[2];    /* Cryptographic suite selector */

struct {
    ProtocolVersion legacy_version = { 254,253 }; // DTLSv1.2
    Random random;
    opaque legacy_session_id<0..32>;
    opaque legacy_cookie<0..2^8-1>;                  // DTLS
    CipherSuite cipher_suites<2..2^16-2>;
    opaque legacy_compression_methods<1..2^8-1>;
    Extension extensions<8..2^16-1>;
} ClientHello;
```

legacy_version: In previous versions of DTLS, this field was used for version negotiation and represented the highest version number supported by the client. Experience has shown that many servers do not properly implement version negotiation, leading to "version intolerance" in which the server rejects an otherwise acceptable ClientHello with a version number higher than it supports. In DTLS 1.3, the client indicates its version preferences in the "supported_versions" extension (see Section 4.2.1 of [TLS13]) and the legacy_version field **MUST** be set to {254, 253}, which was the version number for DTLS 1.2. The supported_versions entries for DTLS 1.0 and DTLS 1.2 are 0xfeff and 0xfefd (to match the wire versions). The value 0xfefc is used to indicate DTLS 1.3.

random: Same as for TLS 1.3, except that the downgrade sentinels described in Section 4.1.3 of [TLS13] when TLS 1.2 and TLS 1.1 and below are negotiated apply to DTLS 1.2 and DTLS 1.0, respectively.

legacy_session_id: Versions of TLS and DTLS before version 1.3 supported a "session resumption" feature, which has been merged with pre-shared keys (PSK) in version 1.3. A client which has a cached session ID set by a pre-DTLS 1.3 server **SHOULD** set this field to that value. Otherwise, it **MUST** be set as a zero-length vector (i.e., a zero-valued single byte length field).

legacy_cookie: A DTLS 1.3-only client **MUST** set the legacy_cookie field to zero length. If a DTLS 1.3 ClientHello is received with any other value in this field, the server **MUST** abort the handshake with an "illegal_parameter" alert.

cipher_suites: Same as for TLS 1.3; only suites with DTLS-OK=Y may be used.

legacy_compression_methods: Same as for TLS 1.3.

extensions: Same as for TLS 1.3.

5.4. ServerHello Message

The DTLS 1.3 ServerHello message is the same as the TLS 1.3 ServerHello message, except that the legacy_version field is set to 0xfefd, indicating DTLS 1.2.

5.5. Handshake Message Fragmentation and Reassembly

As described in Section 4.3, one or more handshake messages may be carried in a single datagram. However, handshake messages are potentially bigger than the size allowed by the underlying datagram transport. DTLS provides a mechanism for fragmenting a handshake message over a number of records, each of which can be transmitted in separate datagrams, thus avoiding IP fragmentation.

When transmitting the handshake message, the sender divides the message into a series of N contiguous data ranges. The ranges **MUST NOT** overlap. The sender then creates N DTLSHandshake messages, all with the same message_seq value as the original DTLSHandshake message. Each new message is labeled with the fragment_offset (the number of bytes contained in previous fragments) and the fragment_length (the length of this fragment). The length field in all messages is the same as the length field of the original message. An unfragmented message is a degenerate case with fragment_offset=0 and fragment_length=length. Each handshake message fragment that is placed into a record **MUST** be delivered in a single UDP datagram.

When a DTLS implementation receives a handshake message fragment corresponding to the next expected handshake message sequence number, it **MUST** process it, either by buffering it until it has the entire handshake message or by processing any in-order portions of the message. The transcript consists of complete TLS Handshake messages (reassembled as necessary). Note that this requires removing the message_seq, fragment_offset, and fragment_length fields to create the Handshake structure.

DTLS implementations **MUST** be able to handle overlapping fragment ranges. This allows senders to retransmit handshake messages with smaller fragment sizes if the PMTU estimate changes. Senders **MUST NOT** change handshake message bytes upon retransmission. Receivers **MAY** check that retransmitted bytes are identical and **SHOULD** abort the handshake with an "illegal_parameter" alert if the value of a byte changes.

Note that as with TLS, multiple handshake messages may be placed in the same DTLS record, provided that there is room and that they are part of the same flight. Thus, there are two acceptable ways to pack two DTLS handshake messages into the same datagram: in the same record or in separate records.

5.6. EndOfEarlyData Message

The DTLS 1.3 handshake has one important difference from the TLS 1.3 handshake: the EndOfEarlyData message is omitted both from the wire and the handshake transcript. Because DTLS records have epochs, EndOfEarlyData is not necessary to determine when the early data is complete, and because DTLS is lossy, attackers can trivially mount the deletion attacks that EndOfEarlyData prevents in TLS. Servers **SHOULD NOT** accept records from epoch 1 indefinitely once they are able to process records from epoch 3. Though reordering of IP packets can result in records from epoch 1 arriving after records from epoch 3, this is not likely to persist for very long relative to the round trip time. Servers could discard epoch 1 keys after the first epoch 3 data arrives, or retain keys for processing epoch 1 data for a short period. (See Section 6.1 for the definitions of each epoch.)

5.7. DTLS Handshake Flights

DTLS handshake messages are grouped into a series of message flights. A flight starts with the handshake message transmission of one peer and ends with the expected response from the other peer. Table 1 contains a complete list of message combinations that constitute flights.

Note	Client	Server	Handshake Messages
	x		ClientHello
		x	HelloRetryRequest
		x	ServerHello, EncryptedExtensions, CertificateRequest, Certificate, CertificateVerify, Finished
1	x		Certificate, CertificateVerify, Finished
1		x	NewSessionTicket

Table 1: Flight Handshake Message Combinations

Remarks:

- Table 1 does not highlight any of the optional messages.

- Regarding note (1): When a handshake flight is sent without any expected response, as is the case with the client's final flight or with the NewSessionTicket message, the flight must be acknowledged with an ACK message.

Below are several example message exchanges illustrating the flight concept. The notational conventions from [TLS13] are used.

Figure 7: Message Flights for a Full DTLS Handshake (with Cookie Exchange)

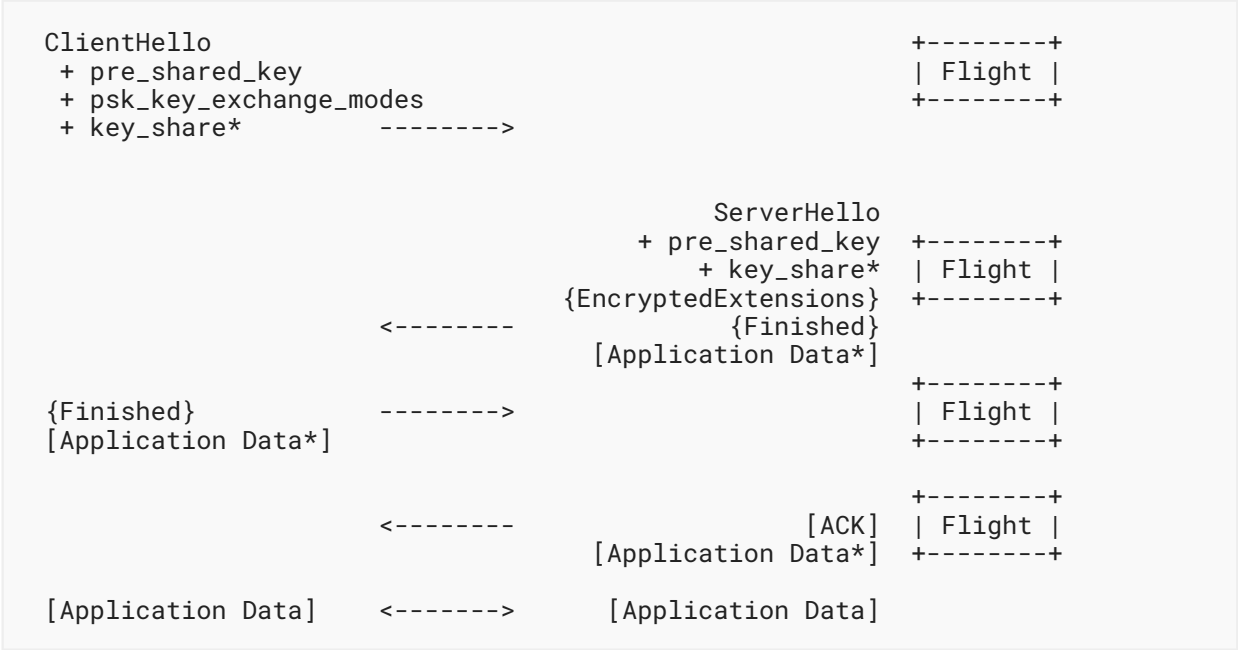


Figure 8: Message Flights for Resumption and PSK Handshake (without Cookie Exchange)

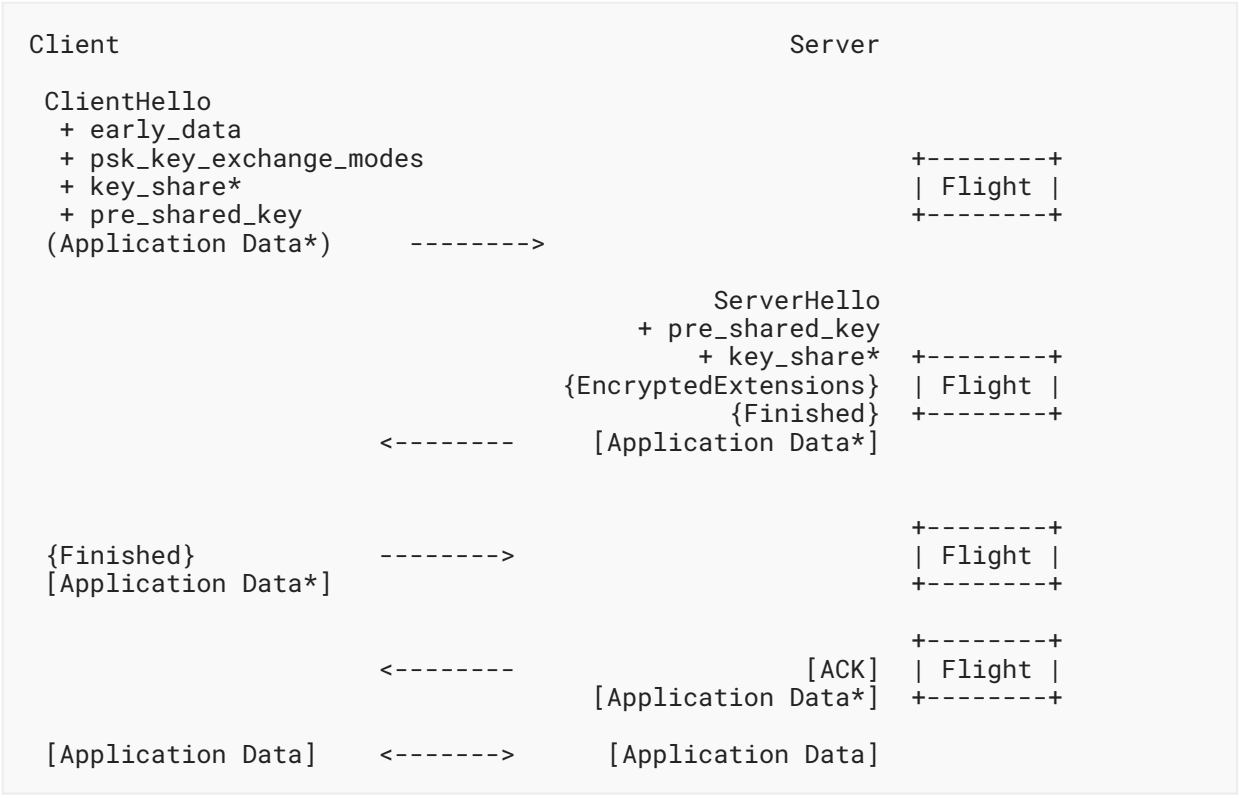


Figure 9: Message Flights for the Zero-RTT Handshake

Figure 10: Message Flights for the NewSessionTicket Message

KeyUpdate, NewConnectionId, and RequestConnectionId follow a similar pattern to NewSessionTicket: a single message sent by one side followed by an ACK by the other.

5.8. Timeout and Retransmission

5.8.1. State Machine

DTLS uses a simple timeout and retransmission scheme with the state machine shown in Figure 11.

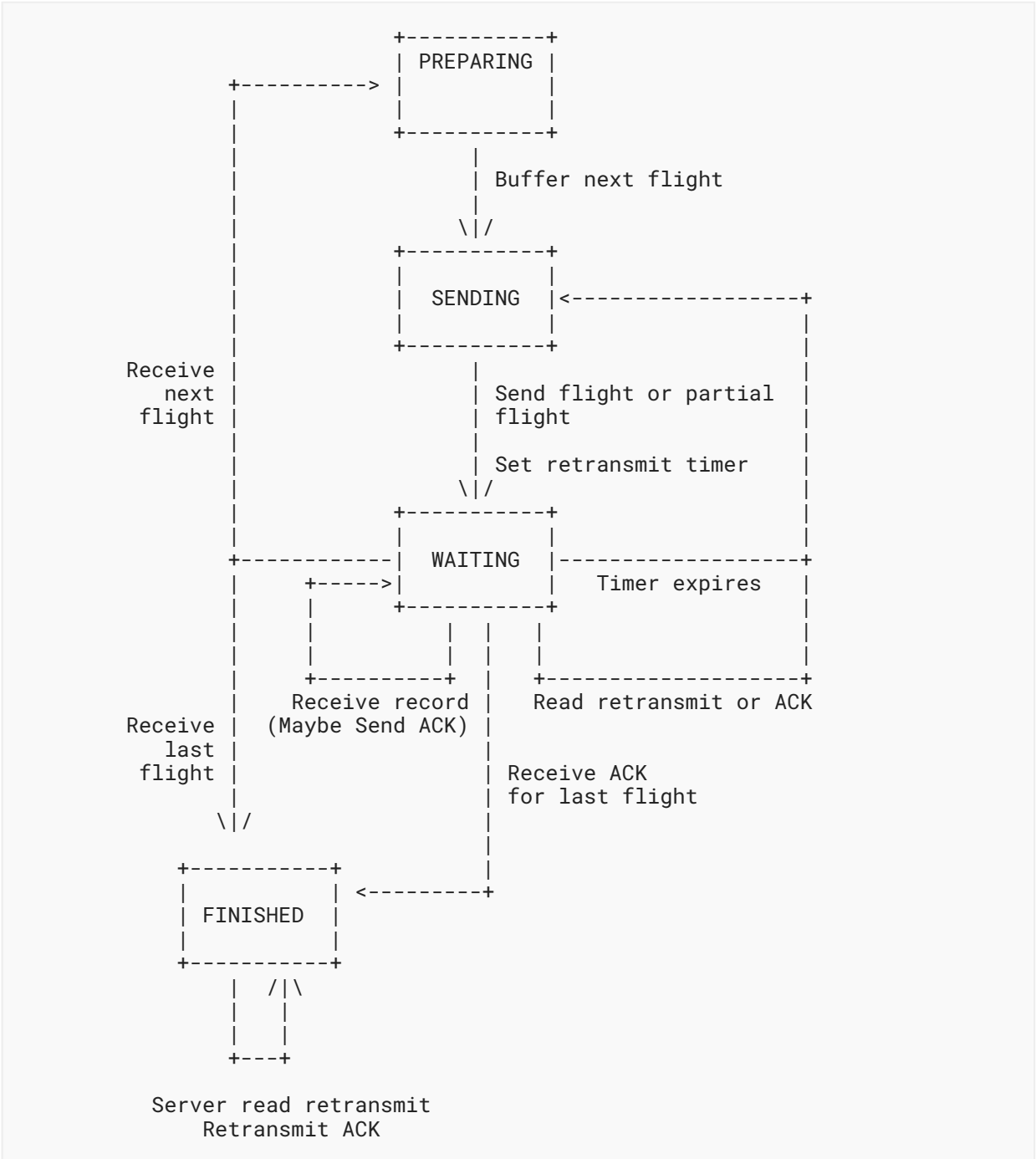


Figure 11: DTLS Timeout and Retransmission State Machine

The state machine has four basic states: PREPARING, SENDING, WAITING, and FINISHED.

In the PREPARING state, the implementation does whatever computations are necessary to prepare the next flight of messages. It then buffers them up for transmission (emptying the transmission buffer first) and enters the SENDING state.

In the SENDING state, the implementation transmits the buffered flight of messages. If the implementation has received one or more ACKs (see Section 7) from the peer, then it **SHOULD** omit any messages or message fragments which have already been acknowledged. Once the messages have been sent, the implementation then sets a retransmit timer and enters the WAITING state.

There are four ways to exit the WAITING state:

1. The retransmit timer expires: the implementation transitions to the SENDING state, where it retransmits the flight, adjusts and re-arms the retransmit timer (see Section 5.8.2), and returns to the WAITING state.
2. The implementation reads an ACK from the peer: upon receiving an ACK for a partial flight (as mentioned in Section 7.1), the implementation transitions to the SENDING state, where it retransmits the unacknowledged portion of the flight, adjusts and re-arms the retransmit timer, and returns to the WAITING state. Upon receiving an ACK for a complete flight, the implementation cancels all retransmissions and either remains in WAITING, or, if the ACK was for the final flight, transitions to FINISHED.
3. The implementation reads a retransmitted flight from the peer when none of the messages that it sent in response to that flight have been acknowledged: the implementation transitions to the SENDING state, where it retransmits the flight, adjusts and re-arms the retransmit timer, and returns to the WAITING state. The rationale here is that the receipt of a duplicate message is the likely result of timer expiry on the peer and therefore suggests that part of one's previous flight was lost.
4. The implementation receives some or all of the next flight of messages: if this is the final flight of messages, the implementation transitions to FINISHED. If the implementation needs to send a new flight, it transitions to the PREPARING state. Partial reads (whether partial messages or only some of the messages in the flight) may also trigger the implementation to send an ACK, as described in Section 7.1.

Because DTLS clients send the first message (ClientHello), they start in the PREPARING state. DTLS servers start in the WAITING state, but with empty buffers and no retransmit timer.

In addition, for at least twice the default MSL defined for [RFC0793], when in the FINISHED state, the server **MUST** respond to retransmission of the client's final flight with a retransmit of its ACK.

Note that because of packet loss, it is possible for one side to be sending application data even though the other side has not received the first side's Finished message. Implementations **MUST** either discard or buffer all application data records for epoch 3 and above until they have received the Finished message from the peer. Implementations **MAY** treat receipt of application data with a new epoch prior to receipt of the corresponding Finished message as evidence of reordering or packet loss and retransmit their final flight immediately, shortcutting the retransmission timer.

5.8.2. Timer Values

The configuration of timer settings varies with implementations, and certain deployment environments require timer value adjustments. Mishandling of the timer can lead to serious congestion problems -- for example, if many instances of a DTLS time out early and retransmit too quickly on a congested link.

Unless implementations have deployment-specific and/or external information about the round trip time, implementations **SHOULD** use an initial timer value of 1000 ms and double the value at each retransmission, up to no less than 60 seconds (the maximum as specified in RFC 6298 [RFC6298]). Application-specific profiles **MAY** recommend shorter or longer timer values. For instance:

- Profiles for specific deployment environments, such as in low-power, multi-hop mesh scenarios as used in some Internet of Things (IoT) networks, **MAY** specify longer timeouts. See [IOT-PROFILE] for more information about one such DTLS 1.3 IoT profile.
- Real-time protocols **MAY** specify shorter timeouts. It is **RECOMMENDED** that for DTLS-SRTP [RFC5764], a default timeout of 400 ms be used; because customer experience degrades with one-way latencies of greater than 200 ms, real-time deployments are less likely to have long latencies.

In settings where there is external information (for instance, from an ICE [RFC8445] handshake, or from previous connections to the same server) about the RTT, implementations **SHOULD** use 1.5 times that RTT estimate as the retransmit timer.

Implementations **SHOULD** retain the current timer value until a message is transmitted and acknowledged without having to be retransmitted, at which time the value **SHOULD** be adjusted to 1.5 times the measured round trip time for that message. After a long period of idleness, no less than 10 times the current timer value, implementations **MAY** reset the timer to the initial value.

Note that because retransmission is for the handshake and not dataflow, the effect on congestion of shorter timeouts is smaller than in generic protocols such as TCP or QUIC. Experience with DTLS 1.2, which uses a simpler "retransmit everything on timeout" approach, has not shown serious congestion problems in practice.

5.8.3. Large Flight Sizes

DTLS does not have any built-in congestion control or rate control; in general, this is not an issue because messages tend to be small. However, in principle, some messages -- especially Certificate -- can be quite large. If all the messages in a large flight are sent at once, this can result in network congestion. A better strategy is to send out only part of the flight, sending more when messages are acknowledged. Several extensions have been standardized to reduce the size of the Certificate message -- for example, the "cached_info" extension [RFC7924]; certificate compression [RFC8879]; and [RFC6066], which defines the "client_certificate_url" extension allowing DTLS clients to send a sequence of Uniform Resource Locators (URLs) instead of the client certificate.

DTLS stacks **SHOULD NOT** send more than 10 records in a single transmission.

5.8.4. State Machine Duplication for Post-Handshake Messages

DTLS 1.3 makes use of the following categories of post-handshake messages:

1. NewSessionTicket
2. KeyUpdate
3. NewConnectionId
4. RequestConnectionId
5. Post-handshake client authentication

Messages of each category can be sent independently, and reliability is established via independent state machines, each of which behaves as described in Section 5.8.1. For example, if a server sends a NewSessionTicket and a CertificateRequest message, two independent state machines will be created.

Sending multiple instances of messages of a given category without having completed earlier transmissions is allowed for some categories, but not for others. Specifically, a server **MAY** send multiple NewSessionTicket messages at once without awaiting ACKs for earlier NewSessionTicket messages first. Likewise, a server **MAY** send multiple CertificateRequest messages at once without having completed earlier client authentication requests before. In contrast, implementations **MUST NOT** send KeyUpdate, NewConnectionId, or RequestConnectionId messages if an earlier message of the same type has not yet been acknowledged.

> Note: Except for post-handshake client authentication, which involves handshake messages in both directions, post-handshake messages are single-flight, and their respective state machines on the sender side reduce to waiting for an ACK and retransmitting the original message. In particular, note that a RequestConnectionId message does not force the receiver to send a NewConnectionId message in reply, and both messages are therefore treated independently.

Creating and correctly updating multiple state machines requires feedback from the handshake logic to the state machine layer, indicating which message belongs to which state machine. For example, if a server sends multiple CertificateRequest messages and receives a Certificate message in response, the corresponding state machine can only be determined after inspecting the certificate_request_context field. Similarly, a server sending a single CertificateRequest and receiving a NewConnectionId message in response can only decide that the NewConnectionId message should be treated through an independent state machine after inspecting the handshake message type.

5.9. Cryptographic Label Prefix

Section 7.1 of [TLS13] specifies that HKDF-Expand-Label uses a label prefix of "tls13 ". For DTLS 1.3, that label **SHALL** be "dtls13". This ensures key separation between DTLS 1.3 and TLS 1.3. Note that there is no trailing space; this is necessary in order to keep the overall label size inside of one hash iteration because "DTLS" is one letter longer than "TLS".

5.10. Alert Messages

Note that alert messages are not retransmitted at all, even when they occur in the context of a handshake. However, a DTLS implementation which would ordinarily issue an alert **SHOULD** generate a new alert message if the offending record is received again (e.g., as a retransmitted handshake message). Implementations **SHOULD** detect when a peer is persistently sending bad messages and terminate the local connection state after such misbehavior is detected. Note that alerts are not reliably transmitted; implementations **SHOULD NOT** depend on receiving alerts in order to signal errors or connection closure.

Any data received with an epoch/sequence number pair after that of a valid received closure alert **MUST** be ignored. Note: this is a change from TLS 1.3 which depends on the order of receipt rather than the epoch and sequence number.

5.11. Establishing New Associations with Existing Parameters

If a DTLS client-server pair is configured in such a way that repeated connections happen on the same host/port quartet, then it is possible that a client will silently abandon one connection and then initiate another with the same parameters (e.g., after a reboot). This will appear to the server as a new handshake with epoch=0. In cases where a server believes it has an existing association on a given host/port quartet and it receives an epoch=0 ClientHello, it **SHOULD** proceed with a new handshake but **MUST NOT** destroy the existing association until the client has demonstrated reachability either by completing a cookie exchange or by completing a complete handshake including delivering a verifiable Finished message. After a correct Finished message is received, the server **MUST** abandon the previous association to avoid confusion between two valid associations with overlapping epochs. The reachability requirement prevents off-path/blind attackers from destroying associations merely by sending forged ClientHellos.

> Note: It is not always possible to distinguish which association a given record is from. For instance, if the client performs a handshake, abandons the connection, and then immediately starts a new handshake, it may not be possible to tell which connection a given protected record is for. In these cases, trial decryption may be necessary, though implementations could use CIDs to avoid the 5-tuple-based ambiguity.

6. Example of Handshake with Timeout and Retransmission

The following is an example of a handshake with lost packets and retransmissions. Note that the client sends an empty ACK message because it can only acknowledge Record 2 sent by the server once it has processed messages in Record 0 needed to establish epoch 2 keys, which are needed to encrypt or decrypt messages found in Record 2. Section 7 provides the necessary background details for this interaction. Note: For simplicity, we are not resetting record numbers in this diagram, so "Record 1" is really "Epoch 2, Record 0", etc.

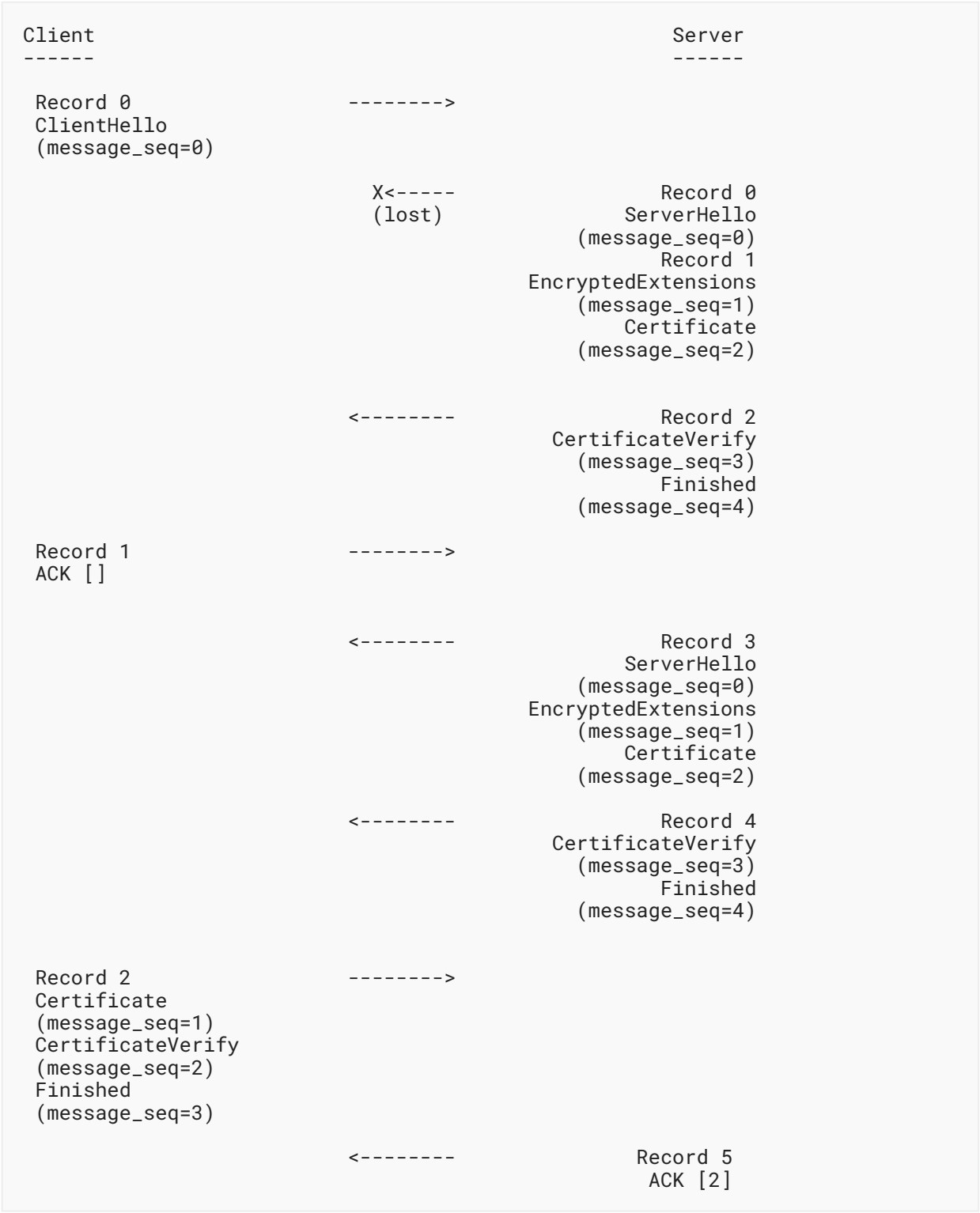


Figure 12: Example DTLS Exchange Illustrating Message Loss

6.1. Epoch Values and Rekeying

A recipient of a DTLS message needs to select the correct keying material in order to process an incoming message. With the possibility of message loss and reordering, an identifier is needed to determine which cipher state has been used to protect the record payload. The epoch value fulfills this role in DTLS. In addition to the TLS 1.3-defined key derivation steps (see Section 7 of [TLS13]), a sender may want to rekey at any time during the lifetime of the connection. It therefore needs to indicate that it is updating its sending cryptographic keys.

This version of DTLS assigns dedicated epoch values to messages in the protocol exchange to allow identification of the correct cipher state:

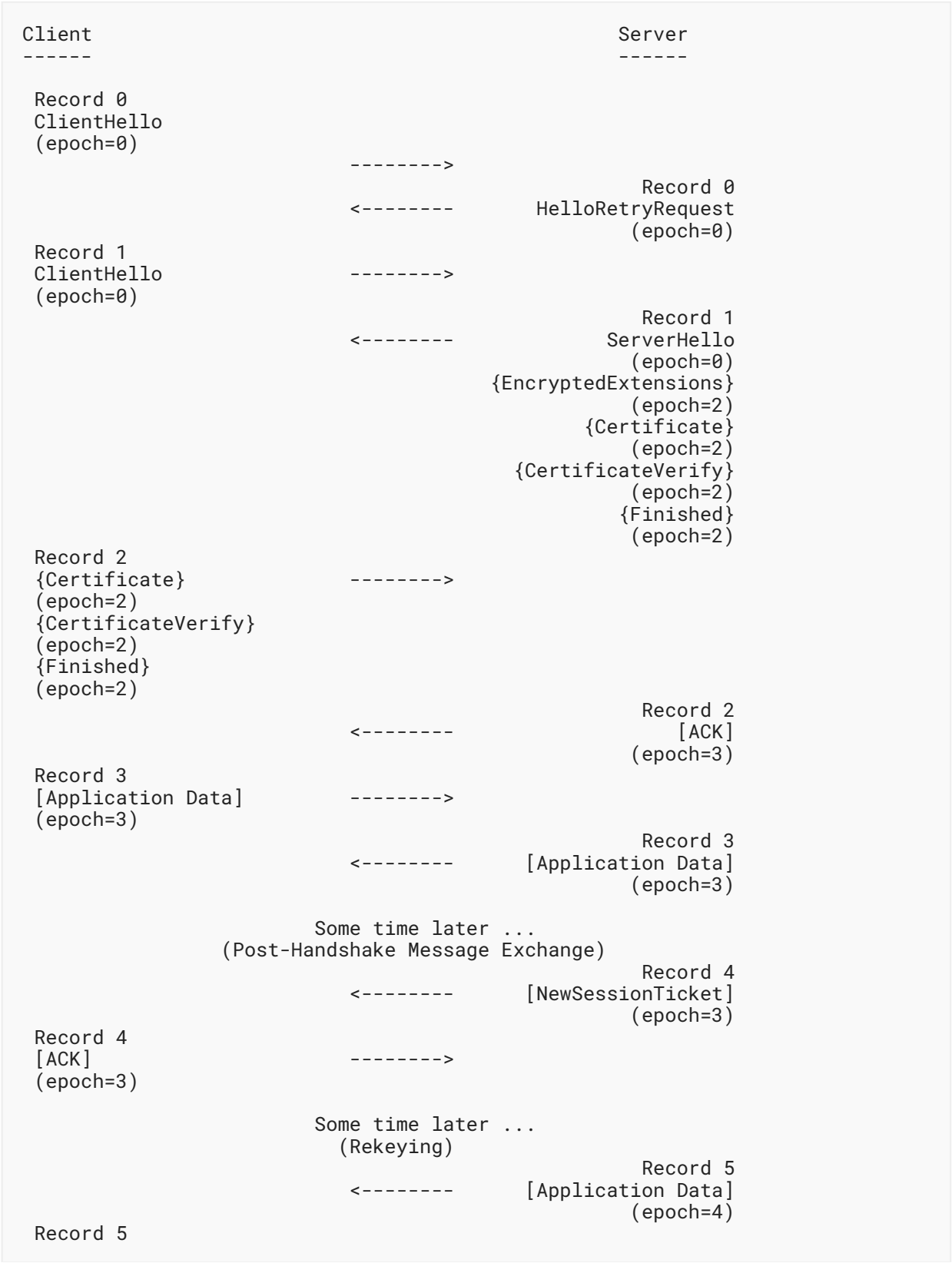
- Epoch value (0) is used with unencrypted messages. There are three unencrypted messages in DTLS, namely ClientHello, ServerHello, and HelloRetryRequest.
- Epoch value (1) is used for messages protected using keys derived from client_early_traffic_secret. Note that this epoch is skipped if the client does not offer early data.
- Epoch value (2) is used for messages protected using keys derived from [sender]_handshake_traffic_secret. Messages transmitted during the initial handshake, such as EncryptedExtensions, CertificateRequest, Certificate, CertificateVerify, and Finished, belong to this category. Note, however, that post-handshake messages are protected under the appropriate application traffic key and are not included in this category.
- Epoch value (3) is used for payloads protected using keys derived from the initial [sender]_application_traffic_secret_0. This may include handshake messages, such as post-handshake messages (e.g., a NewSessionTicket message).
- Epoch values (4 to 2^64-1) are used for payloads protected using keys from the [sender]_application_traffic_secret_N (N>0).

Using these reserved epoch values, a receiver knows what cipher state has been used to encrypt and integrity protect a message. Implementations that receive a record with an epoch value for which no corresponding cipher state can be determined **SHOULD** handle it as a record which fails deprotection.

Note that epoch values do not wrap. If a DTLS implementation would need to wrap the epoch value, it **MUST** terminate the connection.

The traffic key calculation is described in Section 7.3 of [TLS13].

Figure 13 illustrates the epoch values in an example DTLS handshake.


Client
Server
Record 0
ClientHello
(epoch=0)
Record 0
HelloRetryRequest
(epoch=0)
Record 1
ClientHello
(epoch=0)
Record 1
ServerHello
(epoch=0)
{EncryptedExtensions}
(epoch=2)
{Certificate}
(epoch=2)
{CertificateVerify}
(epoch=2)
{Finished}
(epoch=2)
Record 2
{Certificate}
(epoch=2)
{CertificateVerify}
(epoch=2)
{Finished}
(epoch=2)
Record 2
[ACK]
(epoch=3)
Record 3
[Application Data]
(epoch=3)
Record 3
[Application Data]
(epoch=3)
Some time later ...
(Post-Handshake Message Exchange)
Record 4
[NewSessionTicket]
(epoch=3)
Record 4
[ACK]
(epoch=3)
Some time later ...
(Rekeying)
Record 5
[Application Data]
(epoch=4)
Record 5

```
[Application Data]         -------->
(epoch=4)
```

Figure 13: Example DTLS Exchange with Epoch Information

7. ACK Message

The ACK message is used by an endpoint to indicate which handshake records it has received and processed from the other side. ACK is not a handshake message but is rather a separate content type, with code point 26. This avoids having ACK being added to the handshake transcript. Note that ACKs can still be sent in the same UDP datagram as handshake records.

```
struct {
    RecordNumber record_numbers<0..2^16-1>;
} ACK;
```

record_numbers: A list of the records containing handshake messages in the current flight which the endpoint has received and either processed or buffered, in numerically increasing order.

Implementations **MUST NOT** acknowledge records containing handshake messages or fragments which have not been processed or buffered. Otherwise, deadlock can ensue. As an example, implementations **MUST NOT** send ACKs for handshake messages which they discard because they are not the next expected message.

During the handshake, ACKs only cover the current outstanding flight (this is possible because DTLS is generally a lock-step protocol). In particular, receiving a message from a handshake flight implicitly acknowledges all messages from the previous flight(s). Accordingly, an ACK from the server would not cover both the ClientHello and the client's Certificate message, because the ClientHello and client Certificate are in different flights. Implementations can accomplish this by clearing their ACK list upon receiving the start of the next flight.

For post-handshake messages, ACKs **SHOULD** be sent once for each received and processed handshake record (potentially subject to some delay) and **MAY** cover more than one flight. This includes records containing messages which are discarded because a previous copy has been received.

During the handshake, ACK records **MUST** be sent with an epoch which is equal to or higher than the record which is being acknowledged. Note that some care is required when processing flights spanning multiple epochs. For instance, if the client receives only the ServerHello and Certificate and wishes to ACK them in a single record, it must do so in epoch 2, as it is required to use an epoch greater than or equal to 2 and cannot yet send with any greater epoch. Implementations **SHOULD** simply use the highest current sending epoch, which will generally be the highest available. After the handshake, implementations **MUST** use the highest available sending epoch.

7.1. Sending ACKs

When an implementation detects a disruption in the receipt of the current incoming flight, it **SHOULD** generate an ACK that covers the messages from that flight which it has received and processed so far. Implementations have some discretion about which events to treat as signs of disruption, but it is **RECOMMENDED** that they generate ACKs under two circumstances:

- When they receive a message or fragment which is out of order, either because it is not the next expected message or because it is not the next piece of the current message.
- When they have received part of a flight and do not immediately receive the rest of the flight (which may be in the same UDP datagram). "Immediately" is hard to define. One approach is to set a timer for 1/4 the current retransmit timer value when the first record in the flight is received and then send an ACK when that timer expires. Note: The 1/4 value here is somewhat arbitrary. Given that the round trip estimates in the DTLS handshake are generally very rough (or the default), any value will be an approximation, and there is an inherent compromise due to competition between retransmission due to over-aggressive ACKing and over-aggressive timeout-based retransmission. As a comparison point, QUIC's loss-based recovery algorithms ([RFC9002], Section 6.1.2) work out to a delay of about 1/3 of the retransmit timer.

In general, flights **MUST** be ACKed unless they are implicitly acknowledged. In the present specification, the following flights are implicitly acknowledged by the receipt of the next flight, which generally immediately follows the flight:

1. Handshake flights other than the client's final flight of the main handshake.
2. The server's post-handshake CertificateRequest.

ACKs **SHOULD NOT** be sent for these flights unless the responding flight cannot be generated immediately. All other flights **MUST** be ACKed. In this case, implementations **MAY** send explicit ACKs for the complete received flight even though it will eventually also be implicitly acknowledged through the responding flight. A notable example for this is the case of client authentication in constrained environments, where generating the CertificateVerify message can take considerable time on the client. Implementations **MAY** acknowledge the records corresponding to each transmission of each flight or simply acknowledge the most recent one. In general, implementations **SHOULD** ACK as many received packets as can fit into the ACK record, as this provides the most complete information and thus reduces the chance of spurious retransmission; if space is limited, implementations **SHOULD** favor including records which have not yet been acknowledged.

> Note: While some post-handshake messages follow a request/response pattern, this does not necessarily imply receipt. For example, a KeyUpdate sent in response to a KeyUpdate with request_update set to "update_requested" does not implicitly acknowledge the earlier KeyUpdate message because the two KeyUpdate messages might have crossed in flight.

ACKs **MUST NOT** be sent for records of any content type other than handshake or for records which cannot be deprotected.

Note that in some cases it may be necessary to send an ACK which does not contain any record numbers. For instance, a client might receive an EncryptedExtensions message prior to receiving a ServerHello. Because it cannot decrypt the EncryptedExtensions, it cannot safely acknowledge it (as it might be damaged). If the client does not send an ACK, the server will eventually retransmit its first flight, but this might take far longer than the actual round trip time between client and server. Having the client send an empty ACK shortcuts this process.

7.2. Receiving ACKs

When an implementation receives an ACK, it **SHOULD** record that the messages or message fragments sent in the records being ACKed were received and omit them from any future retransmissions. Upon receipt of an ACK that leaves it with only some messages from a flight having been acknowledged, an implementation **SHOULD** retransmit the unacknowledged messages or fragments. Note that this requires implementations to track which messages appear in which records. Once all the messages in a flight have been acknowledged, the implementation **MUST** cancel all retransmissions of that flight. Implementations **MUST** treat a record as having been acknowledged if it appears in any ACK; this prevents spurious retransmission in cases where a flight is very large and the receiver is forced to elide acknowledgements for records which have already been ACKed. As noted above, the receipt of any record responding to a given flight **MUST** be taken as an implicit acknowledgement for the entire flight to which it is responding.

7.3. Design Rationale

ACK messages are used in two circumstances, namely:

- On sign of disruption, or lack of progress; and
- To indicate complete receipt of the last flight in a handshake.

In the first case, the use of the ACK message is optional, because the peer will retransmit in any case and therefore the ACK just allows for selective or early retransmission, as opposed to the timeout-based whole flight retransmission in previous versions of DTLS. When DTLS 1.3 is used in deployments with lossy networks, such as low-power, long-range radio networks as well as low-power mesh networks, the use of ACKs is recommended.

The use of the ACK for the second case is mandatory for the proper functioning of the protocol. For instance, the ACK message sent by the client in Figure 13 acknowledges receipt and processing of Record 4 (containing the NewSessionTicket message), and if it is not sent, the server will continue retransmission of the NewSessionTicket indefinitely until its maximum retransmission count is reached.

8. Key Updates

As with TLS 1.3, DTLS 1.3 implementations send a KeyUpdate message to indicate that they are updating their sending keys. As with other handshake messages with no built-in response, KeyUpdates **MUST** be acknowledged. In order to facilitate epoch reconstruction (Section 4.2.2), implementations **MUST NOT** send records with the new keys or send a new KeyUpdate until the previous KeyUpdate has been acknowledged (this avoids having too many epochs in active use).

Due to loss and/or reordering, DTLS 1.3 implementations may receive a record with an older epoch than the current one (the requirements above preclude receiving a newer record). They **SHOULD** attempt to process those records with that epoch (see Section 4.2.2 for information on determining the correct epoch) but **MAY** opt to discard such out-of-epoch records.

Due to the possibility of an ACK message for a KeyUpdate being lost and thereby preventing the sender of the KeyUpdate from updating its keying material, receivers **MUST** retain the pre-update keying material until receipt and successful decryption of a message using the new keys.

Figure 14 shows an example exchange illustrating that successful ACK processing updates the keys of the KeyUpdate message sender, which is reflected in the change of epoch values.

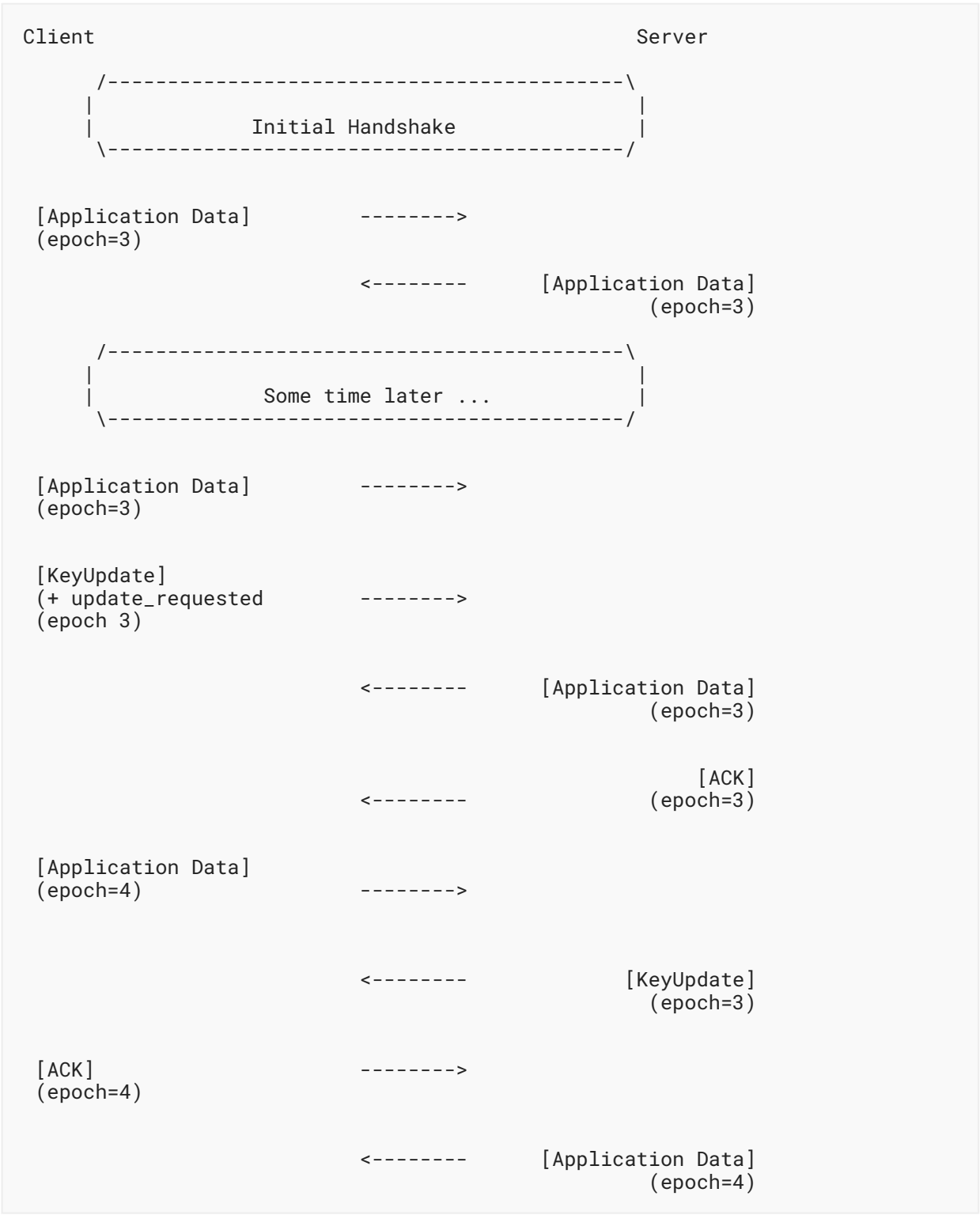


Figure 14: Example DTLS Key Update

With a 128-bit key as in AES-128, rekeying 2^64 times has a high probability of key reuse within a given connection. Note that even if the key repeats, the IV is also independently generated. In order to provide an extra margin of security, sending implementations **MUST NOT** allow the epoch to exceed 2^48-1. In order to allow this value to be changed later, receiving implementations **MUST NOT** enforce this rule. If a sending implementation receives a KeyUpdate with request_update set to "update_requested", it **MUST NOT** send its own KeyUpdate if that would cause it to exceed these limits and **SHOULD** instead ignore the "update_requested" flag. Note: this overrides the requirement in TLS 1.3 to always send a KeyUpdate in response to "update_requested".

9. Connection ID Updates

If the client and server have negotiated the "connection_id" extension [RFC9146], either side can send a new CID that it wishes the other side to use in a NewConnectionId message.

```
enum {
    cid_immediate(0), cid_spare(1), (255)
} ConnectionIdUsage;

opaque ConnectionId<0..2^8-1>;

struct {
    ConnectionId cids<0..2^16-1>;
    ConnectionIdUsage usage;
} NewConnectionId;
```

cids: Indicates the set of CIDs that the sender wishes the peer to use.

usage: Indicates whether the new CIDs should be used immediately or are spare. If usage is set to "cid_immediate", then one of the new CIDs **MUST** be used immediately for all future records. If it is set to "cid_spare", then either an existing or new CID **MAY** be used.

Endpoints **SHOULD** use receiver-provided CIDs in the order they were provided. Implementations which receive more spare CIDs than they wish to maintain **MAY** simply discard any extra CIDs. Endpoints **MUST NOT** have more than one NewConnectionId message outstanding.

Implementations which either did not negotiate the "connection_id" extension or which have negotiated receiving an empty CID **MUST NOT** send NewConnectionId. Implementations **MUST NOT** send RequestConnectionId when sending an empty Connection ID. Implementations which detect a violation of these rules **MUST** terminate the connection with an "unexpected_message" alert.

Implementations **SHOULD** use a new CID whenever sending on a new path and **SHOULD** request new CIDs for this purpose if path changes are anticipated.

```
struct {
  uint8 num_cids;
} RequestConnectionId;
```

num_cids: The number of CIDs desired.

Endpoints **SHOULD** respond to RequestConnectionId by sending a NewConnectionId with usage "cid_spare" containing num_cids CIDs as soon as possible. Endpoints **MUST NOT** send a RequestConnectionId message when an existing request is still unfulfilled; this implies that endpoints need to request new CIDs well in advance. An endpoint **MAY** handle requests which it considers excessive by responding with a NewConnectionId message containing fewer than num_cids CIDs, including no CIDs at all. Endpoints **MAY** handle an excessive number of RequestConnectionId messages by terminating the connection using a "too_many_cids_requested" (alert number 52) alert.

Endpoints **MUST NOT** send either of these messages if they did not negotiate a CID. If an implementation receives these messages when CIDs were not negotiated, it **MUST** abort the connection with an "unexpected_message" alert.

9.1. Connection ID Example

Below is an example exchange for DTLS 1.3 using a single CID in each direction.

> Note: The "connection_id" extension, which is used in ClientHello and ServerHello messages, is defined in [RFC9146].

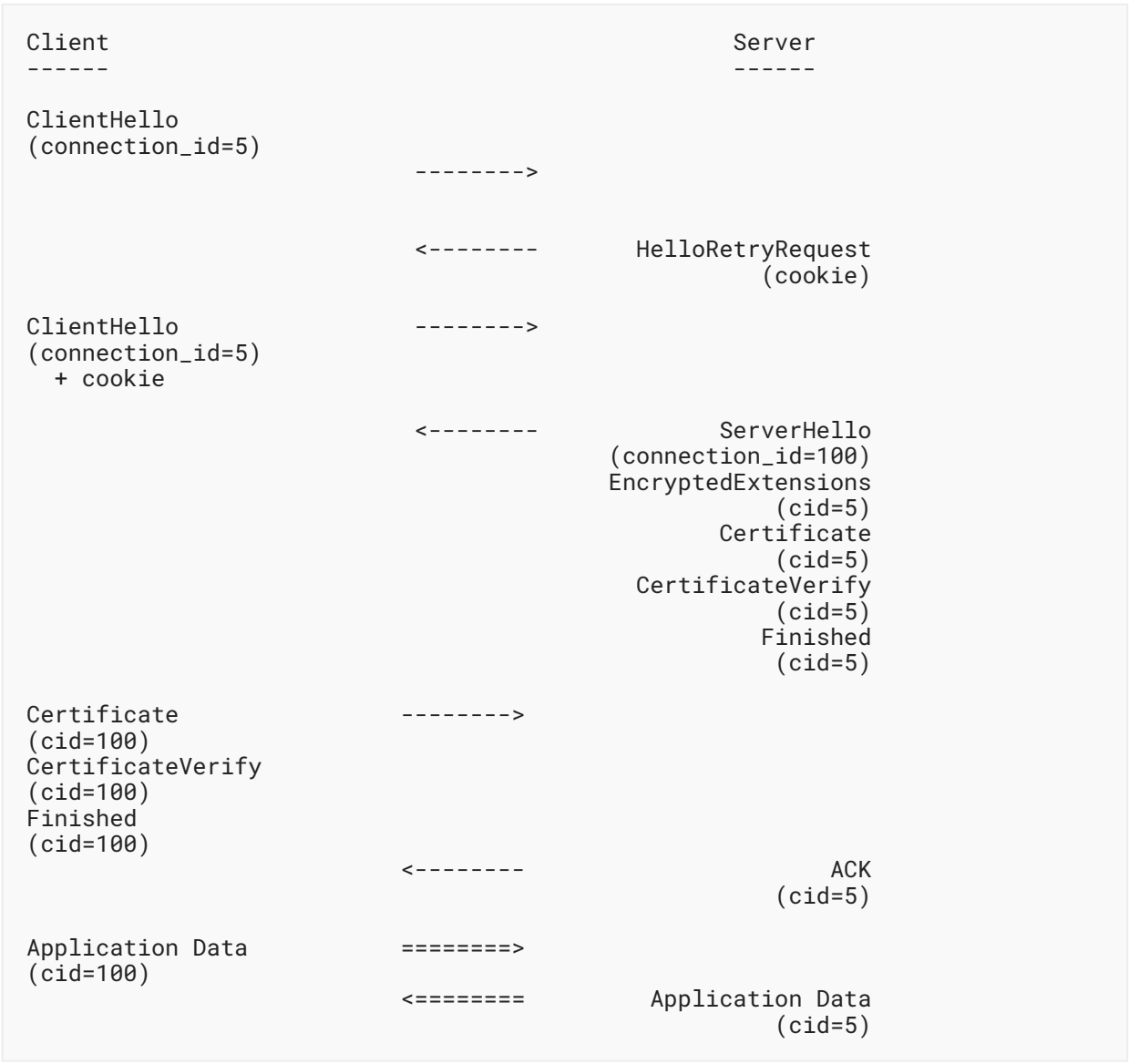


Figure 15: Example DTLS 1.3 Exchange with CIDs

If no CID is negotiated, then the receiver **MUST** reject any records it receives that contain a CID.

10. Application Data Protocol

Application data messages are carried by the record layer and are split into records and encrypted based on the current connection state. The messages are treated as transparent data to the record layer.

11. Security Considerations

Security issues are discussed primarily in [TLS13].

The primary additional security consideration raised by DTLS is that of denial of service by excessive resource consumption. DTLS includes a cookie exchange designed to protect against denial of service. However, implementations that do not use this cookie exchange are still vulnerable to DoS. In particular, DTLS servers that do not use the cookie exchange may be used as attack amplifiers even if they themselves are not experiencing DoS. Therefore, DTLS servers **SHOULD** use the cookie exchange unless there is good reason to believe that amplification is not a threat in their environment. Clients **MUST** be prepared to do a cookie exchange with every handshake.

Some key properties required of the cookie for the cookie-exchange mechanism to be functional are described in Section 3.3 of [RFC2522]:

- The cookie **MUST** depend on the client's address.
- It **MUST NOT** be possible for anyone other than the issuing entity to generate cookies that are accepted as valid by that entity. This typically entails an integrity check based on a secret key.
- Cookie generation and verification are triggered by unauthenticated parties, and as such their resource consumption needs to be restrained in order to avoid having the cookie-exchange mechanism itself serve as a DoS vector.

Although the cookie must allow the server to produce the right handshake transcript, it **SHOULD** be constructed so that knowledge of the cookie is insufficient to reproduce the ClientHello contents. Otherwise, this may create problems with future extensions such as Encrypted Client Hello [TLS-ECH].

When cookies are generated using a keyed authentication mechanism, it should be possible to rotate the associated secret key, so that temporary compromise of the key does not permanently compromise the integrity of the cookie-exchange mechanism. Though this secret is not as high-value as, e.g., a session-ticket-encryption key, rotating the cookie-generation key on a similar timescale would ensure that the key rotation functionality is exercised regularly and thus in working order.

The cookie exchange provides address validation during the initial handshake. DTLS with Connection IDs allows for endpoint addresses to change during the association; any such updated addresses are not covered by the cookie exchange during the handshake. DTLS implementations **MUST NOT** update the address they send to in response to packets from a different address unless they first perform some reachability test; no such test is defined in this specification and a future specification would need to specify a complete procedure for how and when to update addresses. Even with such a test, an active on-path adversary can also black-hole traffic or create a reflection attack against third parties because a DTLS peer has no means to

distinguish a genuine address update event (for example, due to a NAT rebinding) from one that is malicious. This attack is of concern when there is a large asymmetry of request/response message sizes.

With the exception of order protection and non-replayability, the security guarantees for DTLS 1.3 are the same as TLS 1.3. While TLS always provides order protection and non-replayability, DTLS does not provide order protection and may not provide replay protection.

Unlike TLS implementations, DTLS implementations **SHOULD NOT** respond to invalid records by terminating the connection.

TLS 1.3 requires replay protection for 0-RTT data (or rather, for connections that use 0-RTT data; see Section 8 of [TLS13]). DTLS provides an optional per-record replay-protection mechanism, since datagram protocols are inherently subject to message reordering and replay. These two replay-protection mechanisms are orthogonal, and neither mechanism meets the requirements for the other.

DTLS 1.3's handshake transcript does not include the new DTLS fields, which makes it have the same format as TLS 1.3. However, the DTLS 1.3 and TLS 1.3 transcripts are disjoint because they use different version numbers. Additionally, the DTLS 1.3 key schedule uses a different label and so will produce different keys for the same transcript.

The security and privacy properties of the CID for DTLS 1.3 build on top of what is described for DTLS 1.2 in [RFC9146]. There are, however, several differences:

- In both versions of DTLS, extension negotiation is used to agree on the use of the CID feature and the CID values. In both versions, the CID is carried in the DTLS record header (if negotiated). However, the way the CID is included in the record header differs between the two versions.
- The use of the post-handshake message allows the client and the server to update their CIDs, and those values are exchanged with confidentiality protection.
- The ability to use multiple CIDs allows for improved privacy properties in multihomed scenarios. When only a single CID is in use on multiple paths from such a host, an adversary can correlate the communication interaction across paths, which adds further privacy concerns. In order to prevent this, implementations **SHOULD** attempt to use fresh CIDs whenever they change local addresses or ports (though this is not always possible to detect). The RequestConnectionId message can be used by a peer to ask for new CIDs to ensure that a pool of suitable CIDs is available.
- The mechanism for encrypting sequence numbers (Section 4.2.3) prevents trivial tracking by on-path adversaries that attempt to correlate the pattern of sequence numbers received on different paths; such tracking could occur even when different CIDs are used on each path, in the absence of sequence number encryption. Switching CIDs based on certain events, or even regularly, helps against tracking by on-path adversaries. Note that sequence number encryption is used for all encrypted DTLS 1.3 records irrespective of whether a CID is used or not. Unlike the sequence number, the epoch is not encrypted because it acts as a key

identifier, which may improve correlation of packets from a single connection across different network paths.

- DTLS 1.3 encrypts handshake messages much earlier than in previous DTLS versions. Therefore, less information identifying the DTLS client, such as the client certificate, is available to an on-path adversary.

12. Changes since DTLS 1.2

Since TLS 1.3 introduces a large number of changes with respect to TLS 1.2, the list of changes from DTLS 1.2 to DTLS 1.3 is equally large. For this reason, this section focuses on the most important changes only.

- New handshake pattern, which leads to a shorter message exchange.
- Only AEAD ciphers are supported. Additional data calculation has been simplified.
- Removed support for weaker and older cryptographic algorithms.
- HelloRetryRequest of TLS 1.3 used instead of HelloVerifyRequest.
- More flexible cipher suite negotiation.
- New session resumption mechanism.
- PSK authentication redefined.
- New key derivation hierarchy utilizing a new key derivation construct.
- Improved version negotiation.
- Optimized record layer encoding and thereby its size.
- Added CID functionality.
- Sequence numbers are encrypted.

13. Updates Affecting DTLS 1.2

This document defines several changes that optionally affect implementations of DTLS 1.2, including those which do not also support DTLS 1.3.

- A version downgrade protection mechanism as described in [TLS13], Section 4.1.3 and applying to DTLS as described in Section 5.3.
- The updates described in [TLS13], Section 1.3.
- The new compliance requirements described in [TLS13], Section 9.3.

14. IANA Considerations

IANA has allocated the content type value 26 in the "TLS ContentType" registry for the ACK message, defined in Section 7. The value for the "DTLS-OK" column is "Y". IANA has reserved the content type range 32-63 so that content types in this range are not allocated.

IANA has allocated value 52 for the "too_many_cids_requested" alert in the "TLS Alerts" registry. The value for the "DTLS-OK" column is "Y".

IANA has allocated two values in the "TLS HandshakeType" registry, defined in [TLS13], for request_connection_id (9) and new_connection_id (10), as defined in this document. The value for the "DTLS-OK" column is "Y".

IANA has added this RFC as a reference to the "TLS Cipher Suites" registry along with the following Note:

> Any TLS cipher suite that is specified for use with DTLS **MUST** define limits on the use of the associated AEAD function that preserves margins for both confidentiality and integrity, as specified in Section 4.5.3 of RFC 9147.

15. References

15.1. Normative References

[CHACHA] Nir, Y. and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols", RFC 8439, DOI 10.17487/RFC8439, June 2018, <https://www.rfc-editor.org/info/rfc8439>.

[RFC0768] Postel, J., "User Datagram Protocol", STD 6, RFC 768, DOI 10.17487/RFC0768, August 1980, <https://www.rfc-editor.org/info/rfc768>.

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <https://www.rfc-editor.org/info/rfc793>.

[RFC1191] Mogul, J. and S. Deering, "Path MTU discovery", RFC 1191, DOI 10.17487/RFC1191, November 1990, <https://www.rfc-editor.org/info/rfc1191>.

[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>.

[RFC4443] Conta, A., Deering, S., and M. Gupta, Ed., "Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification", STD 89, RFC 4443, DOI 10.17487/RFC4443, March 2006, <https://www.rfc-editor.org/info/rfc4443>.

[RFC4821] Mathis, M. and J. Heffner, "Packetization Layer Path MTU Discovery", RFC 4821, DOI 10.17487/RFC4821, March 2007, <https://www.rfc-editor.org/info/rfc4821>.

[RFC6298] Paxson, V., Allman, M., Chu, J., and M. Sargent, "Computing TCP's Retransmission Timer", RFC 6298, DOI 10.17487/RFC6298, June 2011, <https://www.rfc-editor.org/info/rfc6298>.

[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>.

[RFC9146] Rescorla, E., Ed., Tschofenig, H., Ed., Fossati, T., and A. Kraus, "Connection Identifier for DTLS 1.2", RFC 9146, DOI 10.17487/RFC9146, March 2022, <https://www.rfc-editor.org/info/rfc9146>.

[TLS13] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

15.2. Informative References

[AEAD-LIMITS] Günther, F., Thomson, M., and C. A. Wood, "Usage Limits on AEAD Algorithms", Work in Progress, Internet-Draft, draft-irtf-cfrg-aead-limits-04, 7 March 2022, <https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-aead-limits-04>.

[AEBounds] Luykx, A. and K. Paterson, "Limits on Authenticated Encryption Use in TLS", 28 August 2017, <https://www.isg.rhul.ac.uk/~kp/TLS-AEbounds.pdf>.

[CCM-ANALYSIS] Jonsson, J., "On the Security of CTR + CBC-MAC", Selected Areas in Cryptography pp. 76-93, DOI 10.1007/3-540-36492-7_7, February 2003, <https://doi.org/10.1007/3-540-36492-7_7>.

[DEPRECATE] Moriarty, K. and S. Farrell, "Deprecating TLS 1.0 and TLS 1.1", BCP 195, RFC 8996, DOI 10.17487/RFC8996, March 2021, <https://www.rfc-editor.org/info/rfc8996>.

[IOT-PROFILE] Tschofenig, H. and T. Fossati, "TLS/DTLS 1.3 Profiles for the Internet of Things", Work in Progress, Internet-Draft, draft-ietf-uta-tls13-iot-profile-04, 7 March 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-uta-tls13-iot-profile-04>.

[RFC2522] Karn, P. and W. Simpson, "Photuris: Session-Key Management Protocol", RFC 2522, DOI 10.17487/RFC2522, March 1999, <https://www.rfc-editor.org/info/rfc2522>.

[RFC4303] Kent, S., "IP Encapsulating Security Payload (ESP)", RFC 4303, DOI 10.17487/RFC4303, December 2005, <https://www.rfc-editor.org/info/rfc4303>.

[RFC4340] Kohler, E., Handley, M., and S. Floyd, "Datagram Congestion Control Protocol (DCCP)", RFC 4340, DOI 10.17487/RFC4340, March 2006, <https://www.rfc-editor.org/info/rfc4340>.

[RFC4346] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1", RFC 4346, DOI 10.17487/RFC4346, April 2006, <https://www.rfc-editor.org/info/rfc4346>.

[RFC4347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security", RFC 4347, DOI 10.17487/RFC4347, April 2006, <https://www.rfc-editor.org/info/rfc4347>.

[RFC4960] Stewart, R., Ed., "Stream Control Transmission Protocol", RFC 4960, DOI 10.17487/RFC4960, September 2007, <https://www.rfc-editor.org/info/rfc4960>.

[RFC5238] Phelan, T., "Datagram Transport Layer Security (DTLS) over the Datagram Congestion Control Protocol (DCCP)", RFC 5238, DOI 10.17487/RFC5238, May 2008, <https://www.rfc-editor.org/info/rfc5238>.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <https://www.rfc-editor.org/info/rfc5246>.

[RFC5763] Fischl, J., Tschofenig, H., and E. Rescorla, "Framework for Establishing a Secure Real-time Transport Protocol (SRTP) Security Context Using Datagram Transport Layer Security (DTLS)", RFC 5763, DOI 10.17487/RFC5763, May 2010, <https://www.rfc-editor.org/info/rfc5763>.

[RFC5764] McGrew, D. and E. Rescorla, "Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP)", RFC 5764, DOI 10.17487/RFC5764, May 2010, <https://www.rfc-editor.org/info/rfc5764>.

[RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, DOI 10.17487/RFC6066, January 2011, <https://www.rfc-editor.org/info/rfc6066>.

[RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347, January 2012, <https://www.rfc-editor.org/info/rfc6347>.

[RFC7296] Kaufman, C., Hoffman, P., Nir, Y., Eronen, P., and T. Kivinen, "Internet Key Exchange Protocol Version 2 (IKEv2)", STD 79, RFC 7296, DOI 10.17487/RFC7296, October 2014, <https://www.rfc-editor.org/info/rfc7296>.

[RFC7525] Sheffer, Y., Holz, R., and P. Saint-Andre, "Recommendations for Secure Use of Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", BCP 195, RFC 7525, DOI 10.17487/RFC7525, May 2015, <https://www.rfc-editor.org/info/rfc7525>.

[RFC7924] Santesson, S. and H. Tschofenig, "Transport Layer Security (TLS) Cached Information Extension", RFC 7924, DOI 10.17487/RFC7924, July 2016, <https://www.rfc-editor.org/info/rfc7924>.

[RFC7983] Petit-Huguenin, M. and G. Salgueiro, "Multiplexing Scheme Updates for Secure Real-time Transport Protocol (SRTP) Extension for Datagram Transport Layer Security (DTLS)", RFC 7983, DOI 10.17487/RFC7983, September 2016, <https://www.rfc-editor.org/info/rfc7983>.

[RFC8201] McCann, J., Deering, S., Mogul, J., and R. Hinden, Ed., "Path MTU Discovery for IP version 6", STD 87, RFC 8201, DOI 10.17487/RFC8201, July 2017, <https://www.rfc-editor.org/info/rfc8201>.

[RFC8445] Keranen, A., Holmberg, C., and J. Rosenberg, "Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal", RFC 8445, DOI 10.17487/RFC8445, July 2018, <https://www.rfc-editor.org/info/rfc8445>.

[RFC8879] Ghedini, A. and V. Vasiliev, "TLS Certificate Compression", RFC 8879, DOI 10.17487/RFC8879, December 2020, <https://www.rfc-editor.org/info/rfc8879>.

[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/info/rfc9000>.

[RFC9002] Iyengar, J., Ed. and I. Swett, Ed., "QUIC Loss Detection and Congestion Control", RFC 9002, DOI 10.17487/RFC9002, May 2021, <https://www.rfc-editor.org/info/rfc9002>.

[ROBUST] Fischlin, M., Günther, F., and C. Janson, "Robust Channels: Handling Unreliable Networks in the Record Layers of QUIC and DTLS 1.3", received 15 June 2020, last revised 22 February 2021, <https://eprint.iacr.org/2020/718>.

[TLS-ECH] Rescorla, E., Oku, K., Sullivan, N., and C.A. Wood, "TLS Encrypted Client Hello", Work in Progress, Internet-Draft, draft-ietf-tls-esni-14, 13 February 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-tls-esni-14>.

Appendix A. Protocol Data Structures and Constant Values

This section provides the normative protocol types and constants definitions.

A.1. Record Layer

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

struct {
     opaque content[DTLSPlaintext.length];
     ContentType type;
     uint8 zeros[length_of_padding];
} DTLSInnerPlaintext;

struct {
    opaque unified_hdr[variable];
    opaque encrypted_record[length];
} DTLSCiphertext;

0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+
|0|0|1|C|S|L|E E|
+-+-+-+-+-+-+-+-+
| Connection ID |   Legend:
| (if any,      |
/  length as    /   C   - Connection ID (CID) present
|  negotiated)  |   S   - Sequence number length
+-+-+-+-+-+-+-+-+   L   - Length present
|  8 or 16 bit  |   E   - Epoch
|Sequence Number|
+-+-+-+-+-+-+-+-+
| 16 bit Length |
| (if present)  |
+-+-+-+-+-+-+-+-+

struct {
    uint64 epoch;
    uint64 sequence_number;
} RecordNumber;
```

A.2. Handshake Protocol

```
enum {
    hello_request_RESERVED(0),
    client_hello(1),
    server_hello(2),
    hello_verify_request_RESERVED(3),
    new_session_ticket(4),
    end_of_early_data(5),
    hello_retry_request_RESERVED(6),
```

```
        encrypted_extensions(8),
        request_connection_id(9),           /* New */
        new_connection_id(10),              /* New */
        certificate(11),
        server_key_exchange_RESERVED(12),
        certificate_request(13),
        server_hello_done_RESERVED(14),
        certificate_verify(15),
        client_key_exchange_RESERVED(16),
        finished(20),
        certificate_url_RESERVED(21),
        certificate_status_RESERVED(22),
        supplemental_data_RESERVED(23),
        key_update(24),
        message_hash(254),
        (255)
    } HandshakeType;

    struct {
        HandshakeType msg_type;    /* handshake type */
        uint24 length;             /* bytes in message */
        uint16 message_seq;        /* DTLS-required field */
        uint24 fragment_offset;    /* DTLS-required field */
        uint24 fragment_length;    /* DTLS-required field */
        select (msg_type) {
            case client_hello:          ClientHello;
            case server_hello:          ServerHello;
            case end_of_early_data:     EndOfEarlyData;
            case encrypted_extensions:  EncryptedExtensions;
            case certificate_request:   CertificateRequest;
            case certificate:           Certificate;
            case certificate_verify:    CertificateVerify;
            case finished:              Finished;
            case new_session_ticket:    NewSessionTicket;
            case key_update:            KeyUpdate;
            case request_connection_id: RequestConnectionId;
            case new_connection_id:     NewConnectionId;
        } body;
    } Handshake;

    uint16 ProtocolVersion;
    opaque Random[32];

    uint8 CipherSuite[2];    /* Cryptographic suite selector */

    struct {
        ProtocolVersion legacy_version = { 254,253 }; // DTLSv1.2
        Random random;
        opaque legacy_session_id<0..32>;
        opaque legacy_cookie<0..2^8-1>;                  // DTLS
        CipherSuite cipher_suites<2..2^16-2>;
        opaque legacy_compression_methods<1..2^8-1>;
        Extension extensions<8..2^16-1>;
    } ClientHello;
```

A.3. ACKs

```
struct {
    RecordNumber record_numbers<0..2^16-1>;
} ACK;
```

A.4. Connection ID Management

```
enum {
    cid_immediate(0), cid_spare(1), (255)
} ConnectionIdUsage;

opaque ConnectionId<0..2^8-1>;

struct {
    ConnectionId cids<0..2^16-1>;
    ConnectionIdUsage usage;
} NewConnectionId;

struct {
  uint8 num_cids;
} RequestConnectionId;
```

Appendix B. Analysis of Limits on CCM Usage

TLS [TLS13] and [AEBounds] do not specify limits on key usage for AEAD_AES_128_CCM. However, any AEAD that is used with DTLS requires limits on use that ensure that both confidentiality and integrity are preserved. This section documents that analysis for AEAD_AES_128_CCM.

[CCM-ANALYSIS] is used as the basis of this analysis. The results of that analysis are used to derive usage limits that are based on those chosen in [TLS13].

This analysis uses symbols for multiplication (*), division (/), and exponentiation (^), plus parentheses for establishing precedence. The following symbols are also used:

t: The size of the authentication tag in bits. For this cipher, t is 128.

n: The size of the block function in bits. For this cipher, n is 128.

l: The number of blocks in each packet (see below).

q: The number of genuine packets created and protected by endpoints. This value is the bound on the number of packets that can be protected before updating keys.

v: The number of forged packets that endpoints will accept. This value is the bound on the number of forged packets that an endpoint can reject before updating keys.

The analysis of AEAD_AES_128_CCM relies on a count of the number of block operations involved in producing each message. For simplicity, and to match the analysis of other AEAD functions in [AEBounds], this analysis assumes a packet length of 2^10 blocks and a packet size limit of 2^14 bytes.

For AEAD_AES_128_CCM, the total number of block cipher operations is the sum of: the length of the associated data in blocks, the length of the ciphertext in blocks, and the length of the plaintext in blocks, plus 1. In this analysis, this is simplified to a value of twice the maximum length of a record in blocks (that is, `2l = 2^11`). This simplification is based on the associated data being limited to one block.

B.1. Confidentiality Limits

For confidentiality, Theorem 2 in [CCM-ANALYSIS] establishes that an attacker gains a distinguishing advantage over an ideal pseudorandom permutation (PRP) of no more than:

```
(2l * q)^2 / 2^n
```

For a target advantage in a single-key setting of 2^-60, which matches that used by TLS 1.3, as summarized in [AEAD-LIMITS], this results in the relation:

```
q <= 2^23
```

That is, endpoints cannot protect more than 2^23 packets with the same set of keys without causing an attacker to gain a larger advantage than the target of 2^-60.

B.2. Integrity Limits

For integrity, Theorem 1 in [CCM-ANALYSIS] establishes that an attacker gains an advantage over an ideal PRP of no more than:

```
v / 2^t + (2l * (v + q))^2 / 2^n
```

The goal is to limit this advantage to 2^-57, to match the target in TLS 1.3, as summarized in [AEAD-LIMITS]. As `t` and `n` are both 128, the first term is negligible relative to the second, so that term can be removed without a significant effect on the result. This produces the relation:

```
v + q <= 2^24.5
```

Using the previously established value of 2^23 for `q` and rounding, this leads to an upper limit on `v` of 2^23.5. That is, endpoints cannot attempt to authenticate more than 2^23.5 packets with the same set of keys without causing an attacker to gain a larger advantage than the target of 2^-57.

B.3. Limits for AEAD_AES_128_CCM_8

The TLS_AES_128_CCM_8_SHA256 cipher suite uses the AEAD_AES_128_CCM_8 function, which uses a short authentication tag (that is, t=64).

The confidentiality limits of AEAD_AES_128_CCM_8 are the same as those for AEAD_AES_128_CCM, as this does not depend on the tag length; see Appendix B.1.

The shorter tag length of 64 bits means that the simplification used in Appendix B.2 does not apply to AEAD_AES_128_CCM_8. If the goal is to preserve the same margins as other cipher suites, then the limit on forgeries is largely dictated by the first term of the advantage formula:

```
v <= 2^7
```

As this represents attempts that fail authentication, applying this limit might be feasible in some environments. However, applying this limit in an implementation intended for general use exposes connections to an inexpensive denial-of-service attack.

This analysis supports the view that TLS_AES_128_CCM_8_SHA256 is not suitable for general use. Specifically, TLS_AES_128_CCM_8_SHA256 cannot be used without additional measures to prevent forgery of records, or to mitigate the effect of forgeries. This might require understanding the constraints that exist in a particular deployment or application. For instance, it might be possible to set a different target for the advantage an attacker gains based on an understanding of the constraints imposed on a specific usage of DTLS.

Appendix C. Implementation Pitfalls

In addition to the aspects of TLS that have been a source of interoperability and security problems (Appendix C.3 of [TLS13]), DTLS presents a few new potential sources of issues, noted here.

- Do you correctly handle messages received from multiple epochs during a key transition? This includes locating the correct key as well as performing replay detection, if enabled.
- Do you retransmit handshake messages that are not (implicitly or explicitly) acknowledged (Section 5.8)?
- Do you correctly handle handshake message fragments received, including when they are out of order?
- Do you correctly handle handshake messages received out of order? This may include either buffering or discarding them.
- Do you limit how much data you send to a peer before its address is validated?
- Do you verify that the explicit record length is contained within the datagram in which it is contained?

Contributors

Many people have contributed to previous DTLS versions, and they are acknowledged in prior versions of DTLS specifications or in the referenced specifications.

Hanno Becker
Arm Limited
Email: Hanno.Becker@arm.com

David Benjamin
Google
Email: davidben@google.com

Thomas Fossati
Arm Limited
Email: thomas.fossati@arm.com

Tobias Gondrom
Huawei
Email: tobias.gondrom@gondrom.org

Felix Günther
ETH Zurich
Email: mail@felixguenther.info

Benjamin Kaduk
Akamai Technologies
Email: kaduk@mit.edu

Ilari Liusvaara
Independent
Email: ilariliusvaara@welho.com

Martin Thomson
Mozilla
Email: martin.thomson@gmail.com

Christopher A. Wood
Cloudflare
Email: caw@heapingbits.net

Yin Xinxing
Huawei
Email: yinxinxing@huawei.com

The sequence number encryption concept is taken from QUIC [RFC9000]. We would like to thank the authors of RFC 9000 for their work. Felix Günther and Martin Thomson contributed the analysis in Appendix B. We would like to thank Jonathan Hammell, Bernard Aboba, and Andy Cunningham for their review comments.

Additionally, we would like to thank the IESG members for their review comments: Martin Duke, Erik Kline, Francesca Palombini, Lars Eggert, Zaheduzzaman Sarker, John Scudder, Éric Vyncke, Robert Wilton, Roman Danyliw, Benjamin Kaduk, Murray Kucherawy, Martin Vigoureux, and Alvaro Retana.

Authors' Addresses

Eric Rescorla
Mozilla
Email: ekr@rtfm.com

Hannes Tschofenig
Arm Limited
Email: hannes.tschofenig@arm.com

Nagendra Modadugu
Google, Inc.
Email: nagendra@cs.stanford.edu