CoRE Working Group
Internet-Draft
Updates: 7252, 8323 (if approved)
Intended status: Standards Track
Expires: May 1, 2020

K. Hartke
Ericsson

October 29, 2019

Extended Tokens and Stateless Clients in the Constrained Application Protocol (CoAP) draft-ietf-core-stateless-03

Abstract

This document provides considerations for alleviating CoAP clients and intermediaries of keeping per-request state. To facilitate this, this document additionally introduces a new, optional CoAP protocol extension for extended token lengths.

This document updates RFCs 7252 and 8323.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

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

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

This Internet-Draft will expire on May 1, 2020.

Table of Contents

1. Introduction

The Constrained Application Protocol (CoAP) [RFC7252] is a RESTful application-layer protocol for constrained environments [RFC7228]. In CoAP, clients (or intermediaries in the client role) make requests to servers (or intermediaries in the server role), which satisfy the requests by returning responses.

While a request is ongoing, a client typically needs to keep some state that it requires for processing the response when it arrives. Identification of this state is done by means of a _token_ in CoAP, an opaque sequence of bytes chosen by the client and included in the CoAP request, which is returned by the server verbatim in any resulting CoAP response (Figure 1).

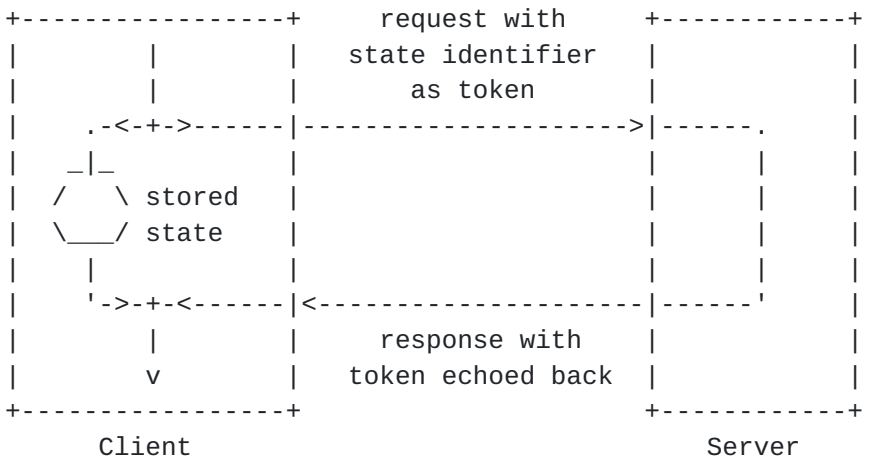


Figure 1: Token as an Identifier for Request State

In some scenarios, it can be beneficial to reduce the amount of state that is stored at the client at the cost of increased message sizes. A client can opt into this by serializing (parts of) its state into the token itself and then recovering this state from the token in the response (Figure 2).

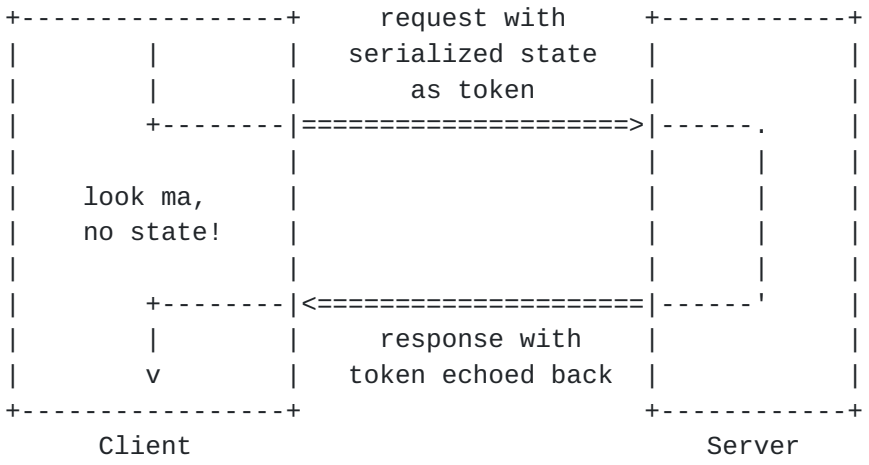


Figure 2: Token as Serialization of Request State

Section 3 of this document provides considerations for clients opting into becoming "stateless" in this way. (As those considerations will show, the term "stateless" is not entirely accurate. The clients still need to maintain per-server state and other kinds of state. So it is more accurate to say the these clients are just avoiding per-request state.)

Serializing state into tokens is complicated by the fact that both CoAP over UDP [RFC7252] and CoAP over reliable transports [RFC8323] limit the maximum token length to 8 bytes. To overcome this

limitation, Section 2 of this document first introduces a CoAP protocol extension for extended token lengths.

While the use case (avoiding per-request state) and the mechanism (extended token lengths) presented in this document are closely related, both can be used independently of each other: Some implementations may be able to fit their state in just 8 bytes; some implementations may have other use cases for extended token lengths.

1.1. Terminology

In this document, the term "stateless" refers to an implementation strategy for a client (or intermediary in the client role) that does not require it to keep state for the individual requests it sends to a server (or intermediary in the server role). The client still needs to keep state for each server it communicates with (e.g., for token generation, message retransmission, and congestion control).

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

2. Extended Tokens

This document updates the message formats defined for CoAP over UDP [RFC7252] and CoAP over TCP, TLS, and WebSockets [RFC8323] with a new definition of the TKL field.

2.1. Extended Token Length (TKL) Field

The definition of the TKL field is updated as follows:

Token Length (TKL): 4-bit unsigned integer. A value between 0 and 12 inclusive indicates the length of the variable-length Token field in bytes. Three values are reserved for special constructs:

13: An 8-bit unsigned integer precedes the Token field and indicates the length of the Token field minus 13.

14: A 16-bit unsigned integer in network byte order precedes the Token field and indicates the length of the Token field minus 269.

15: Reserved. This value MUST NOT be sent and MUST be processed as a message format error.

This increases the maximum token length that can be represented in a message to 65804 bytes. The maximum token length that sender and recipient implementations support may be shorter. For example, a constrained node of Class 1 [RFC7228] might support extended token lengths only up to 32 bytes.

All other fields retain their definition.

The updated message formats are illustrated in Appendix A.

2.2. Discovering Support

Extended token lengths require support from the server. Support can be discovered by a client in one of two ways:

- Where Capabilities and Settings Messages (CSMs) are available, such as in CoAP over TCP, support can be discovered using the Extended-Token-Length Capability Option defined in Section 2.2.1.

- Otherwise, such as in CoAP over UDP, support can only be discovered by trial-and-error, as described in Section 2.2.2.

2.2.1. Extended-Token-Length Capability Option

A server can use the elective Extended-Token-Length Capability Option to indicate the maximum token length it can accept in requests.

#	C	R	Applies to	Name	Format	Length	Base Value
TBD			CSM	Extended-Token-Length	uint	0-3	8

C=Critical, R=Repeatable

Table 1: The Extended-Token-Length Capability Option

As per Section 3 of RFC 7252, the base value (and the value used when this option is not implemented) is 8.

The active value of the Extended-Token-Length Option is replaced each time the option is sent with a modified value. Its starting value is its base value.

The option value MUST NOT be less than 8 or greater than 65804. If an option value outside this range is received, the value MUST be clamped to this range.

Any option value greater than 8 implies support for the new definition of the TKL field specified in Section 2.1. Indication of support by a server does not oblige a client to actually make use of token lengths greater than 8.

If a server receives a request with a token of a length greater than it indicated in its Extended-Token-Length Option, it MUST handle the request as a message format error.

Note: The Extended-Token-Length Capability Option does not apply to responses. The sender of a request is simply expected not to send a token of a length greater than it is willing to accept in a response.

2.2.2. Trial-and-Error

A server that does not support the updated definition of the TKL field specified in Section 2.1 will consider a request with a TKL field value outside the range 0 to 8 a message format error and rejected it (Section 3 of RFC 7252). A client can therefore determine support by sending a request with an extended token length and checking whether it's rejected by the server or not.

In CoAP over UDP, a Confirmable message with a message format error is rejected with a Reset message (Section 4.2 of RFC 7252). A Non-confirmable message with a message format error may be rejected with a Reset message or just silently ignored (Section 4.3 of RFC 7252). It is therefore RECOMMENDED that clients use a Confirmable message for determining support.

As per RFC 7252, Reset messages are empty and do not contain a token; they only return the Message ID (Figure 3). They also do not contain any indication of what caused a message format error. To avoid any ambiguity, it is therefore RECOMMENDED that clients use a request that has no potential message format error other than the extended token length.

An example of a suitable request is a Confirmable GET request that includes an If-None-Match option and a token of the greatest length that the client intends to use. Any response with the same token echoed back indicates that tokens up to that length are supported by the server.

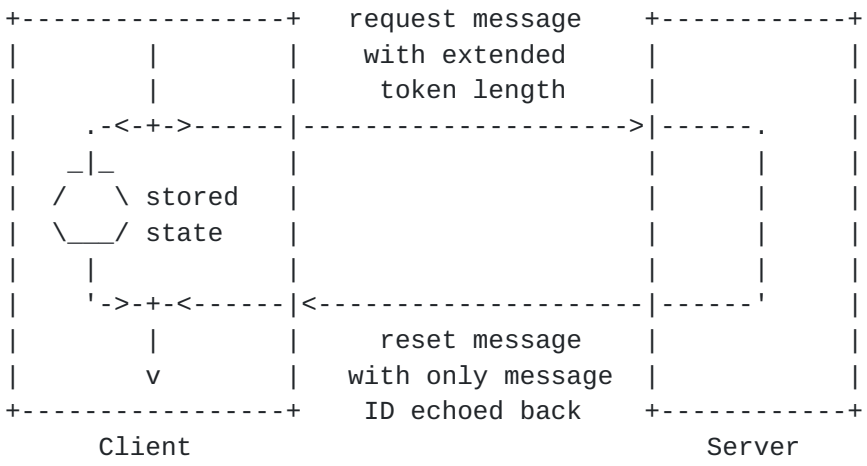


Figure 3: A Confirmable Request With an Extended Token is Rejected With a Reset Message if the Server Does Not Have Support

A client SHOULD NOT assume that extended token lengths are supported by a server 60 minutes after receiving a response with an extended token length, as network addresses may change.

If a server supports extended token lengths but receives a request with a token of a length it is unwilling or unable to handle, it MUST NOT reject the message, as that would imply that extended token lengths are not supported at all. Instead, if the server cannot handle the request at the time, it SHOULD return a 5.03 (Service Unavailable) response; if the server will never be able to handle (e.g., because the token is too large), it SHOULD return a 4.00 (Bad Request) response.

Design Note: The requirement to return an error response when a request cannot be handled might seem somewhat contradictory. However, handling a request usually involves a number of steps from receiving the message to handing it over to application logic. The idea is that a server implementing this document at least should support large tokens in the first few steps, enough to return an error response rather than a Reset message.

Design Note: For simplicity, no mechanism to indicate the maximum supported token length is defined: A client implementation would probably already choose the shortest token possible for the task (such as being stateless, as described in Section 3), so it probably would not be able to reduce the length any further anyway, should a server indicate a lower limit.

2.3. Intermediaries

Tokens are a hop-by-hop feature: If there are one or more intermediaries between a client and a server, every token is scoped to the exchange between a node in the client role and the node in the server role that it is immediately interacting with.

When an intermediary receives a request, the only requirement is that it echoes the token back in any resulting response. There is no requirement or expectation that an intermediary passes a client's token on to a server or that an intermediary uses extended token lengths itself in a request to satisfy a request with an extended token length. Discovery needs to be performed for each hop.

3. Stateless Clients

A client can be alleviated of keeping per-request state by serializing the state into a sequence of bytes and sending the bytes as the token of the request. The server returns the token verbatim in the response to the client, which allows the client to recover the state and process the response as if it had kept the state locally.

The format of the serialized state is generally an implementation detail of the client and opaque to any server. However, transporting client state in requests and responses has significant security and privacy implications that need to be taken into consideration by a client implementation.

Furthermore, there are several non-obvious implications from CoAP protocol features that should be taken into consideration by a client implementation.

The following subsections discuss some of these considerations.

3.1. Serializing Client State

Serialized state information is an attractive target for both unwanted nodes (attackers between the node in client role and the node in server role) and wanted nodes (the node in server role itself) on the path. Therefore, a node in the client role SHOULD integrity protect the state information, unless processing a response does not modify state or cause any other significant side effects.

Even when the serialized state is integrity protected, an attacker may still replay a response, making the client believe it sent the same request twice. Therefore, the node in client role SHOULD implement replay protection (e.g., by using sequence numbers and a replay window), unless processing a response does not modify state or

cause other any significant side effects. Integrity protection is REQUIRED for replay protection.

If processing a response without keeping request state is sensitive to the time elapsed to sending the request, then the serialized state SHOULD include freshness information (e.g., a timestamp).

Information in the serialized state may be privacy sensitive. A node in client role SHOULD encrypt the serialized state if it contains privacy sensitive information that an attacker would not get otherwise. For example, an intermediary that serializes client information into its token leaks that information to the node in server role, which may be undesirable.

3.2. Stateless Intermediaries

Tokens are a hop-by-hop feature: If a client makes a request to an intermediary, that intermediary needs to store the client's token (along with the client's transport address) while it makes its own request towards the origin server and waits for the response. When the intermediary receives the response, it looks up the client's token and transport address for the received request and sends an appropriate response to the client.

Such an intermediary might want to be "stateless" as well, i.e., be alleviated of storing the client's token and transport address for received requests. This can be implemented by serializing this information along the request state into the token towards the origin server. When the intermediary receives the response, it can recover the information from the token and use it to satisfy the client's request.

The drawback of this approach is that an intermediary, without keeping request state, is unable to aggregate multiple requests for the same target resource, which can significantly reduce efficiency.

In particular, when multiple clients observe [RFC7641] the same resource, aggregating requests is REQUIRED (Section 3.1 of RFC 7641). This requirement cannot be satisfied without keeping request state; therefore, an intermediary MUST NOT include an Observe Option in requests it sends without keeping request state.

When using block-wise transfers [RFC7959], a server might not be able to distinguish blocks originating from different clients once they have been forwarded by an intermediary. To ensure that this does not lead to inconsistent resource state, a stateless intermediary MUST include the Request-Tag Option [I-D.ietf-core-echo-request-tag] in

block-wise transfers with a value that uniquely identifies the client in the intermediary's namespace.

3.3. Extended Tokens

A client (or intermediary in the role of a client) that depends on support for extended token lengths (Section 2) from the server (or intermediary in the role of a server) to avoid keeping request state SHOULD perform a discovery of support (Section 2.2) before it can be stateless.

This discovery MUST be performed in a stateful way, i.e., keeping state for the request (Figure 4): If the client was stateless from the start and the server does not support extended tokens, then any error message could not be processed since the state would neither be present at the client nor returned in the Reset message (Figure 5).

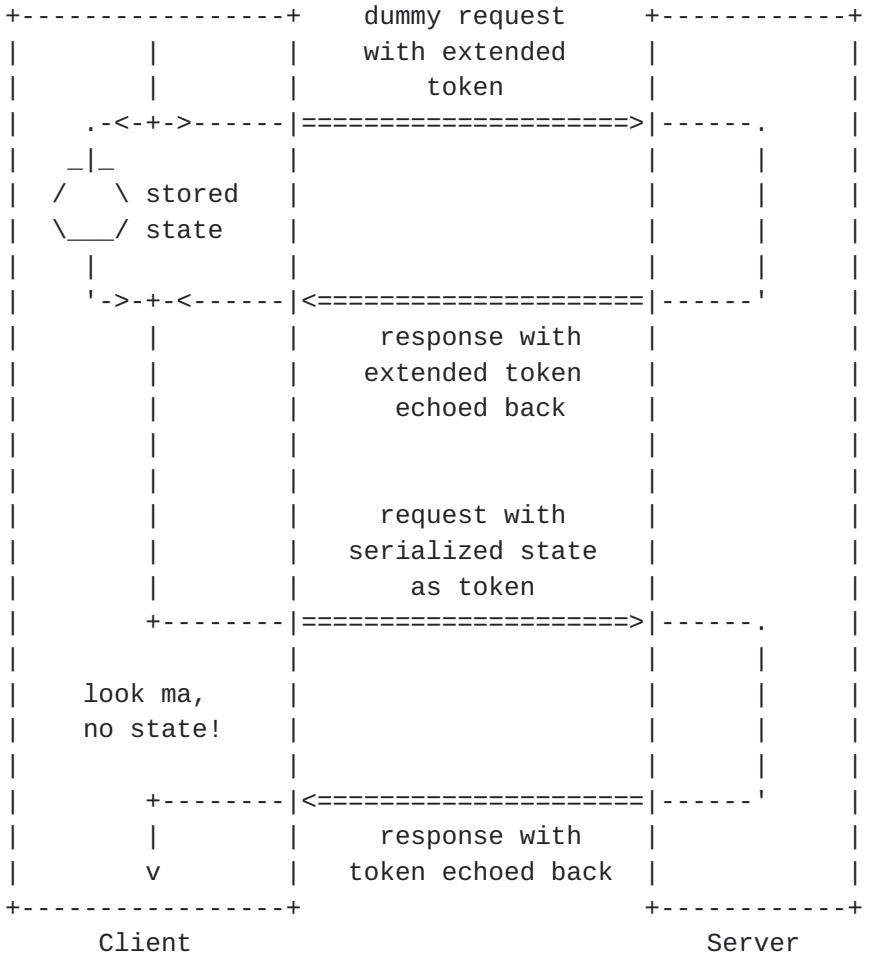


Figure 4: Depending on Extended Tokens for Being Stateless First Requires a Successful Stateful Discovery of Support

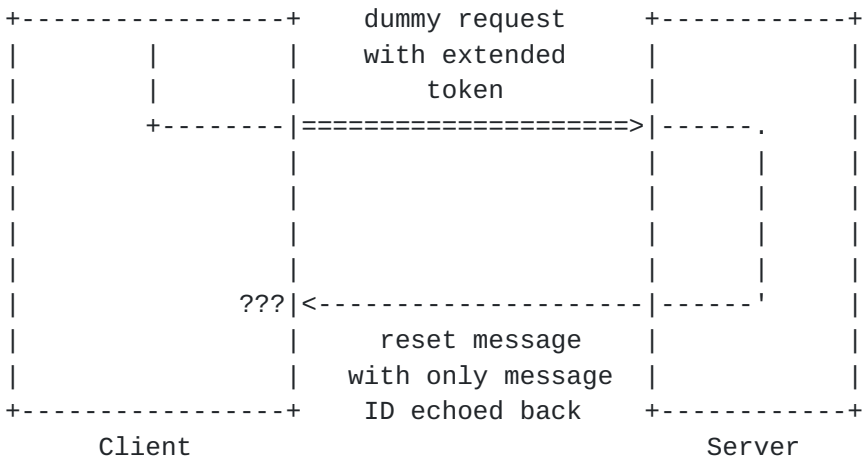


Figure 5: Stateless Discovery of Support Does Not Work

In environments where support can be reliably discovered through some other means, the discovery of support is OPTIONAL. An example for this is the Constrained Join Protocol (CoJP) in a 6TiSCH network [I-D.ietf-6tisch-minimal-security], where support for extended tokens is required from all relevant parties.

3.4. Stateless Message Transmission

As a further step, a client (or intermediary in the client role) might want to also avoid keeping message transmission state when using CoAP over UDP [RFC7252].

Generally, a client can use Confirmable or Non-confirmable messages for requests. When using Confirmable messages, it needs to keep message exchange state for performing retransmissions and handling Acknowledgement and Reset messages. When using Non-confirmable messages, it can keep no message exchange state. (However, in either case the client needs to keep congestion control state. That is, it needs to maintain state for each node it communicates with and, e.g., enforce NSTART.)

As per RFC 7252, a client must be prepared to receive a response as a piggybacked response, a separate response or Non-confirmable response (Section 5.2 of RFC 7252), regardless of the message type used for the request. A stateless client MUST handle these response types as follows:

- If a piggybacked response passes the token integrity protection and freshness checks, the client processes the message as specified in RFC 7252; otherwise, it silently discards the message.

- If a separate response passes the token integrity protection and freshness checks, the client processes the message as specified in RFC 7252; otherwise, it rejects the message as specified in Section 4.2 of RFC 7252.

- If a Non-confirmable response passes the token integrity protection and freshness checks, the client processes the message as specified in RFC 7252; otherwise, it rejects the message as specified in Section 4.3 of RFC 7252.

4. Security Considerations

4.1. Extended Tokens

Tokens significantly larger than the 8 bytes specified in RFC 7252 have implications in particular for nodes with constrained memory size that need to be mitigated. A node in the server role supporting extended token lengths may be vulnerable to a denial-of-service when an attacker (either on-path or a malicious client) sends large tokens to fill up the memory of the node. Implementations should be prepared to handle such messages.

4.2. Stateless Clients

Transporting the state needed by a client to process a response as serialized state information in the token has several significant and non-obvious security and privacy implications that need to be mitigated; see Section 3.1.

The use of encryption, integrity protection, and replay protection of serialized state is recommended in general, unless a careful analysis of any potential attacks to security and privacy is performed. AES-CCM with a 64 bit tag is recommended, combined with a sequence number and a replay window. Where encryption is not needed, HMAC-SHA-256, combined with a sequence number and a replay window, may be used.

When using AES-CCM, repeated use of the same nonce under the same key causes the cipher to fail catastrophically. If a nonce is ever used for more than one encryption operation with the same key, then the same key stream gets used to encrypt both plaintexts and the confidentiality guarantees are voided. Devices with low-entropy sources -- as is typical with constrained devices, which incidentally happen to be a natural candidate for the stateless mechanism described in this document -- need to carefully pick a nonce generation mechanism that provides the above uniqueness guarantee. Additionally, since it can be difficult to use AES-CCM securely when using statically configured keys, implementations should use automated key management [RFC4107].

5. IANA Considerations

5.1. CoAP Signaling Option Number

The following entries are added to the "CoAP Signaling Option Numbers" registry within the "CoRE Parameters" registry.

Applies to	Number	Name	Reference
7.01	TBD	Extended-Token-Length	[[this document]]

6. References

6.1. Normative References

[I-D.ietf-core-echo-request-tag]
Amsuess, C., Mattsson, J., and G. Selander, "CoAP: Echo, Request-Tag, and Token Processing", draft-ietf-core-echo-request-tag-07 (work in progress), September 2019.

[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>.

[RFC4107] Bellovin, S. and R. Housley, "Guidelines for Cryptographic Key Management", BCP 107, RFC 4107, DOI 10.17487/RFC4107, June 2005, <https://www.rfc-editor.org/info/rfc4107>.

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

[RFC7641] Hartke, K., "Observing Resources in the Constrained Application Protocol (CoAP)", RFC 7641, DOI 10.17487/RFC7641, September 2015, <https://www.rfc-editor.org/info/rfc7641>.

[RFC7959] Bormann, C. and Z. Shelby, Ed., "Block-Wise Transfers in the Constrained Application Protocol (CoAP)", RFC 7959, DOI 10.17487/RFC7959, August 2016, <https://www.rfc-editor.org/info/rfc7959>.

[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>.

[RFC8323] Bormann, C., Lemay, S., Tschofenig, H., Hartke, K., Silverajan, B., and B. Raymor, Ed., "CoAP (Constrained Application Protocol) over TCP, TLS, and WebSockets", RFC 8323, DOI 10.17487/RFC8323, February 2018, <https://www.rfc-editor.org/info/rfc8323>.

6.2. Informative References

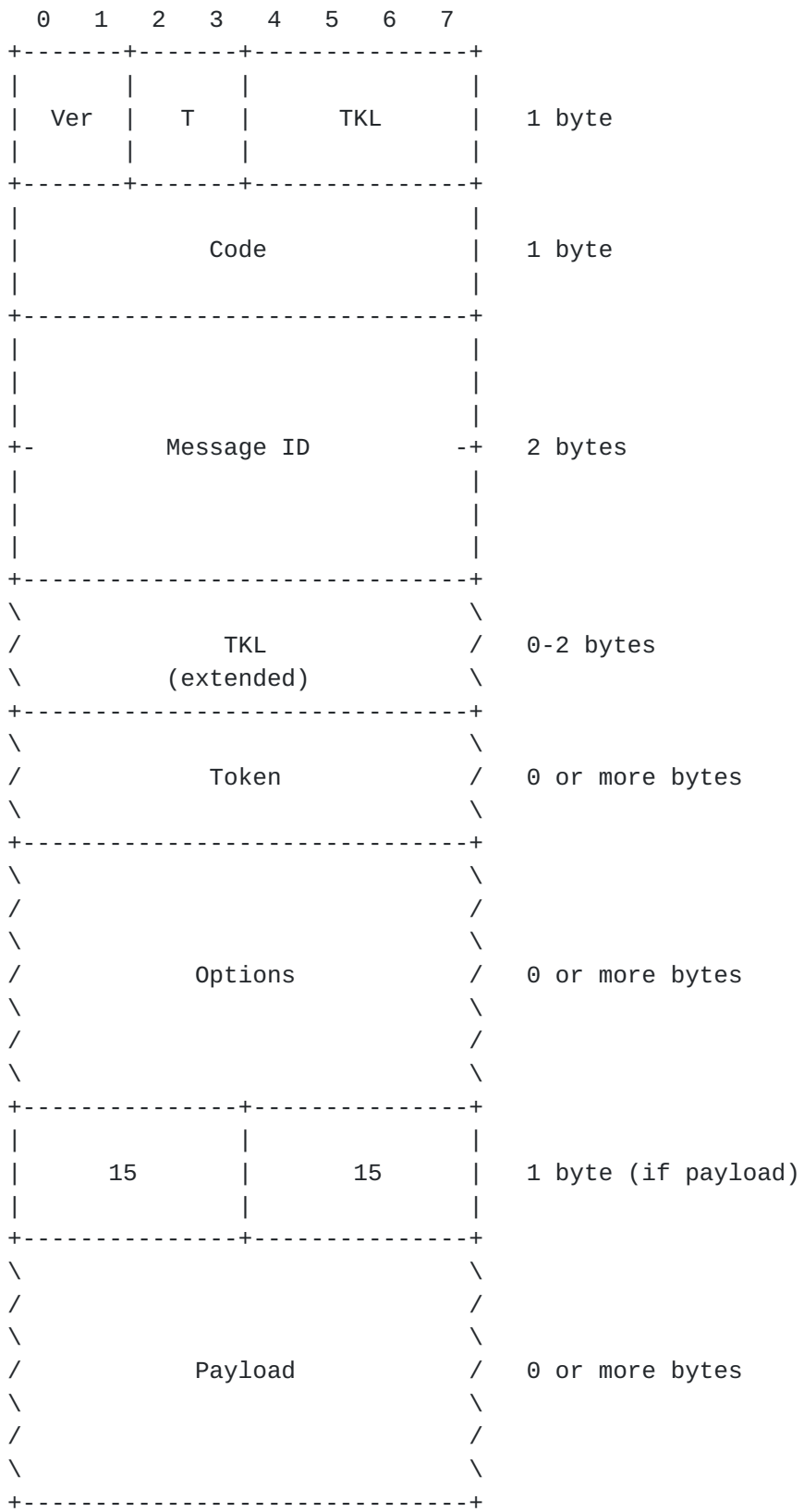
[I-D.ietf-6tisch-minimal-security] Vucinic, M., Simon, J., Pister, K., and M. Richardson, "Minimal Security Framework for 6TiSCH", draft-ietf-6tisch-minimal-security-13 (work in progress), October 2019.

[RFC7228] Bormann, C., Ersue, M., and A. Keranen, "Terminology for Constrained-Node Networks", RFC 7228, DOI 10.17487/RFC7228, May 2014, <https://www.rfc-editor.org/info/rfc7228>.

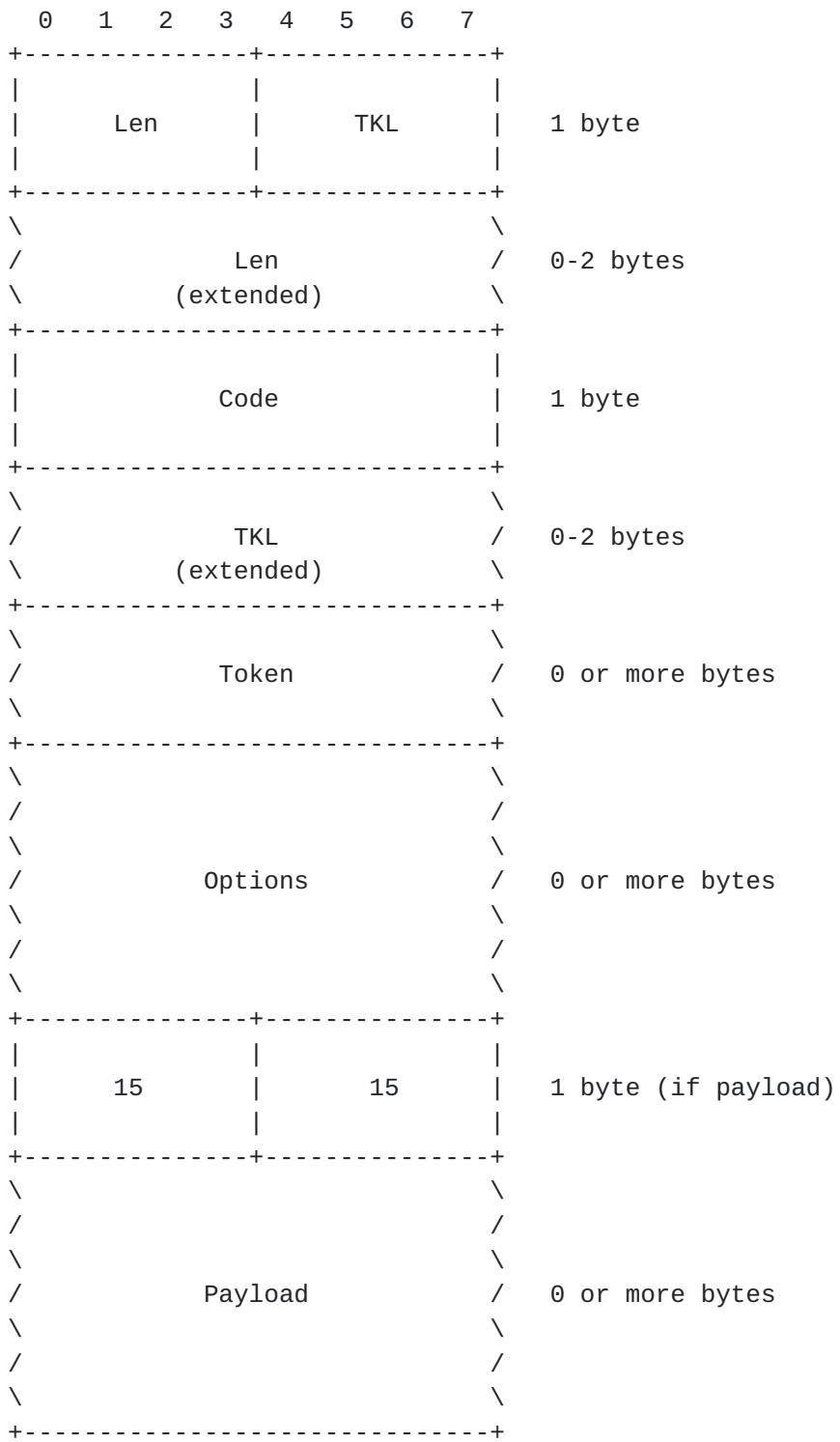
Appendix A. Updated Message Formats

This appendix illustrates the CoAP message formats updated with the new definition of the TKL field (Section 2).

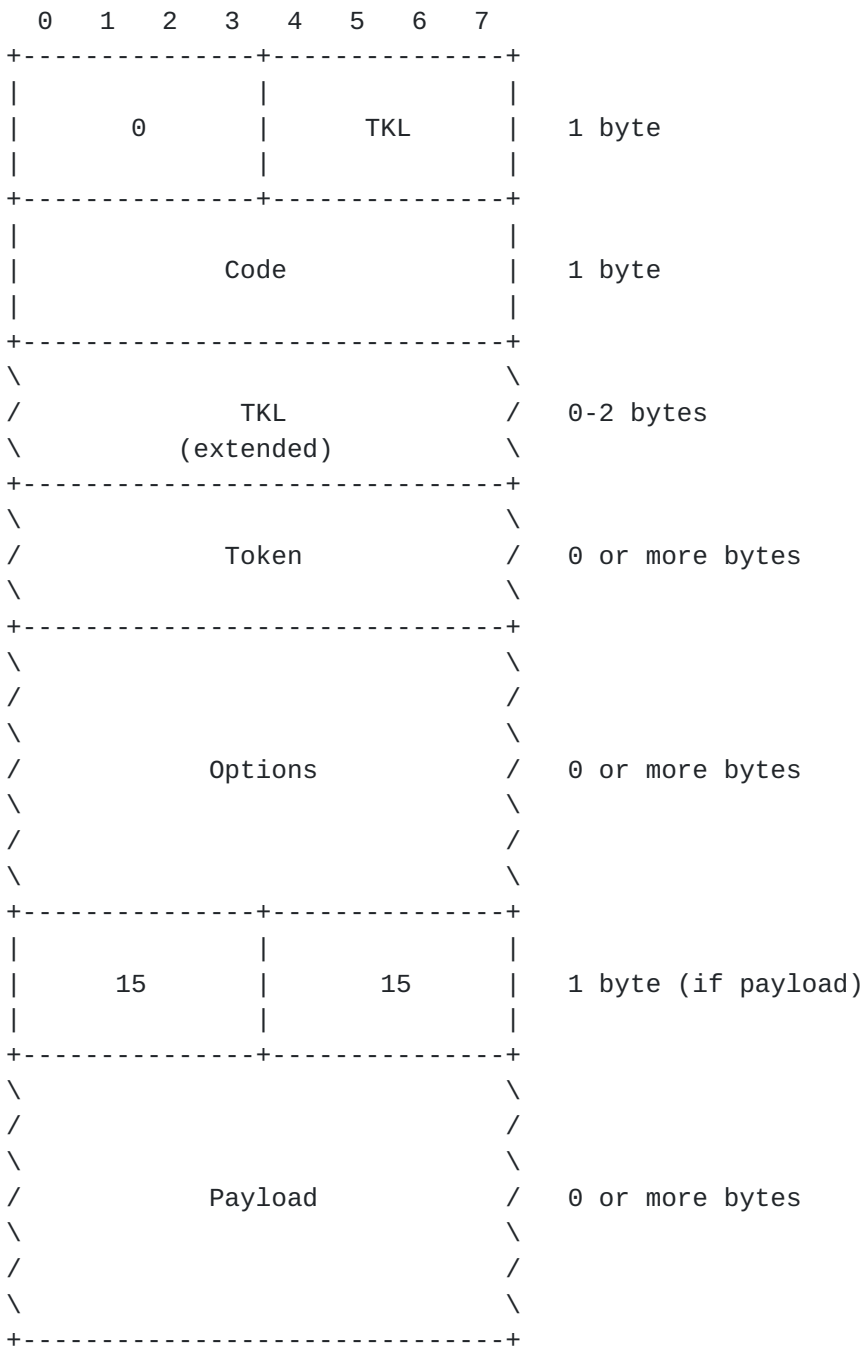
A.1. CoAP over UDP



A.2. CoAP over TCP



A.3. CoAP over WebSockets



Acknowledgements

This document is based on the requirements of and work on the Minimal Security Framework for 6TiSCH [I-D.ietf-6tisch-minimal-security] by Malisa Vucinic, Jonathan Simon, Kris Pister, and Michael Richardson.

Thanks to Christian Amsuess, Carsten Bormann, Ari Keranen, John Mattsson, Jim Schaad, Goeran Selander, and Malisa Vucinic for helpful comments and discussions that have shaped the document.

Special thanks to Thomas Fossati for his in-depth review, copious comments, and suggested text.

Author's Address

Klaus Hartke
Ericsson
Torshamnsgatan 23
Stockholm SE-16483
Sweden

Email: klaus.hartke@ericsson.com