CoRE Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 4, 2013

Z. Shelby
Sensinode
K. Hartke
C. Bormann
Universitaet Bremen TZI
B. Frank
SkyFoundry

October 1, 2012

Constrained Application Protocol (CoAP)
draft-ietf-core-coap-12

Abstract

The Constrained Application Protocol (CoAP) is a specialized web transfer protocol for use with constrained nodes and constrained (e.g., low-power, lossy) networks. The nodes often have 8-bit microcontrollers with small amounts of ROM and RAM, while constrained networks such as 6LoWPAN often have high packet error rates and a typical throughput of 10s of kbit/s. The protocol is designed for machine-to-machine (M2M) applications such as smart energy and building automation.

CoAP provides a request/response interaction model between application endpoints, supports built-in discovery of services and resources, and includes key concepts of the Web such as URIs and Internet media types. CoAP easily interfaces with HTTP for integration with the Web while meeting specialized requirements such as multicast support, very low overhead and simplicity for constrained environments.

Status of this Memo

Copyright Notice

Table of Contents

1. Introduction

The use of web services on the Internet has become ubiquitous in most applications, and depends on the fundamental Representational State Transfer [REST] architecture of the web.

The Constrained RESTful Environments (CoRE) work aims at realizing the REST architecture in a suitable form for the most constrained nodes (e.g. 8-bit microcontrollers with limited RAM and ROM) and networks (e.g. 6LoWPAN, [RFC4944]). Constrained networks like 6LoWPAN support the expensive fragmentation of IPv6 packets into small link-layer frames. One design goal of CoAP has been to keep message overhead small, thus limiting the use of fragmentation.

One of the main goals of CoAP is to design a generic web protocol for the special requirements of this constrained environment, especially considering energy, building automation and other machine-to-machine (M2M) applications. The goal of CoAP is not to blindly compress HTTP [RFC2616], but rather to realize a subset of REST common with HTTP but optimized for M2M applications. Although CoAP could be used for compressing simple HTTP interfaces, it more importantly also offers features for M2M such as built-in discovery, multicast support and asynchronous message exchanges.

This document specifies the Constrained Application Protocol (CoAP), which easily translates to HTTP for integration with the existing web while meeting specialized requirements such as multicast support, very low overhead and simplicity for constrained environments and M2M applications.

1.1. Features

CoAP has the following main features:

- Constrained web protocol fulfilling M2M requirements.
- UDP binding with optional reliability supporting unicast and multicast requests.
- Asynchronous message exchanges.
- Low header overhead and parsing complexity.
- URI and Content-type support.
- Simple proxy and caching capabilities.

- A stateless HTTP mapping, allowing proxies to be built providing access to CoAP resources via HTTP in a uniform way or for HTTP simple interfaces to be realized alternatively over CoAP.

- Security binding to Datagram Transport Layer Security (DTLS).

1.2. 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 [RFC2119] when they appear in ALL CAPS. These words may also appear in this document in lower case as plain English words, absent their normative meanings.

This specification requires readers to be familiar with all the terms and concepts that are discussed in [RFC2616]. In addition, this specification defines the following terminology:

Endpoint
: An entity participating in the CoAP protocol. Colloquially, an endpoint lives on a "Node", although "Host" would be more consistent with Internet standards usage, and is further identified by transport layer multiplexing information that can include a UDP port number and a security association (Section 4.1).

Sender
: The originating endpoint of a message. When the aspect of identification of the specific sender is in focus, also "source endpoint".

Recipient
: The destination endpoint of a message. When the aspect of identification of the specific recipient is in focus, also "destination endpoint".

Client
: The originating endpoint of a request; the destination endpoint of a response.

Server
: The destination endpoint of a request; the originating endpoint of a response.

Origin Server
The server on which a given resource resides or is to be created.

Intermediary
A CoAP endpoint that acts both as a server and as a client towards (possibly via further intermediaries) an origin server. A common form of an intermediary is a proxy; several classes of such proxies are discussed in this specification.

Proxy
An intermediary that mainly is concerned with forwarding requests and relaying back responses, possibly performing caching, namespace translation, or protocol translation in the process. As opposed to intermediaries in the general sense, proxies generally do not implement specific application semantics. Based on the position in the overall structure of the request forwarding, there are two common forms of proxy: forward-proxy and reverse-proxy. In some cases, a single endpoint might act as an origin server, forward-proxy, or reverse-proxy, switching behavior based on the nature of each request.

Forward-Proxy
A "forward-proxy" is an endpoint selected by a client, usually via local configuration rules, to perform requests on behalf of the client, doing any necessary translations. Some translations are minimal, such as for proxy requests for "coap" URIs, whereas other requests might require translation to and from entirely different application-layer protocols.

Reverse-Proxy
A "reverse-proxy" is an endpoint that stands in for one or more other server(s) and satisfies requests on behalf of these, doing any necessary translations. Unlike a forward-proxy, the client may not be aware that it is communicating with a reverse-proxy; a reverse-proxy receives requests as if it was the origin server for the target resource.

Cross-Proxy
A cross-protocol proxy, or "cross-proxy" for short, is a proxy that translates between different protocols, such as a CoAP-to-HTTP proxy or an HTTP-to-CoAP proxy. While this specification makes very specific demands of CoAP-to-CoAP proxies, there is more variation possible in cross-proxies.

Confirmable Message
Some messages require an acknowledgement. These messages are called "Confirmable". When no packets are lost, each confirmable message elicits exactly one return message of type Acknowledgement

or type Reset.

Non-Confirmable Message
Some other messages do not require an acknowledgement. This is particularly true for messages that are repeated regularly for application requirements, such as repeated readings from a sensor where eventual success is sufficient.

Acknowledgement Message
An Acknowledgement message acknowledges that a specific Confirmable Message arrived. It does not indicate success or failure of any encapsulated request.

Reset Message
A Reset message indicates that a specific message (confirmable or non-confirmable) was received, but some context is missing to properly process it. This condition is usually caused when the receiving node has rebooted and has forgotten some state that would be required to interpret the message.

Piggy-backed Response
A Piggy-backed Response is included right in a CoAP Acknowledgement (ACK) message that is sent to acknowledge receipt of the Request for this Response (Section 5.2.1).

Separate Response
When a Confirmable message carrying a Request is acknowledged with an empty message (e.g., because the server doesn’t have the answer right away), a Separate Response is sent in a separate message exchange (Section 5.2.2).

Critical Option
An option that would need to be understood by the endpoint receiving the message in order to properly process the message (Section 5.4.1). Note that the implementation of critical options is, as the name "Option" implies, generally optional: unsupported critical options lead to an error response or summary rejection of the message.

Elective Option
An option that is intended to be ignored by an endpoint that does not understand it. Processing the message even without understanding the option is acceptable (Section 5.4.1).

Unsafe Option
An option that would need to be understood by a proxy receiving the message in order to safely forward the message (Section 5.4.2).

Safe Option
An option that is intended to be safe for forwarding by a proxy that does not understand it. Forwarding the message even without understanding the option is acceptable (Section 5.4.2).

Resource Discovery
The process where a CoAP client queries a server for its list of hosted resources (i.e., links, Section 7).

Content-Format
The combination of an Internet media type, potentially with specific parameters given, and a content-coding (which is often the identity content-coding), identified by a numeric identifier defined by the CoAP Content-Format registry.

In this specification, the term "byte" is used in its now customary sense as a synonym for "octet".

All multi-byte integers in this protocol are interpreted in network byte order.

Where arithmetic is used, this specification uses the notation familiar from the programming language C, except that the operator "**" stands for exponentiation.

2. Constrained Application Protocol

The interaction model of CoAP is similar to the client/server model of HTTP. However, machine-to-machine interactions typically result in a CoAP implementation acting in both client and server roles. A CoAP request is equivalent to that of HTTP, and is sent by a client to request an action (using a method code) on a resource (identified by a URI) on a server. The server then sends a response with a response code; this response may include a resource representation.

Unlike HTTP, CoAP deals with these interchanges asynchronously over a datagram-oriented transport such as UDP. This is done logically using a layer of messages that supports optional reliability (with exponential back-off). CoAP defines four types of messages: Confirmable, Non-Confirmable, Acknowledgement, Reset; method codes and response codes included in some of these messages make them carry requests or responses. The basic exchanges of the four types of messages are somewhat orthogonal to the request/response interactions; requests can be carried in Confirmable and Non-Confirmable messages, and responses can be carried in these as well as piggy-backed in Acknowledgement messages.

One could think of CoAP logically as using a two-layer approach, a CoAP messaging layer used to deal with UDP and the asynchronous nature of the interactions, and the request/response interactions using Method and Response codes (see Figure 1). CoAP is however a single protocol, with messaging and request/response just features of the CoAP header.

```
                      +----------------------+
                      |      Application     |
                      +----------------------+
                      +----------------------+  \
                      |  Requests/Responses  |  |
                      |----------------------|  | CoAP
                      |       Messages       |  |
                      +----------------------+  /
                      +----------------------+
                      |          UDP         |
                      +----------------------+
```

Figure 1: Abstract layering of CoAP

2.1. Messaging Model

The CoAP messaging model is based on the exchange of messages over UDP between endpoints.

CoAP uses a short fixed-length binary header (4 bytes) that may be followed by compact binary options and a payload. This message format is shared by requests and responses. The CoAP message format is specified in Section 3. Each message contains a Message ID used to detect duplicates and for optional reliability.

Reliability is provided by marking a message as Confirmable (CON). A Confirmable message is retransmitted using a default timeout and exponential back-off between retransmissions, until the recipient sends an Acknowledgement message (ACK) with the same Message ID (for example, 0x7d34) from the corresponding endpoint; see Figure 2. When a recipient is not at all able to process a Confirmable message (i.e., not even able to provide a suitable error response), it replies with a Reset message (RST) instead of an Acknowledgement (ACK).

```
                        Client              Server
                           |                  |
                           |   CON [0x7d34]   |
                           +----------------->|
                           |                  |
                           |   ACK [0x7d34]   |
                           |<-----------------+
                           |                  |
```

Figure 2: Reliable message transmission

A message that does not require reliable transmission, for example each single measurement out of a stream of sensor data, can be sent as a Non-confirmable message (NON). These are not acknowledged, but still have a Message ID for duplicate detection; see Figure 3. When a recipient is not able to process a Non-confirmable message, it may reply with a Reset message (RST).

```
                        Client              Server
                           |                  |
                           |   NON [0x01a0]   |
                           +----------------->|
                           |                  |
```

Figure 3: Unreliable message transmission

See Section 4 for details of CoAP messages.

As CoAP is based on UDP, it also supports the use of multicast IP destination addresses, enabling multicast CoAP requests. Section 8 discusses the proper use of CoAP messages with multicast addresses and precautions for avoiding response congestion.

Several security modes are defined for CoAP in Section 9 ranging from no security to certificate-based security. The use of IPsec along with a binding to DTLS are specified for securing the protocol.

2.2. Request/Response Model

CoAP request and response semantics are carried in CoAP messages, which include either a Method code or Response code, respectively. Optional (or default) request and response information, such as the URI and payload media type are carried as CoAP options. A Token Option is used to match responses to requests independently from the underlying messages (Section 5.3).

A request is carried in a Confirmable (CON) or Non-confirmable (NON) message, and if immediately available, the response to a request carried in a Confirmable message is carried in the resulting Acknowledgement (ACK) message. This is called a piggy-backed response, detailed in Section 5.2.1. Two examples for a basic GET request with piggy-backed response are shown in Figure 4, one successful, one resulting in a 4.04 (Not Found) response.

```
     Client              Server       Client              Server
        |                  |             |                  |
        |   CON [0xbc90]   |             |   CON [0xbc91]   |
        | GET /temperature |             | GET /temperature |
        |   (Token 0x71)   |             |   (Token 0x72)   |
        +----------------->|             +----------------->|
        |                  |             |                  |
        |   ACK [0xbc90]   |             |   ACK [0xbc91]   |
        |   2.05 Content   |             |  4.04 Not Found  |
        |   (Token 0x71)   |             |   (Token 0x72)   |
        |     "22.5 C"     |             |   "Not found"    |
        |<-----------------+             |<-----------------+
        |                  |             |                  |
```

Figure 4: Two GET requests with piggy-backed responses

If the server is not able to respond immediately to a request carried in a Confirmable message, it simply responds with an empty Acknowledgement message so that the client can stop retransmitting the request. When the response is ready, the server sends it in a new Confirmable message (which then in turn needs to be acknowledged by the client). This is called a separate response, as illustrated in Figure 5 and described in more detail in Section 5.2.2.

```
                        Client              Server
                           |                  |
                           |   CON [0x7a10]   |
                           | GET /temperature |
                           |   (Token 0x73)   |
                           +----------------->|
                           |                  |
                           |   ACK [0x7a10]   |
                           |<-----------------+
                           |                  |
                           ... Time Passes  ...
                           |                  |
                           |   CON [0x23bb]   |
                           |   2.05 Content   |
                           |   (Token 0x73)   |
                           |     "22.5 C"     |
                           |<-----------------+
                           |                  |
                           |   ACK [0x23bb]   |
                           +----------------->|
                           |                  |
```

Figure 5: A GET request with a separate response

Likewise, if a request is sent in a Non-Confirmable message, then the response is usually sent using a new Non-Confirmable message, although the server may send a Confirmable message. This type of exchange is illustrated in Figure 6.

```
                        Client              Server
                           |                  |
                           |   NON [0x7a11]   |
                           | GET /temperature |
                           |   (Token 0x74)   |
                           +----------------->|
                           |                  |
                           |   NON [0x23bc]   |
                           |   2.05 Content   |
                           |   (Token 0x74)   |
                           |     "22.5 C"     |
                           |<-----------------+
                           |                  |
```

Figure 6: A NON request and response

CoAP makes use of GET, PUT, POST and DELETE methods in a similar manner to HTTP, with the semantics specified in Section 5.8. (Note that the detailed semantics of CoAP methods are "almost, but not entirely unlike" those of HTTP methods: Intuition taken from HTTP experience generally does apply well, but there are enough differences that make it worthwhile to actually read the present specification.)

URI support in a server is simplified as the client already parses the URI and splits it into host, port, path and query components, making use of default values for efficiency. Response codes correspond to a small subset of HTTP response codes with a few CoAP specific codes added, as defined in Section 5.9.

2.3. Intermediaries and Caching

The protocol supports the caching of responses in order to efficiently fulfill requests. Simple caching is enabled using freshness and validity information carried with CoAP responses. A cache could be located in an endpoint or an intermediary. Caching functionality is specified in Section 5.6.

Proxying is useful in constrained networks for several reasons, including network traffic limiting, to improve performance, to access resources of sleeping devices or for security reasons. The proxying of requests on behalf of another CoAP endpoint is supported in the protocol. When using a proxy, the URI of the resource to request is included in the request, while the destination IP address is set to the address of the proxy. See Section 5.7 for more information on proxy functionality.

As CoAP was designed according to the REST architecture and thus exhibits functionality similar to that of the HTTP protocol, it is quite straightforward to map from CoAP to HTTP and from HTTP to CoAP. Such a mapping may be used to realize an HTTP REST interface using CoAP, or for converting between HTTP and CoAP. This conversion can be carried out by a cross-protocol proxy ("cross-proxy"), which converts the method or response code, media type, and options to the corresponding HTTP feature. Section 10 provides more detail about HTTP mapping.

2.4. Resource Discovery

Resource discovery is important for machine-to-machine interactions, and is supported using the CoRE Link Format [RFC6690] as discussed in Section 7.

3. Message Format

CoAP is based on the exchange of short messages which, by default, are transported over UDP (i.e. each CoAP message occupies the data section of one UDP datagram). CoAP may also be used over Datagram Transport Layer Security (DTLS) (see Section 9.1). It could also be used over other transports such as SMS, TCP or SCTP, the specification of which is out of this document's scope.

CoAP messages are encoded in a simple binary format. A message consists of a fixed-sized CoAP Header followed by options in Type-Length-Value (TLV) format and a payload. The number of options is determined by the header. The payload is made up of the bytes after the options, if any; its length is calculated from the datagram length.

```
  0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Ver| T |  OC   |      Code     |          Message ID           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Options (if any) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Payload (if any) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 7: Message Format

3.1. Header Format

The fields in the header are defined as follows:

Version (Ver): 2-bit unsigned integer. Indicates the CoAP version number. Implementations of this specification MUST set this field to 1. Other values are reserved for future versions.

Type (T): 2-bit unsigned integer. Indicates if this message is of type Confirmable (0), Non-Confirmable (1), Acknowledgement (2) or Reset (3). See Section 4 for the semantics of these message types.

Option Count (OC): 4-bit unsigned integer. Indicates the number of options after the header (0-14). If set to 0, there are no options and the payload (if any) immediately follows the header. If set to 15, then an end-of-options marker is used to indicate the end of options and the start of the payload. The format of options is defined below.

Code: 8-bit unsigned integer. Indicates if the message carries a request (1-31) or a response (64-191), or is empty (0). (All other code values are reserved.) In case of a request, the Code field indicates the Request Method; in case of a response a Response Code. Possible values are maintained in the CoAP Code Registry (Section 12.1). See Section 5 for the semantics of requests and responses.

Message ID: 16-bit unsigned integer in network byte order. Used for the detection of message duplication, and to match messages of type Acknowledgement/Reset and messages of type Confirmable/Non-confirmable. See Section 4 for Message ID generation rules and how messages are matched.

3.2. Option Format

Options MUST appear in order of their Option Number (see Section 5.4.6). A delta encoding is used between options: The Option Number for each Option is calculated as the sum of its Option Delta field and the Option Number of the preceding Option in the message, if any. For the first Option in the message, the Option Delta becomes the Option Number (i.e., an implementation can simply initialize the number variable as zero). Multiple options with the same Option Number can be included by using an Option Delta of zero. The Option Jump mechanism (Section 3.3) is used when the delta to the next option number is greater than 14.

Following the Option Delta, each option has a Length field which specifies the length of the Option Value, in bytes. The Length field can be extended for options with values longer than 14 bytes by adding extension bytes. The maximum length for an option is 1034 bytes. The Option Value immediately follows the Length field.

```
for 0..14:
  0   1   2   3   4   5   6   7
+---+---+---+---+---+---+---+---+
| Option Delta  |    Length     |
+---+---+---+---+---+---+---+---+
|   Option Value ...
+---+---+---+---+---+---+---+---+

for 15..269:
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| Option Delta  | 1   1   1   1 |          Length - 15          |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|   Option Value ...
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

for 270..524:
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| Option Delta  | 1   1   1   1 | 1   1   1   1   1   1   1   1 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|         Length - 270          |   Option Value ...
+---+---+---+---+---+---+---+---+
|
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

for 525..779:
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| Option Delta  | 1   1   1   1 | 1   1   1   1   1   1   1   1 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 1   1   1   1   1   1   1   1 |        Length - 525           |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|   Option Value ...
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

for 780..1034:
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| Option Delta  | 1   1   1   1 | 1   1   1   1   1   1   1   1 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| 1   1   1   1   1   1   1   1 | 1   1   1   1   1   1   1   1 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|         Length - 780          |   Option Value ...
+---+---+---+---+---+---+---+---+
|
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

Figure 8: Option Format

The fields in an option are defined as follows:

Option Delta: 4-bit unsigned integer. Indicates the difference between the Option Number of this option and the previous option (or zero for the first option). In other words, the Option Number is calculated by simply summing the Option Delta fields of this and previous options before it. The Option Delta 15 is reserved for special constructs such as the end-of-options marker (see below) and Option Jumps. The Option Jump mechanism (Section 3.3) is used when the delta to the next option number is larger than 14.

Length: Indicates the length of the Option Value, in bytes. Normally Length is a 4-bit unsigned integer allowing value lengths of 0-14 bytes. When the Length field is set to 15, another byte is added as an 8-bit unsigned integer whose value is added to the 15, allowing option value lengths of 15-270 bytes. For option lengths beyond 270 bytes, we reserve the value 255 of an extension byte to mean "add 255, read another extension byte". Options that are longer than 1034 bytes MUST NOT be sent; an option that has 255 (all one bits) in the field called "Length - 780" MUST be rejected upon reception as an encoding error.

Value: The length and format of the Option Value depends on the respective option, which MAY define variable length values. See Section 3.4 for the formats the options defined in this document make use of; other options MAY make use of other option value formats.

If the Option Count field in the CoAP header is 15 and the Option Header byte is 0xf0 (the Option Delta is 15 and the Option Length is 0), the option is interpreted as the end-of-options marker instead of the option with the resulting Option Number. (In other words, the end-of-options marker always is just a single byte valued 0xf0.) When this marker is encountered, it is immediately followed by the payload (if any). (Note that, by this special meaning, the Option Delta of 15 is made special, not any specific Option Number.) The sender MUST NOT include the end-of-options marker in an Option in a message with an Option Count other than 15; recipients MUST treat this as an encoding error.

Option Numbers are maintained in the CoAP Option Number Registry (Section 12.2). See Section 5.10 for the semantics of the options defined in this document.

3.3. Option Jump

The following construct can occur in front of any Option:

```
  0   1   2   3   4   5   6   7
+---+---+---+---+---+---+---+---+
| 1   1   1   1 | 0   0   0   1 |  0xf1 (Delta = 15)
+---+---+---+---+---+---+---+---+

  0   1   2   3   4   5   6   7
+---+---+---+---+---+---+---+---+
| 1   1   1   1 | 0   0   1   0 |  0xf2
+---+---+---+---+---+---+---+---+
|       Option Jump Value       |  (Delta/8)-2
+---+---+---+---+---+---+---+---+

  0   1   2   3   4   5   6   7
+---+---+---+---+---+---+---+---+
| 1   1   1   1 | 0   0   1   1 |  0xf3
+---+---+---+---+---+---+---+---+
|                               |
+---    Option Jump Value    ---+  (Delta/8)-258
|                               |
+---+---+---+---+---+---+---+---+
```

Figure 9: Option Jump Format

This construct is not by itself an Option. It can occur in front of any Option to increase the current Option number that then goes into its Option number calculation. The increase is done by 15 or in multiples of eight. For the formats that include an Option Jump Value, the actual addition to the current Option number is computed as follows:

Delta = ((Option Jump Value) + N) * 8

where N is 2 for the one-byte version and N is 258 for the two-byte version.

An Option Jump MUST be followed by an actual Option, i.e., it MUST NOT be followed by another Option Jump or an end-of-options indicator. A message violating this MUST be treated as an encoding error.

Option Jumps do NOT count as Options in the Option Count field of the header (i.e., they cannot by themselves end the Option sequence).

3.4. Option Value Formats

The options defined in this document make use of the following option value formats.

3.4.1. uint

A non-negative integer which is represented in network byte order using the given number of bytes. An option definition may specify a range of permissible numbers of bytes; if it has a choice, a sender SHOULD represent the integer with as few bytes as possible, i.e., without leading zeros. A recipient MUST be prepared to process values with leading zeros.

Implementation Note: The exceptional behavior permitted above is for highly constrained templated implementations (e.g. hardware implementations) that use fixed size options in the templates.

```
Length = 0     (implies value of 0)

                0
                0 1 2 3 4 5 6 7
               +-+-+-+-+-+-+-+-+
Length = 1     |     0-255     |
               +-+-+-+-+-+-+-+-+

                0                   1
                0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
               +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
Length = 2     |            0-65535            |
               +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Length = 3 is 24 bits, Length = 4 is 32 bits etc.

3.4.2. string

A Unicode string which is encoded using UTF-8 [RFC3629] in Net-Unicode form [RFC5198]. Note that here and in all other places where UTF-8 encoding is used in the CoAP protocol, the intention is that the encoded strings can be directly used and compared as opaque byte strings by CoAP protocol implementations. There is no expectation and no need to perform normalization within a CoAP implementation unless Unicode strings that are not known to be normalized are imported from sources outside the CoAP protocol. Note also that ASCII strings (that do not make use of special control characters) are always valid UTF-8 Net-Unicode strings.

3.4.3. opaque

An opaque sequence of bytes.

3.4.4. empty

A zero-length sequence of bytes.

4. Message Transmission

CoAP messages are exchanged asynchronously between CoAP endpoints. They are used to transport CoAP requests and responses, the semantics of which are defined in Section 5.

As CoAP is bound to non-reliable transports such as UDP, CoAP messages may arrive out of order, appear duplicated, or go missing without notice. For this reason, CoAP implements a lightweight reliability mechanism, without trying to re-create the full feature set of a transport like TCP. It has the following features:

- o Simple stop-and-wait retransmission reliability with exponential back-off for Confirmable messages.

- o Duplicate detection for both Confirmable and Non-confirmable messages.

4.1. Messages and Endpoints

A CoAP endpoint is the source or destination of a CoAP message. It is identified depending on the security mode used (see Section 9): With no security, the endpoint is solely identified by an IP address and a UDP port number. With other security modes, the endpoint is identified as defined by the security mode.

There are different types of messages. The type of a message is specified by the T field of the CoAP header.

Separate from the message type, a message may carry a request, a response, or be empty. This is signaled by the Code field in the CoAP header and is relevant to the request/response model. Possible values for the Code field are maintained by the CoAP Code Registry (Section 12.1).

An empty message has the Code field set to 0. The OC field SHOULD be set to 0 and no bytes SHOULD be present after the Message ID field. The OC field and any bytes trailing the header MUST be ignored by any recipient.

4.2. Messages Transmitted Reliably

The reliable transmission of a message is initiated by marking the message as Confirmable in the CoAP header. A Confirmable message always carries either a request or response and MUST NOT be empty. A recipient MUST acknowledge such a message with an Acknowledgement message or, if it lacks context to process the message properly (including the case where the message is empty or has an encoding error), MUST reject it; rejecting a Confirmable message is effected by sending a matching Reset message and otherwise ignoring it. The Acknowledgement message MUST echo the Message ID of the Confirmable message, and MUST carry a response or be empty (see Section 5.2.1 and Section 5.2.2). The Reset message MUST echo the Message ID of the confirmable message, and MUST be empty. Rejecting an Acknowledgement or Reset message is effected by silently ignoring it.

The sender retransmits the Confirmable message at exponentially increasing intervals, until it receives an acknowledgement (or Reset message), or runs out of attempts.

Retransmission is controlled by two things that a CoAP endpoint MUST keep track of for each Confirmable message it sends while waiting for an acknowledgement (or reset): a timeout and a retransmission counter. For a new Confirmable message, the initial timeout is set to a random number between ACK_TIMEOUT and (ACK_TIMEOUT * ACK_RANDOM_FACTOR) (see Section 4.8), and the retransmission counter is set to 0. When the timeout is triggered and the retransmission counter is less than MAX_RETRANSMIT, the message is retransmitted, the retransmission counter is incremented, and the timeout is doubled. If the retransmission counter reaches MAX_RETRANSMIT on a timeout, or if the endpoint receives a Reset message, then the attempt to transmit the message is canceled and the application process informed of failure. On the other hand, if the endpoint receives an acknowledgement message in time, transmission is considered successful.

A CoAP endpoint that sent a Confirmable message MAY give up in attempting to obtain an ACK even before the MAX_RETRANSMIT counter value is reached: E.g., the application has canceled the request as it no longer needs a response, or there is some other indication that the CON message did arrive. In particular, a CoAP request message may have elicited a separate response, in which case it is clear to the requester that only the ACK was lost and a retransmission of the request would serve no purpose. However, a responder MUST NOT in turn rely on this cross-layer behavior from a requester, i.e. it SHOULD retain the state to create the ACK for the request, if needed, even if a Confirmable response was already acknowledged by the requester.

4.3. Messages Transmitted Without Reliability

Some messages do not require an acknowledgement. This is particularly true for messages that are repeated regularly for application requirements, such as repeated readings from a sensor where eventual success is sufficient.

As a more lightweight alternative, a message can be transmitted less reliably by marking the message as Non-confirmable. A Non-confirmable message always carries either a request or response and MUST NOT be empty. A Non-confirmable message MUST NOT be acknowledged by the recipient. If a recipient lacks context to process the message properly (including the case where the message is empty or has an encoding error), it MUST reject the message; rejecting a Non-Confirmable message MAY involve sending a matching Reset message, and apart from the Reset message the rejected message MUST be silently ignored.

At the CoAP level, there is no way for the sender to detect if a Non-confirmable message was received or not. A sender MAY choose to transmit multiple copies of a Non-confirmable message within MAX_TRANSMIT_SPAN, or the network may duplicate the message in transit. To enable the receiver to act only once on the message, Non-confirmable messages specify a Message ID as well. (This Message ID is drawn from the same number space as the Message IDs for Confirmable messages.)

4.4. Message Correlation

An Acknowledgement or Reset message is related to a Confirmable message or Non-confirmable message by means of a Message ID along with additional address information of the corresponding endpoint. The Message ID is a 16-bit unsigned integer that is generated by the sender of a Confirmable or Non-confirmable message and included in the CoAP header. The Message ID MUST be echoed in the Acknowledgement or Reset message by the recipient.

The same Message ID MUST NOT be re-used (in communicating with the same endpoint) within the EXCHANGE_LIFETIME (Section 4.8.2).

Implementation Note: Several implementation strategies can be employed for generating Message IDs. In the simplest case a CoAP endpoint generates Message IDs by keeping a single Message ID variable, which is changed each time a new confirmable or non-confirmable message is sent regardless of the destination address or port. Endpoints dealing with large numbers of transactions could keep multiple Message ID variables, for example per prefix or destination address. The initial variable value should be randomized.

For an Acknowledgement or Reset message to match a Confirmable or Non-confirmable message, the Message ID and source endpoint of the Acknowledgement or Reset message MUST match the Message ID and destination endpoint of the Confirmable or Non-confirmable message.

4.5. Message Deduplication

A recipient MUST be prepared to receive the same Confirmable message (as indicated by the Message ID and source endpoint) multiple times within the EXCHANGE_LIFETIME (Section 4.8.2), for example, when its Acknowledgement went missing or didn’t reach the original sender before the first timeout. The recipient SHOULD acknowledge each duplicate copy of a Confirmable message using the same Acknowledgement or Reset message, but SHOULD process any request or response in the message only once. This rule MAY be relaxed in case the Confirmable message transports a request that is idempotent (see Section 5.1) or can be handled in an idempotent fashion. Examples for relaxed message deduplication:

- A server MAY relax the requirement to answer all retransmissions of an idempotent request with the same response (Section 4.2), so that it does not have to maintain state for Message IDs. For example, an implementation might want to process duplicate transmissions of a GET, PUT or DELETE request as separate requests if the effort incurred by duplicate processing is less expensive than keeping track of previous responses would be.

- A constrained server MAY even want to relax this requirement for certain non-idempotent requests if the application semantics make this trade-off favorable. For example, if the result of a POST request is just the creation of some short-lived state at the server, it may be less expensive to incur this effort multiple times for a request than keeping track of whether a previous transmission of the same request already was processed.

A recipient MUST be prepared to receive the same Non-confirmable message (as indicated by the Message ID and source endpoint) multiple times within NON_LIFETIME (Section 4.8.2). As a general rule that may be relaxed based on the specific semantics of a message, the recipient SHOULD silently ignore any duplicated Non-confirmable message, and SHOULD process any request or response in the message only once.

4.6. Message Size

While specific link layers make it beneficial to keep CoAP messages small enough to fit into their link layer packets (see Section 1), this is a matter of implementation quality. The CoAP specification itself provides only an upper bound to the message size. Messages larger than an IP fragment result in undesired packet fragmentation. A CoAP message, appropriately encapsulated, SHOULD fit within a single IP packet (i.e., avoid IP fragmentation) and (by fitting into one UDP payload) obviously MUST fit within a single IP datagram. If the Path MTU is not known for a destination, an IP MTU of 1280 bytes SHOULD be assumed; if nothing is known about the size of the headers, good upper bounds are 1152 bytes for the message size and 1024 bytes for the payload size.

Implementation Note: CoAP’s choice of message size parameters works well with IPv6 and with most of today’s IPv4 paths. (However, with IPv4, it is harder to absolutely ensure that there is no IP fragmentation. If IPv4 support on unusual networks is a consideration, implementations may want to limit themselves to more conservative IPv4 datagram sizes such as 576 bytes; worse, the absolute minimum value of the IP MTU for IPv4 is as low as 68 bytes, which would leave only 40 bytes minus security overhead for a UDP payload. Implementations extremely focused on this problem set might also set the IPv4 DF bit and perform some form of path MTU discovery; this should generally be unnecessary in most realistic use cases for CoAP, however.) A more important kind of fragmentation in many constrained networks is that on the adaptation layer (e.g., 6LoWPAN L2 packets are limited to 127 bytes including various overheads); this may motivate implementations to be frugal in their packet sizes and to move to block-wise transfers [I-D.ietf-core-block] when approaching three-digit message sizes.

Message sizes are also of considerable importance to implementations on constrained nodes. Many implementations will need to allocate a buffer for incoming messages. If an implementation is too constrained to allow for allocating the above-mentioned upper bound, it could apply the following implementation strategy: Implementations receiving a datagram into a buffer that is too small are usually able to determine if the trailing portion of a datagram was discarded and to retrieve the initial portion. So, if not all of the payload, at least the CoAP header and options are likely to fit within the buffer. A server can thus fully interpret a request and return a 4.13 (Request Entity Too Large) response code if the payload was truncated. A client sending an idempotent request and receiving a response larger than would fit in the buffer can repeat the request with a suitable value for the Block Option [I-D.ietf-core-block].

4.7. Congestion Control

Basic congestion control for CoAP is provided by the exponential back-off mechanism in Section 4.2.

In order not to cause congestion, Clients (including proxies) MUST strictly limit the number of simultaneous outstanding interactions that they maintain to a given server (including proxies) to NSTART. An outstanding interaction is either a CON for which an ACK has not yet been received but is still expected (message layer) or a request for which neither a response nor an Acknowledgment message has yet been received but is still expected (which may both occur at the same time, counting as one outstanding interaction). The default value of NSTART for this specification is 1.

Further congestion control optimizations and considerations are expected in the future, which may for example provide automatic initialization of the CoAP transmission parameters defined in Section 4.8, and thus may allow a value for NSTART greater than one.

A client stops expecting a response to a Confirmable request for which no acknowledgment message was received, after EXCHANGE_LIFETIME. The specific algorithm by which a client stops to "expect" a response to a Confirmable request that was acknowledged, or to a Non-confirmable request, is not defined. Unless this is modified by additional congestion control optimizations, it MUST be chosen in such a way that an endpoint does not exceed an average data rate of PROBING_RATE in sending to another endpoint that does not respond.

Note: CoAP places the onus of congestion control mostly on the clients. However, clients may malfunction or actually be attackers, e.g. to perform amplification attacks (Section 11.3). To limit the damage (to the network and to its own energy resources), a server SHOULD implement some rate limiting for its response transmission based on reasonable assumptions about application requirements. This is most helpful if the rate limit can be made effective for the misbehaving endpoints, only.

4.8. Transmission Parameters

Message transmission is controlled by the following parameters:

name	default value
ACK_TIMEOUT	2 seconds
ACK_RANDOM_FACTOR	1.5
MAX_RETRANSMIT	4
NSTART	1
DEFAULT_LEISURE	5 seconds
PROBING_RATE	1 Byte/second

4.8.1. Changing The Parameters

The values for ACK_TIMEOUT, ACK_RANDOM_FACTOR, MAX_RETRANSMIT, NSTART, DEFAULT_LEISURE, and PROBING_RATE may be configured to values specific to the application environment (including dynamically adjusted values), however the configuration method is out of scope of this document. It is recommended that an application environment use consistent values for these parameters.

The transmission parameters have been chosen to achieve a behavior in the presence of congestion that is safe in the Internet. If a configuration desires to use different values, the onus is on the configuration to ensure these congestion control properties are not violated. In particular, a decrease of ACK_TIMEOUT below 1 second would violate the guidelines of [RFC5405]. ([I-D.allman-tcpm-rto-consider] provides some additional background.) CoAP was designed to enable implementations that do not maintain round-trip-time (RTT) measurements. However, where it is desired to decrease the ACK_TIMEOUT significantly or increase NSTART, this can only be done safely when maintaining such measurements. Configurations MUST NOT decrease ACK_TIMEOUT or increase NSTART without using mechanisms that ensure congestion control safety, either defined in the configuration or in future standards documents.

ACK_RANDOM_FACTOR MUST NOT be decreased below 1.0, and it SHOULD have a value that is sufficiently different from 1.0 to provide some protection from synchronization effects.

MAX_RETRANSMIT can be freely adjusted, but a too small value will reduce the probability that a confirmable message is actually received, while a larger value than given here will require further adjustments in the time values (see discussion below).

If the choice of transmission parameters leads to an increase of derived time values (see below), the configuration mechanism MUST ensure the adjusted value is also available to all the endpoints in communicating with which these adjusted values are to be used.

4.8.2. Time Values derived from Transmission Parameters

The combination of ACK_TIMEOUT, ACK_RANDOM_FACTOR and MAX_RETRANSMIT influences the timing of retransmissions, which in turn influences how long certain information items need to be kept by an implementation. To be able to unambiguously reference these derived time values, we give them names as follows:

- MAX_TRANSMIT_SPAN is the maximum time from the first transmission of a confirmable message to its last retransmission. For the default transmission parameters, the value is (2+4+8+16)*1.5 = 45 seconds, or more generally:

  ACK_TIMEOUT * (2 ** MAX_RETRANSMIT - 1) * ACK_RANDOM_FACTOR

- MAX_TRANSMIT_WAIT is the maximum time from the first transmission of a confirmable message to the time when the sender gives up on receiving an acknowledgement or reset. For the default transmission parameters, the value is (2+4+8+16+32)*1.5 = 93 seconds, or more generally:

  ACK_TIMEOUT * (2 ** (MAX_RETRANSMIT + 1) - 1) * ACK_RANDOM_FACTOR

In addition, some assumptions need to be made on the characteristics of the network and the nodes.

- MAX_LATENCY is the maximum time a datagram is expected to take from the start of its transmission to the completion of its reception. This constant is related to the MSL (Maximum Segment Lifetime) of [RFC0793], which is "arbitrarily defined to be 2 minutes" ([RFC0793] glossary, page 81). Note that this is not necessarily smaller than MAX_TRANSMIT_WAIT, as MAX_LATENCY is not intended to describe a situation when the protocol works well, but the worst case situation against which the protocol has to guard. We, also arbitrarily, define MAX_LATENCY to be 100 seconds. Apart from being reasonably realistic for the bulk of configurations as well as close to the historic choice for TCP, this value also allows message ID lifetime timers to be represented in 8 bits (when measured in seconds). In these calculations, there is no assumption that the direction of the transmission is irrelevant (i.e. that the network is symmetric), just that the same value can reasonably be used as a maximum value for both directions. If that is not the case, the following calculations become only slightly more complex.

- PROCESSING_DELAY is the time a node takes to turn around a confirmable message into an acknowledgement. We assume the node will attempt to send an ACK before having the sender time out, so as a conservative assumption we set it equal to ACK_TIMEOUT.

- MAX_RTT is the maximum round-trip time, or:

  2 * MAX_LATENCY + PROCESSING_DELAY

From these values, we can derive the following values relevant to the protocol operation:

- EXCHANGE_LIFETIME is the time from starting to send a confirmable message to the time when an acknowledgement is no longer expected, i.e. message layer information about the message exchange can be purged. EXCHANGE_LIFETIME includes a MAX_TRANSMIT_SPAN, a MAX_LATENCY forward, PROCESSING_DELAY, and a MAX_LATENCY for the way back. Note that there is no need to consider MAX_TRANSMIT_WAIT if the configuration is chosen such that the last waiting period (ACK_TIMEOUT * (2 ** MAX_RETRANSMIT) or the difference between MAX_TRANSMIT_SPAN and MAX_TRANSMIT_WAIT) is less than MAX_LATENCY -- which is a likely choice, as MAX_LATENCY is a worst case value unlikely to be met in the real world. In this case, EXCHANGE_LIFETIME simplifies to:

  (ACK_TIMEOUT * (2 ** MAX_RETRANSMIT - 1) * ACK_RANDOM_FACTOR) +
  (2 * MAX_LATENCY) + PROCESSING_DELAY

  or 248 seconds with the default transmission parameters.

- NON_LIFETIME is the time from sending a non-confirmable message to the time its message-ID can be safely reused. If multiple transmission of a NON message is not used, its value is MAX_LATENCY, or 100 seconds. However, a CoAP sender might send a NON message multiple times, in particular for multicast applications. While the period of re-use is not bounded by the specification, an expectation of reliable detection of duplication at the receiver is in the timescales of MAX_TRANSMIT_SPAN. Therefore, for this purpose, it is safer to use the value:

  MAX_TRANSMIT_SPAN + MAX_LATENCY

  or 145 seconds with the default transmission parameters; however, an implementation that just wants to use a single timeout value for retiring message-IDs can safely use the larger value for EXCHANGE_LIFETIME.

5. Request/Response Semantics

CoAP operates under a similar request/response model as HTTP: a CoAP endpoint in the role of a "client" sends one or more CoAP requests to a "server", which services the requests by sending CoAP responses. Unlike HTTP, requests and responses are not sent over a previously established connection, but exchanged asynchronously over CoAP messages.

5.1. Requests

A CoAP request consists of the method to be applied to the resource, the identifier of the resource, a payload and Internet media type (if any), and optional meta-data about the request.

CoAP supports the basic methods of GET, POST, PUT, DELETE, which are easily mapped to HTTP. They have the same properties of safe (only retrieval) and idempotent (you can invoke it multiple times with the same effects) as HTTP (see Section 9.1 of [RFC2616]). The GET method is safe, therefore it MUST NOT take any other action on a resource other than retrieval. The GET, PUT and DELETE methods MUST be performed in such a way that they are idempotent. POST is not idempotent, because its effect is determined by the origin server and dependent on the target resource; it usually results in a new resource being created or the target resource being updated.

A request is initiated by setting the Code field in the CoAP header of a Confirmable or a Non-confirmable message to a Method Code and including request information.

The methods used in requests are described in detail in Section 5.8.

5.2. Responses

After receiving and interpreting a request, a server responds with a CoAP response, which is matched to the request by means of a client-generated token.

A response is identified by the Code field in the CoAP header being set to a Response Code. Similar to the HTTP Status Code, the CoAP Response Code indicates the result of the attempt to understand and satisfy the request. These codes are fully defined in Section 5.9. The Response Code numbers to be set in the Code field of the CoAP header are maintained in the CoAP Response Code Registry (Section 12.1.2).

```
 0
 0 1 2 3 4 5 6 7
+-+-+-+-+-+-+-+-+
|class|  detail |
+-+-+-+-+-+-+-+-+
```

Figure 10: Structure of a Response Code

The upper three bits of the 8-bit Response Code number define the class of response. The lower five bits do not have any categorization role; they give additional detail to the overall class (Figure 10). There are 3 classes:

2 - Success: The request was successfully received, understood, and accepted.

4 - Client Error: The request contains bad syntax or cannot be fulfilled.

5 - Server Error: The server failed to fulfill an apparently valid request.

The response codes are designed to be extensible: Response Codes in the Client Error and Server Error class that are unrecognized by an endpoint MUST be treated as being equivalent to the generic Response Code of that class (4.00 and 5.00, respectively). However, there is no generic Response Code indicating success, so a Response Code in the Success class that is unrecognized by an endpoint can only be used to determine that the request was successful without any further details.

As a human readable notation for specifications and protocol diagnostics, the numeric value of a response code is indicated by giving the upper three bits in decimal, followed by a dot and then the lower five bits in a two-digit decimal. E.g., "Not Found" is written as 4.04 -- indicating a value of hexadecimal 0x84 or decimal 132. In other words, the dot "." functions as a short-cut for "*32+".

The possible response codes are described in detail in Section 5.9.

Responses can be sent in multiple ways, which are defined below.

5.2.1. Piggy-backed

In the most basic case, the response is carried directly in the Acknowledgement message that acknowledges the request (which requires that the request was carried in a Confirmable message). This is called a "Piggy-backed" Response.

The response is returned in the Acknowledgement message independent of whether the response indicates success or failure. In effect, the response is piggy-backed on the Acknowledgement message, so no separate message is required to both acknowledge that the request was received and return the response.

Implementation Note: The protocol leaves the decision whether to piggy-back a response or not (i.e., send a separate response) to the server. The client MUST be prepared to receive either. On the quality of implementation level, there is a strong expectation that servers will implement code to piggy-back whenever possible -- saving resources in the network and both at the client and at the server.

5.2.2. Separate

It may not be possible to return a piggy-backed response in all cases. For example, a server might need longer to obtain the representation of the resource requested than it can wait sending back the Acknowledgement message, without risking the client to repeatedly retransmit the request message. Responses to requests carried in a Non-Confirmable message are always sent separately (as there is no Acknowledgement message).

The server maybe initiates the attempt to obtain the resource representation and times out an acknowledgement timer, or it immediately sends an acknowledgement knowing in advance that there will be no piggy-backed response. The acknowledgement effectively is a promise that the request will be acted upon.

When the server finally has obtained the resource representation, it sends the response. When it is desired that this message is not lost, it is sent as a Confirmable message from the server to the client and answered by the client with an Acknowledgement, echoing the new Message ID chosen by the server. (It may also be sent as a Non-Confirmable message; see Section 5.2.3.)

Implementation Notes: Note that, as the underlying datagram transport may not be sequence-preserving, the Confirmable message carrying the response may actually arrive before or after the acknowledgement message for the request. Note also that, while the CoAP protocol itself does not make any specific demands here, there is an expectation that the response will come within a time frame that is reasonable from an application point of view; as there is no underlying transport protocol that could be instructed to run a keep-alive mechanism, the requester MAY want to set up a timeout that is unrelated to CoAP's retransmission timers in case the server is destroyed or otherwise unable to send the response.)

An exchange is separate by definition when the Acknowledgement to the Confirmable request is an empty message. The Acknowledgement to the Confirmable response MUST also be an empty message, i.e. one that carries neither a request nor a response. However, a server MUST stop retransmitting its response on any matching Acknowledgement (silently ignoring any response code or payload) or Reset message.

5.2.3. Non-Confirmable

If the request message is Non-confirmable, then the response SHOULD be returned in a Non-confirmable message as well. However, an endpoint MUST be prepared to receive a Non-confirmable response (preceded or followed by an empty acknowledgement message) in reply to a Confirmable request, or a Confirmable response in reply to a Non-confirmable request.

5.3. Request/Response Matching

Regardless of how a response is sent, it is matched to the request by means of a token that is included by the client in the request as one of the options along with additional address information of the corresponding endpoint. The token MUST be echoed by the server in any resulting response without modification.

The exact rules for matching a response to a request are as follows:

1. The source endpoint of the response MUST be the same as the destination endpoint of the original request.

2. In a piggy-backed response, both the Message ID of the Confirmable request and the Acknowledgement, and the token of the response and original request MUST match. In a separate response, just the token of the response and original request MUST match.

The client SHOULD generate tokens in a way that tokens currently in use for a given source/destination pair are unique. (Note that a client can use the same token for any request if it uses a different source port number each time.)

An endpoint that did not generate a token MUST treat it as opaque and make no assumptions about its format. (Note that there is a default value for the Token Option, so every message carries a token, even if it is not explicitly expressed in a CoAP option.)

In case a message carrying a response is unexpected (i.e. the client is not waiting for a response at the endpoint addressed and/or with the given token), the response is rejected (Section 4.2, Section 4.3).

Implementation Note: A client that receives a response in a CON message may want to clean up the message state right after sending the ACK. If that ACK is lost and the server retransmits the CON, the client may no longer have any state to correlate this response to, making the retransmission an unexpected message; the client may send a Reset message so it does not receive any more retransmissions. This behavior is normal and not an indication of an error. (Clients that are not aggressively optimized in their state memory usage will still have message state that will identify the second CON as a retransmission. Clients that actually expect more messages from the server [I-D.ietf-core-observe] will have to keep state in any case.)

5.4. Options

Both requests and responses may include a list of one or more options. For example, the URI in a request is transported in several options, and meta-data that would be carried in an HTTP header in HTTP is supplied as options as well.

CoAP defines a single set of options that are used in both requests and responses:

- o Content-Format
- o ETag
- o Location-Path
- o Location-Query
- o Max-Age
- o Proxy-Uri
- o Token
- o Uri-Host
- o Uri-Path
- o Uri-Port

- Uri-Query
- Accept
- If-Match
- If-None-Match

The semantics of these options along with their properties are defined in detail in Section 5.10.

Not all options are defined for use with all methods and response codes. The possible options for methods and response codes are defined in Section 5.8 and Section 5.9 respectively. In case an option is not defined for a method or response code, it MUST NOT be included by a sender and MUST be treated like an unrecognized option by a recipient.

An Option number is constructed with a bit mask to indicate if an option is Critical/Elective, Unsafe/Safe and in the case of Safe, also a Cache-Key as indicated by the following figure. When bit 7 (the least significant bit) is 1, an option is Critical (and likewise Elective when 0). When bit 6 is 1, an option is Unsafe (and likewise Safe when 0). When an option is not Unsafe, it is not a Cache-Key (NoCacheKey) if and only if bits 3-5 are all set to 1; all other bit combinations mean that it indeed is a Cache-Key. These classes of options are explained in the next sections.

```
  0   1   2   3   4   5   6   7
+---+---+---+---+---+---+---+---+
|           | NoCacheKey| U | C |
+---+---+---+---+---+---+---+---+
```

Figure 11: Option Number Mask

An endpoint may use an equivalent of the following C code to derive the characteristics of an option number "onum":

```
Critical = (onum & 1);
UnSafe = (onum & 2);
NoCacheKey = ((onum & 0x1e) == 0x1c);
```

Figure 12

5.4.1. Critical/Elective

Options fall into one of two classes: "critical" or "elective". The difference between these is how an option unrecognized by an endpoint is handled:

- o Upon reception, unrecognized options of class "elective" MUST be silently ignored.

- o Unrecognized options of class "critical" that occur in a confirmable request MUST cause the return of a 4.02 (Bad Option) response. This response SHOULD include a diagnostic message describing the unrecognized option(s) (see Section 5.5.2).

- o Unrecognized options of class "critical" that occur in a confirmable response, or piggy-backed in an acknowledgement, MUST cause the response to be rejected (Section 4.2).

- o Unrecognized options of class "critical" that occur in a non-confirmable message MUST cause the message to be rejected (Section 4.3).

Note that, whether critical or elective, an option is never "mandatory" (it is always optional): These rules are defined in order to enable implementations to stop processing options they do not understand or implement.

Critical/Elective rules apply to non-proxying endpoints. A proxy processes options based on Unsafe/Safe classes as defined in Section 5.7.

5.4.2. Proxy Unsafe/Safe and Cache-Key

In addition to an option being marked as Critical or Elective, options are also classified based on how a proxy is to deal with the option if it does not recognize it. For this purpose, an option can either be considered Unsafe to Forward (UnSafe is set) or Safe to Forward (UnSafe is clear).

In addition, for options that are marked Safe to Forward, the option indicates whether it is intended to be part of the Cache-Key in a request (NoCacheKey is not all set) or not (NoCacheKey is set).

Note: The Cache-Key indication is relevant only for proxies that do not implement the given option as a request option and instead rely on the Safe/Unsafe indication only. E.g., for ETag, actually using the request option as a cache key is grossly inefficient, but it is the best thing one can do if ETag is not implemented by a proxy, as the reponse is going to differ based on the presence of the request option. A more useful proxy that does implement the ETag request option is not using ETag as a cache key.

Proxy behavior with regard to these classes is defined in Section 5.7.

5.4.3. Length

Option values are defined to have a specific length, often in the form of an upper and lower bound. If the length of an option value in a request is outside the defined range, that option MUST be treated like an unrecognized option (see Section 5.4.1).

5.4.4. Default Values

Options may be defined to have a default value. If the value of option is intended to be this default value, the option SHOULD NOT be included in the message. If the option is not present, the default value MUST be assumed.

Where a critical option has a default value, this is chosen in such a way that the absence of the option in a message can be processed properly both by implementations unaware of the critical option and by implementations that interpret this absence as the presence of the default value for the option.

5.4.5. Repeatable Options

The definition of an option MAY specify the option to be repeatable. An option that is repeatable MAY be included one or more times in a message. An option that is not repeatable MUST NOT be included more than once in a message.

If a message includes an option with more occurrences than the option is defined for, the additional option occurrences MUST be treated like an unrecognized option (see Section 5.4.1).

5.4.6. Option Numbers

Options are identified by an option number. Odd numbers indicate a critical option, while even numbers indicate an elective option. (Note that this is not just a convention, it is a feature of the protocol: Whether an option is elective or critical is entirely determined by whether its option number is even or odd.)

The option numbers for the options defined in this document are listed in the CoAP Option Number Registry (Section 12.2).

5.5. Payload

Both requests and responses may include payload, depending on the method or response code respectively. If a method or response code is not defined to have a payload, then a sender MUST NOT include one, and a recipient MUST ignore it.

5.5.1. Representation

The payload of requests or of responses indicating success is typically a representation of a resource or the result of the requested action. Its format is specified by the Internet media type and content coding given by the Content-Format Option. In the absence of this option, no default value is assumed and the format must be inferred by the application (e.g., from the application context or by "sniffing" the payload).

5.5.2. Diagnostic Message

The payload of responses indicating a client or server error is a brief human-readable diagnostic message, explaining the error situation. This diagnostic message MUST be encoded using UTF-8 [RFC3629], more specifically using Net-Unicode form [RFC5198]. The Content-Format Option MUST NOT be included by the sender and MUST be treated like an unrecognized option by the recipient.

The message is similar to the Reason-Phrase on an HTTP status line. It is not intended for end-users but for software engineers that during debugging need to interpret it in the context of the present, English-language specification; therefore no mechanism for language tagging is needed or provided. In contrast to what is usual in HTTP, the message SHOULD be empty if there is no additional information beyond the response code.

5.6. Caching

CoAP endpoints MAY cache responses in order to reduce the response time and network bandwidth consumption on future, equivalent requests.

The goal of caching in CoAP is to reuse a prior response message to satisfy a current request. In some cases, a stored response can be reused without the need for a network request, reducing latency and network round-trips; a "freshness" mechanism is used for this purpose (see Section 5.6.1). Even when a new request is required, it is often possible to reuse the payload of a prior response to satisfy the request, thereby reducing network bandwidth usage; a "validation" mechanism is used for this purpose (see Section 5.6.2).

Unlike HTTP, the cacheability of CoAP responses does not depend on the request method, but the Response Code. The cacheability of each Response Code is defined along the Response Code definitions in Section 5.9. Response Codes that indicate success and are unrecognized by an endpoint MUST NOT be cached.

For a presented request, a CoAP endpoint MUST NOT use a stored response, unless:

- the presented request method and that used to obtain the stored response match,

- all options match between those in the presented request and those of the request used to obtain the stored response (which includes the request URI), except that there is no need for a match of the Token, Max-Age, or ETag request option(s), or any request options marked as NoCacheKey (Section 5.4), and

- the stored response is either fresh or successfully validated as defined below.

5.6.1. Freshness Model

When a response is "fresh" in the cache, it can be used to satisfy subsequent requests without contacting the origin server, thereby improving efficiency.

The mechanism for determining freshness is for an origin server to provide an explicit expiration time in the future, using the Max-Age Option (see Section 5.10.6). The Max-Age Option indicates that the response is to be considered not fresh after its age is greater than the specified number of seconds.

The Max-Age Option defaults to a value of 60. Thus, if it is not present in a cacheable response, then the response is considered not fresh after its age is greater than 60 seconds. If an origin server wishes to prevent caching, it MUST explicitly include a Max-Age Option with a value of zero seconds.

If a client has a fresh stored response and makes a new request matching the request for that stored response, the new response invalidates the old response.

5.6.2. Validation Model

When an endpoint has one or more stored responses for a GET request, but cannot use any of them (e.g., because they are not fresh), it can use the ETag Option (Section 5.10.7) in the GET request to give the origin server an opportunity to both select a stored response to be used, and to update its freshness. This process is known as "validating" or "revalidating" the stored response.

When sending such a request, the endpoint SHOULD add an ETag Option specifying the entity-tag of each stored response that is applicable.

A 2.03 (Valid) response indicates the stored response identified by the entity-tag given in the response's ETag Option can be reused, after updating its freshness with the value of the Max-Age Option that is included (explicitly, or implicitly as a default value) with the response (see Section 5.9.1.3).

Any other response code indicates that none of the stored responses nominated in the request is suitable. Instead, the response SHOULD be used to satisfy the request and MAY replace the stored response.

5.7. Proxying

A proxy is a CoAP endpoint that can be tasked by CoAP clients to perform requests on their behalf. This may be useful, for example, when the request could otherwise not be made, or to service the response from a cache in order to reduce response time and network bandwidth or energy consumption.

In an overall architecture for a Constrained RESTful Environment, proxies can serve quite different purposes. Proxies can be explicitly selected by clients, a role that we term "forward-proxy". Proxies can also be inserted to stand in for origin servers, a role that we term "reverse-proxy". Orthogonal to this distinction, a proxy can map from a CoAP request to a CoAP request (CoAP-to-CoAP proxy) or translate from or to a different protocol ("cross-proxy"). Full definitions of these terms are provided in Section 1.2.

Notes: The terminology in this specification has been selected to be culturally compatible with the terminology used in the wider Web application environments, without necessarily matching it in every detail (which may not even be relevant to Constrained RESTful Environments). Not too much semantics should be ascribed to the components of the terms (such as "forward", "reverse", or "cross").

HTTP proxies, besides acting as HTTP proxies, often offer a transport protocol proxying function ("CONNECT") to enable end-to-end transport layer security through the proxy. No such function is defined for CoAP-to-CoAP proxies in this specification, as forwarding of UDP packets is unlikely to be of much value in Constrained RESTful environments. See also Section 10.2.7 for the cross-proxy case.

5.7.1. Proxy Operation

A proxy generally needs a way to determine potential request parameters for a request to a destination based on the request it received. This way is fully specified for a forward-proxy, but may depend on the specific configuration for a reverse-proxy. In particular, the client of a reverse-proxy generally does not indicate a locator for the destination, necessitating some form of namespace translation in the reverse-proxy. However, some aspects of the operation of proxies are common to all its forms.

If a proxy does not employ a cache, then it simply forwards the translated request to the determined destination. Otherwise, if it does employ a cache but does not have a stored response that matches the translated request and is considered fresh, then it needs to refresh its cache according to Section 5.6. For options in the request that the proxy recognizes, it knows whether the option is intended to act as part of the key used in looking up the cached value or not. E.g., since requests for different Uri-Path values address different resources, Uri-Path values are always parts of the cache key, while, e.g., Token values are never part of the cache key. For options that the proxy does not recognize but that are marked Safe in the option number, the option also indicates whether it is to be included in the cache key (NoCacheKey is not all set) or not (NoCacheKey is all set). (Options that are unrecognized and marked Unsafe lead to 4.02 Bad Option.)

If the request to the destination times out, then a 5.04 (Gateway Timeout) response MUST be returned. If the request to the destination returns a response that cannot be processed by the proxy (e.g, due to unrecognized critical options, encoding errors), then a 5.02 (Bad Gateway) response MUST be returned. Otherwise, the proxy returns the response to the client.

If a response is generated out of a cache, it MUST be generated with a Max-Age Option that does not extend the max-age originally set by the server, considering the time the resource representation spent in the cache. E.g., the Max-Age Option could be adjusted by the proxy for each response using the formula:

    proxy-max-age = original-max-age - cache-age

For example if a request is made to a proxied resource that was refreshed 20 seconds ago and had an original Max-Age of 60 seconds, then that resource’s proxied max-age is now 40 seconds. Considering potential network delays on the way from the origin server, a proxy

SHOULD be conservative in the max-age values offered.

All options present in a proxy request MUST be processed at the proxy. Unsafe options in a request that are not recognized by the proxy MUST lead to a 4.02 (Bad Option) response being returned by the proxy. A CoAP-to-CoAP proxy MUST forward to the origin server all Safe options that it does not recognize. Similarly, Unsafe options in a response that are not recognized by the CoAP-to-CoAP proxy server MUST lead to a 5.02 (Bad Gateway) response. Again, Safe options that are not recognized MUST be forwarded.

Additional considerations for cross-protocol proxying between CoAP and HTTP are discussed in Section 10.

5.7.2. Forward-Proxies

CoAP distinguishes between requests made (as if) to an origin server and a request made through a forward-proxy. CoAP requests to a forward-proxy are made as normal confirmable or non-confirmable requests to the forward-proxy endpoint, but specify the request URI in a different way: The request URI in a proxy request is specified as a string in the Proxy-Uri Option (see Section 5.10.3), while the request URI in a request to an origin server is split into the Uri-Host, Uri-Port, Uri-Path and Uri-Query Options (see Section 5.10.2).

When a proxy request is made to an endpoint and the endpoint is unwilling or unable to act as proxy for the request URI, it MUST return a 5.05 (Proxying Not Supported) response. If the authority (host and port) is recognized as identifying the proxy endpoint itself (see Section 5.10.3), then the request MUST be treated as a local (non-proxied) request.

Unless a proxy is configured to forward the proxy request to another proxy, it MUST translate the request as follows: The scheme of the request URI defines the outgoing protocol and its details (e.g., CoAP is used over UDP for the "coap" scheme and over DTLS for the "coaps" scheme.) For a CoAP-to-CoAP proxy, the origin server's IP address and port are determined by the authority component of the request URI, and the request URI is decoded and split into the Uri-Host, Uri-Port, Uri-Path and Uri-Query Options. This consumes the Proxy-URI option, which is therefore not forwarded to the origin server.

5.7.3. Reverse-Proxies

Reverse-proxies do not make use of the Proxy-Uri option, but need to determine the destination (next hop) of a request from information in the request and information in their configuration. E.g., a reverse-proxy might offer various resources the existence of which it has learned through resource discovery as if they were its own resources. The reverse-proxy is free to build a namespace for the URIs that identify these resources. A reverse-proxy may also build a namespace that gives the client more control over where the request goes, e.g. by embedding host identifiers and port numbers into the URI path of the resources offered.

In processing the response, a reverse-proxy has to be careful about namespacing the ETag option. In many cases, it can be forwarded unchanged. If the mapping from a resource offered by the reverse-proxy to resources offered by its various origin servers is not unique, the reverse-proxy may need to generate a new ETag, making sure the semantics of this option are properly preserved.

5.8. Method Definitions

In this section each method is defined along with its behavior. A request with an unrecognized or unsupported Method Code MUST generate a 4.05 (Method Not Allowed) piggy-backed response.

5.8.1. GET

The GET method retrieves a representation for the information that currently corresponds to the resource identified by the request URI. If the request includes one or more Accept Options, they indicate the preferred content-format of a response. If the request includes an ETag Option, the GET method requests that ETag be validated and that the representation be transferred only if validation failed. Upon success a 2.05 (Content) or 2.03 (Valid) response code SHOULD be present in the response.

The GET method is safe and idempotent.

5.8.2. POST

The POST method requests that the representation enclosed in the request be processed. The actual function performed by the POST method is determined by the origin server and dependent on the target resource. It usually results in a new resource being created or the target resource being updated.

If a resource has been created on the server, the response returned by the server SHOULD have a 2.01 (Created) response code and SHOULD include the URI of the new resource in a sequence of one or more Location-Path and/or Location-Query Options (Section 5.10.8). If the POST succeeds but does not result in a new resource being created on the server, the response SHOULD have a 2.04 (Changed) response code. If the POST succeeds and results in the target resource being deleted, the response SHOULD have a 2.02 (Deleted) response code.

POST is neither safe nor idempotent.

5.8.3. PUT

The PUT method requests that the resource identified by the request URI be updated or created with the enclosed representation. The representation format is specified by the media type and content coding given in the Content-Format Option, if provided.

If a resource exists at the request URI the enclosed representation SHOULD be considered a modified version of that resource, and a 2.04 (Changed) response code SHOULD be returned. If no resource exists then the server MAY create a new resource with that URI, resulting in a 2.01 (Created) response code. If the resource could not be created or modified, then an appropriate error response code SHOULD be sent.

Further restrictions to a PUT can be made by including the If-Match (see Section 5.10.9) or If-None-Match (see Section 5.10.10) options in the request.

PUT is not safe, but is idempotent.

5.8.4. DELETE

The DELETE method requests that the resource identified by the request URI be deleted. A 2.02 (Deleted) response code SHOULD be used on success or in case the resource did not exist before the request.

DELETE is not safe, but is idempotent.

5.9. Response Code Definitions

Each response code is described below, including any options required in the response. Where appropriate, some of the codes will be specified in regards to related response codes in HTTP [RFC2616]; this does not mean that any such relationship modifies the HTTP mapping specified in Section 10.

5.9.1. Success 2.xx

This class of status code indicates that the clients request was successfully received, understood, and accepted.

5.9.1.1. 2.01 Created

Like HTTP 201 "Created", but only used in response to POST and PUT requests. The payload returned with the response, if any, is a representation of the action result.

If the response includes one or more Location-Path and/or Location-Query Options, the values of these options specify the location at which the resource was created. Otherwise, the resource was created at the request URI. A cache receiving this response MUST mark any stored response for the created resource as not fresh.

This response is not cacheable.

5.9.1.2. 2.02 Deleted

Like HTTP 204 "No Content", but only used in response to DELETE requests. The payload returned with the response, if any, is a representation of the action result.

This response is not cacheable. However, a cache SHOULD mark any stored response for the deleted resource as not fresh.

5.9.1.3. 2.03 Valid

Related to HTTP 304 "Not Modified", but only used to indicate that the response identified by the entity-tag identified by the included ETag Option is valid. Accordingly, the response MUST include an ETag Option.

When a cache receives a 2.03 (Valid) response, it MUST update the stored response with the value of the Max-Age Option included in the response (see Section 5.6.2).

5.9.1.4. 2.04 Changed

Like HTTP 204 "No Content", but only used in response to POST and PUT requests. The payload returned with the response, if any, is a representation of the action result.

This response is not cacheable. However, a cache MUST mark any stored response for the changed resource as not fresh.

5.9.1.5. 2.05 Content

Like HTTP 200 "OK", but only used in response to GET requests.

The payload returned with the response is a representation of the target resource.

This response is cacheable: Caches can use the Max-Age Option to determine freshness (see Section 5.6.1) and (if present) the ETag Option for validation (see Section 5.6.2).

5.9.2. Client Error 4.xx

This class of response code is intended for cases in which the client seems to have erred. These response codes are applicable to any request method.

The server SHOULD include a diagnostic message as detailed in Section 5.5.2.

Responses of this class are cacheable: Caches can use the Max-Age Option to determine freshness (see Section 5.6.1). They cannot be validated.

5.9.2.1. 4.00 Bad Request

Like HTTP 400 "Bad Request".

5.9.2.2. 4.01 Unauthorized

The client is not authorized to perform the requested action. The client SHOULD NOT repeat the request without previously improving its authentication status to the server. Which specific mechanism can be used for this is outside this document’s scope; see also Section 9.

5.9.2.3. 4.02 Bad Option

The request could not be understood by the server due to one or more unrecognized or malformed options. The client SHOULD NOT repeat the request without modification.

5.9.2.4. 4.03 Forbidden

Like HTTP 403 "Forbidden".

5.9.2.5. 4.04 Not Found

Like HTTP 404 "Not Found".

5.9.2.6. 4.05 Method Not Allowed

Like HTTP 405 "Method Not Allowed", but with no parallel to the "Allow" header field.

5.9.2.7. 4.06 Not Acceptable

Like HTTP 406 "Not Acceptable", but with no response entity.

5.9.2.8. 4.12 Precondition Failed

Like HTTP 412 "Precondition Failed".

5.9.2.9. 4.13 Request Entity Too Large

Like HTTP 413 "Request Entity Too Large".

5.9.2.10. 4.15 Unsupported Content-Format

Like HTTP 415 "Unsupported Media Type".

5.9.3. Server Error 5.xx

This class of response code indicates cases in which the server is aware that it has erred or is incapable of performing the request. These response codes are applicable to any request method.

The server SHOULD include a diagnostic message as detailed in Section 5.5.2.

Responses of this class are cacheable: Caches can use the Max-Age Option to determine freshness (see Section 5.6.1). They cannot be validated.

5.9.3.1. 5.00 Internal Server Error

Like HTTP 500 "Internal Server Error".

5.9.3.2. 5.01 Not Implemented

Like HTTP 501 "Not Implemented".

5.9.3.3. 5.02 Bad Gateway

Like HTTP 502 "Bad Gateway".

5.9.3.4. 5.03 Service Unavailable

Like HTTP 503 "Service Unavailable", but using the Max-Age Option in place of the "Retry-After" header field to indicate the number of seconds after which to retry.

5.9.3.5. 5.04 Gateway Timeout

Like HTTP 504 "Gateway Timeout".

5.9.3.6. 5.05 Proxying Not Supported

The server is unable or unwilling to act as a forward-proxy for the URI specified in the Proxy-Uri Option (see Section 5.10.3).

5.10. Option Definitions

The individual CoAP options are summarized in Table 1 and explained below.

No.	C	U	N	R	Name	Format	Length	Default
1	x			x	If-Match	opaque	0-8	(none)
3	x	x			Uri-Host	string	1-255	(see below)
4				x	ETag	opaque	1-8	(none)
5	x				If-None-Match	empty	0	(none)
7	x	x			Uri-Port	uint	0-2	(see below)
8				x	Location-Path	string	0-255	(none)
11	x	x		x	Uri-Path	string	0-255	(none)
12					Content-Format	uint	0-2	(none)
14		x			Max-Age	uint	0-4	60
15	x	x		x	Uri-Query	string	1-255	(none)
16				x	Accept	uint	0-2	(none)
19	x	x			Token	opaque	1-8	(empty)
20				x	Location-Query	string	0-255	(none)
35	x	x			Proxy-Uri	string	1-1034	(none)

C=Critical, U=Unsafe, N=No-Cache-Key, R=Repeatable

Table 1: Options

Temporary Note: At the time of submission, the great renumbering is not yet reflected in [I-D.ietf-core-block] and [I-D.ietf-core-observe]. The new numbers are: 6 for Observe, 23 for Block2, 27 for Block1, and 28 for Size. This note to be removed when the satellite drafts are updated.

5.10.1. Token

The Token Option is used to match a response with a request. Every request has a client-generated token which the server MUST echo in any response. The token value is a sequence of 0 to 8 bytes. A default value of the zero-length token is assumed in the absence of the option. A value of 1 to 8 bytes can be sent as an option value. Thus when the token value is empty, the Token Option MUST be elided.

A token is intended for use as a client-local identifier for differentiating between concurrent requests (see Section 5.3). A client SHOULD generate tokens in a way that tokens currently in use for a given source/destination pair are unique. An empty token value is appropriate e.g. when no other tokens are in use to a destination, or when requests are made serially per destination. There are however multiple possible implementation strategies to fulfill this. An endpoint receiving a token MUST treat it as opaque and make no assumptions about its format.

5.10.2. Uri-Host, Uri-Port, Uri-Path and Uri-Query

The Uri-Host, Uri-Port, Uri-Path and Uri-Query Options are used to specify the target resource of a request to a CoAP origin server. The options encode the different components of the request URI in a way that no percent-encoding is visible in the option values and that the full URI can be reconstructed at any involved endpoint. The syntax of CoAP URIs is defined in Section 6.

The steps for parsing URIs into options is defined in Section 6.4. These steps result in zero or more Uri-Host, Uri-Port, Uri-Path and Uri-Query Options being included in a request, where each option holds the following values:

- o the Uri-Host Option specifies the Internet host of the resource being requested,
- o the Uri-Port Option specifies the transport layer port number of the resource,
- o each Uri-Path Option specifies one segment of the absolute path to the resource, and
- o each Uri-Query Option specifies one argument parameterizing the resource.

Note: Fragments ([RFC3986], Section 3.5) are not part of the request URI and thus will not be transmitted in a CoAP request.

The default value of the Uri-Host Option is the IP literal representing the destination IP address of the request message. Likewise, the default value of the Uri-Port Option is the destination UDP port. The default values for the Uri-Host and Uri-Port Options are sufficient for requests to most servers. Explicit Uri-Host and Uri-Port Options are typically used when an endpoint hosts multiple virtual servers.

The Uri-Path and Uri-Query Option can contain any character sequence. No percent-encoding is performed. The value of a Uri-Path Option MUST NOT be "." or ".." (as the request URI must be resolved before parsing it into options).

The steps for constructing the request URI from the options are defined in Section 6.5. Note that an implementation does not necessarily have to construct the URI; it can simply look up the target resource by looking at the individual options.

Examples can be found in Appendix B.

5.10.3. Proxy-Uri

The Proxy-Uri Option is used to make a request to a forward-proxy (see Section 5.7). The forward-proxy is requested to forward the request or service it from a valid cache, and return the response.

The option value is an absolute-URI ([RFC3986], Section 4.3).

Note that the forward-proxy MAY forward the request on to another proxy or directly to the server specified by the absolute-URI. In order to avoid request loops, a proxy MUST be able to recognize all of its server names, including any aliases, local variations, and the numeric IP addresses.

An endpoint receiving a request with a Proxy-Uri Option that is unable or unwilling to act as a forward-proxy for the request MUST cause the return of a 5.05 (Proxying Not Supported) response.

The Proxy-Uri Option MUST take precedence over any of the Uri-Host, Uri-Port, Uri-Path or Uri-Query options (which MUST NOT be included at the same time in a request containing the Proxy-Uri Option).

5.10.4. Content-Format

The Content-Format Option indicates the representation format of the message payload. The representation format is given as a numeric content format identifier that is defined in the CoAP Content Format registry (Section 12.3). In the absence of the option, no default value is assumed, i.e. the representation format of any representation message payload is indeterminate (Section 5.5).

5.10.5. Accept

The CoAP Accept option indicates when included one or more times in a request, one or more Content-Formats, each of which is an acceptable Content-Format for the client, in the order of preference (most preferred first). The representation format is given as a numeric Content-Format identifier that is defined in the CoAP Content-Format registry (Section 12.3). If no Accept options are given, the client does not express a preference (thus no default value is assumed). The client prefers the representation returned by the server to be in one of the Content-Formats indicated. The server SHOULD return one of the preferred Content-Formats if available. If none of the preferred Content-Formats can be returned, then a 4.06 "Not Acceptable" SHOULD be sent as a response.

Note that as a server might not support the Accept option (and thus would ignore it as it is elective), the client needs to be prepared to receive a representation in a different Content-Format. The client can simply discard a representation it can not make use of.

5.10.6. Max-Age

The Max-Age Option indicates the maximum time a response may be cached before it MUST be considered not fresh (see Section 5.6.1).

The option value is an integer number of seconds between 0 and 2**32-1 inclusive (about 136.1 years). A default value of 60 seconds is assumed in the absence of the option in a response.

The value is intended to be current at the time of transmission. Servers that provide resources with strict tolerances on the value of Max-Age SHOULD update the value before each retransmission. (See also Section 5.7.1.)

5.10.7. ETag

The ETag Option in a response provides the current value of the entity-tag for the enclosed representation of the target resource.

An entity-tag is intended for use as a resource-local identifier for differentiating between representations of the same resource that vary over time. It may be generated in any number of ways including a version, checksum, hash or time. An endpoint receiving an entity-tag MUST treat it as opaque and make no assumptions about its format. (Endpoints generating an entity-tag are encouraged to use the most compact representation possible, in particular in regards to clients and intermediaries that may want to store multiple ETag values.)

An endpoint that has one or more representations previously obtained from the resource can specify the ETag Option in a request for each stored response to determine if any of those representations is current (see Section 5.6.2).

The ETag Option MUST NOT occur more than once in a response, and MAY occur one or more times in a request.

5.10.8. Location-Path and Location-Query

The Location-Path and Location-Query Options together indicate a relative URI that consists either of an absolute path, a query string or both. A combination of these options is included in a 2.01 (Created) response to indicate the location of the resource created as the result of a POST request (see Section 5.8.2). The location is resolved relative to the request URI.

If a response with one or more Location-Path and/or Location-Query Options passes through a cache and the implied URI identifies one or more currently stored responses, those entries SHOULD be marked as not fresh.

Each Location-Path Option specifies one segment of the absolute path to the resource, and each Location-Query Option specifies one argument parameterizing the resource. The Location-Path and Location-Query Option can contain any character sequence. No percent-encoding is performed. The value of a Location-Path Option MUST NOT be "." or "..".

The steps for constructing the location URI from the options are analogous to Section 6.5, except that the first five steps are skipped and the result is a relative URI-reference.

More Location-* options may be defined in the future, and have been reserved option numbers 128, 132 and 136. If any of these reserved option numbers occurs in addition to Location-Path and/or Location-Query and are not supported, then a 4.02 (Bad Option) error MUST be returned.

5.10.9. If-Match

The If-Match Option MAY be used to make a request conditional on the current existence or value of an ETag for one or more representations of the target resource. If-Match is generally useful for resource update requests, such as PUT requests, as a means for protecting against accidental overwrites when multiple clients are acting in parallel on the same resource (i.e., the "lost update" problem).

The value of an If-Match option is either an ETag or the empty string. An If-Match option with an ETag matches a representation with that exact ETag. An If-Match option with an empty value matches any existing representation (i.e., it places the precondition on the existence of any current representation for the target resource).

The If-Match Option can occur multiple times. If any of the options match, then the server performs the request method as if the set of If-Match Options were not present.

If there is one or more If-Match Option, but none of the options match, the server MUST NOT perform the requested method. Instead, the server MUST respond with the 4.12 (Precondition Failed) response code.

If the request would, without the If-Match Options, result in anything other than a 2.xx or 4.12 response code, then any If-Match Options MUST be ignored.

5.10.10. If-None-Match

The If-None-Match Option MAY be used to make a request conditional on the non-existence of the target resource. If-None-Match is useful for resource creation requests, such as PUT requests, as a means for protecting against accidental overwrites when multiple clients are acting in parallel on the same resource. The If-None-Match Option carries no value.

If the target resource does exist, then the server MUST NOT perform the requested method. Instead, the server MUST respond with the 4.12 (Precondition Failed) response code.

6. CoAP URIs

CoAP uses the "coap" and "coaps" URI schemes for identifying CoAP resources and providing a means of locating the resource. Resources are organized hierarchically and governed by a potential CoAP origin server listening for CoAP requests ("coap") or DTLS-secured CoAP requests ("coaps") on a given UDP port. The CoAP server is identified via the generic syntax’s authority component, which includes a host component and optional UDP port number. The remainder of the URI is considered to be identifying a resource which can be operated on by the methods defined by the CoAP protocol. The "coap" and "coaps" URI schemes can thus be compared to the "http" and "https" URI schemes respectively.

The syntax of the "coap" and "coaps" URI schemes is specified below in Augmented Backus-Naur Form (ABNF) [RFC5234]. The definitions of "host", "port", "path-abempty", "query", "segment", "IP-literal", "IPv4address" and "reg-name" are adopted from [RFC3986].

Implementation Note: Unfortunately, over time the URI format has acquired significant complexity. Implementers are encouraged to examine [RFC3986] closely. E.g., the ABNF for IPv6 addresses is more complicated than maybe expected. Also, implementers should take care to perform the processing of percent decoding/encoding exactly once on the way from a URI to its decoded components or back. Percent encoding is crucial for data transparency, but may lead to unusual results such as a slash in a path component.

6.1. coap URI Scheme

```
coap-URI = "coap:" "//" host [ ":" port ] path-abempty [ "?" query ]
```

If the host component is provided as an IP-literal or IPv4address, then the CoAP server can be reached at that IP address. If host is a registered name, then that name is considered an indirect identifier and the endpoint might use a name resolution service, such as DNS, to find the address of that host. The host MUST NOT be empty; if a URI is received with a missing authority or an empty host, then it MUST be considered invalid. The port subcomponent indicates the UDP port at which the CoAP server is located. If it is empty or not given, then the default port 5683 is assumed.

The path identifies a resource within the scope of the host and port. It consists of a sequence of path segments separated by a slash character (U+002F SOLIDUS "/").

The query serves to further parameterize the resource. It consists of a sequence of arguments separated by an ampersand character (U+0026 AMPERSAND "&"). An argument is often in the form of a "key=value" pair.

The "coap" URI scheme supports the path prefix "/.well-known/" defined by [RFC5785] for "well-known locations" in the name-space of a host. This enables discovery of policy or other information about a host ("site-wide metadata"), such as hosted resources (see Section 7).

Application designers are encouraged to make use of short, but descriptive URIs. As the environments that CoAP is used in are usually constrained for bandwidth and energy, the trade-off between these two qualities should lean towards the shortness, without ignoring descriptiveness.

6.2. coaps URI Scheme

```
coaps-URI = "coaps:" "//" host [ ":" port ] path-abempty
            [ "?" query ]
```

All of the requirements listed above for the "coap" scheme are also requirements for the "coaps" scheme, except that a default UDP port of [IANA_TBD_PORT] is assumed if the port subcomponent is empty or not given, and the UDP datagrams MUST be secured for privacy through the use of DTLS as described in Section 9.1.

Unlike the "coap" scheme, responses to "coaps" identified requests are never "public" and thus MUST NOT be reused for shared caching unless the cache is able to make equivalent access control decisions to the ones that led to the cached entry (Section 11.2). They can, however, be reused in a private cache if the message is cacheable by default in CoAP.

Resources made available via the "coaps" scheme have no shared identity with the "coap" scheme even if their resource identifiers indicate the same authority (the same host listening to the same UDP port). They are distinct name spaces and are considered to be distinct origin servers.

6.3. Normalization and Comparison Rules

Since the "coap" and "coaps" schemes conform to the URI generic syntax, such URIs are normalized and compared according to the algorithm defined in [RFC3986], Section 6, using the defaults described above for each scheme.

If the port is equal to the default port for a scheme, the normal form is to elide the port subcomponent. Likewise, an empty path component is equivalent to an absolute path of "/", so the normal form is to provide a path of "/" instead. The scheme and host are case-insensitive and normally provided in lowercase; IP-literals are in recommended form [RFC5952]; all other components are compared in a case-sensitive manner. Characters other than those in the "reserved" set are equivalent to their percent-encoded octets (see [RFC3986], Section 2.1): the normal form is to not encode them.

For example, the following three URIs are equivalent, and cause the same options and option values to appear in the CoAP messages:

```
coap://example.com:5683/~sensors/temp.xml
coap://EXAMPLE.com/%7Esensors/temp.xml
coap://EXAMPLE.com:/%7esensors/temp.xml
```

6.4. Decomposing URIs into Options

The steps to parse a request's options from a string /url/ are as follows. These steps either result in zero or more of the Uri-Host, Uri-Port, Uri-Path and Uri-Query Options being included in the request, or they fail.

1. If the /url/ string is not an absolute URI ([RFC3986]), then fail this algorithm.

2. Resolve the /url/ string using the process of reference resolution defined by [RFC3986], with the URL character encoding set to UTF-8 [RFC3629].

   NOTE: It doesn't matter what it is resolved relative to, since we already know it is an absolute URL at this point.

3. If /url/ does not have a <scheme> component whose value, when converted to ASCII lowercase, is "coap" or "coaps", then fail this algorithm.

4. If /url/ has a <fragment> component, then fail this algorithm.

5. If the <host> component of /url/ does not represent the request's destination IP address as an IP-literal or IPv4address, include a Uri-Host Option and let that option's value be the value of the <host> component of /url/, converted to ASCII lowercase, and then converting all percent-encodings ("%" followed by two hexadecimal digits) to the corresponding characters.

   NOTE: In the usual case where the request's destination IP address is derived from the host part, this ensures that a Uri-Host Option is only used for a <host> component of the form reg-name.

6. If /url/ has a <port> component, then let /port/ be that component's value interpreted as a decimal integer; otherwise, let /port/ be the default port for the scheme.

7. If /port/ does not equal the request's destination UDP port, include a Uri-Port Option and let that option's value be /port/.

8. If the value of the <path> component of /url/ is empty or consists of a single slash character (U+002F SOLIDUS "/"), then

move to the next step.

Otherwise, for each segment in the <path> component, include a Uri-Path Option and let that option's value be the segment (not including the delimiting slash characters) after converting all percent-encodings ("%" followed by two hexadecimal digits) to the corresponding characters.

9. If /url/ has a <query> component, then, for each argument in the <query> component, include a Uri-Query Option and let that option's value be the argument (not including the question mark and the delimiting ampersand characters) after converting all percent-encodings to the corresponding characters.

Note that these rules completely resolve any percent-encoding.

6.5. Composing URIs from Options

The steps to construct a URI from a request's options are as follows. These steps either result in a URI, or they fail. In these steps, percent-encoding a character means replacing each of its (UTF-8 encoded) bytes by a "%" character followed by two hexadecimal digits representing the byte, where the digits A-F are in upper case (as defined in [RFC3986] Section 2.1; to reduce variability, the hexadecimal notation for percent-encoding in CoAP URIs MUST use uppercase letters). The definitions of "unreserved" and "sub-delims" are adopted from [RFC3986].

1. If the request is secured using DTLS, let /url/ be the string "coaps://". Otherwise, let /url/ be the string "coap://".

2. If the request includes a Uri-Host Option, let /host/ be that option's value, where any non-ASCII characters are replaced by their corresponding percent-encoding. If /host/ is not a valid reg-name or IP-literal or IPv4address, fail the algorithm. If the request does not include a Uri-Host Option, let /host/ be the IP-literal (making use of the conventions of [RFC5952]) or IPv4address representing the request's destination IP address.

3. Append /host/ to /url/.

4. If the request includes a Uri-Port Option, let /port/ be that option's value. Otherwise, let /port/ be the request's destination UDP port.

5. If /port/ is not the default port for the scheme, then append a single U+003A COLON character (:) followed by the decimal representation of /port/ to /url/.

6. Let /resource name/ be the empty string. For each Uri-Path Option in the request, append a single character U+002F SOLIDUS (/) followed by the option's value to /resource name/, after converting any character that is not either in the "unreserved" set, "sub-delims" set, a U+003A COLON (:) or U+0040 COMMERCIAL AT (@) character, to its percent-encoded form.

7. If /resource name/ is the empty string, set it to a single character U+002F SOLIDUS (/).

8. For each Uri-Query Option in the request, append a single character U+003F QUESTION MARK (?) (first option) or U+0026 AMPERSAND (&) (subsequent options) followed by the option's value to /resource name/, after converting any character that is not either in the "unreserved" set, "sub-delims" set (except U+0026 AMPERSAND (&)), a U+003A COLON (:), U+0040 COMMERCIAL AT (@), U+002F SOLIDUS (/) or U+003F QUESTION MARK (?) character, to its percent-encoded form.

9. Append /resource name/ to /url/.

10. Return /url/.

Note that these steps have been designed to lead to a URI in normal form (see Section 6.3).

7. Discovery

7.1. Service Discovery

A server is discovered by a client by the client knowing or learning a URI that references a resource in the namespace of the server. Alternatively, clients can use Multicast CoAP (see Section 8) and the "All CoAP Nodes" multicast address to find CoAP servers.

Unless the port subcomponent in a "coap" or "coaps" URI indicates the UDP port at which the CoAP server is located, the server is assumed to be reachable at the default port.

The CoAP default port number 5683 MUST be supported by a server that offers resources for resource discovery (see Section 7.2 below) and SHOULD be supported for providing access to other resources. The default port number [IANA_TBD_PORT] for DTLS-secured CoAP MAY be supported by a server for resource discovery and for providing access to other resources. In addition other endpoints may be hosted at other ports, e.g. in the dynamic port space.

Implementation Note: When a CoAP server is hosted by a 6LoWPAN node, header compression efficiency is improved when it also supports a port number in the 61616-61631 compressed UDP port space defined in [RFC4944] (note that, as its UDP port differs from the default port, it is a different endpoint from the server at the default port).

7.2. Resource Discovery

The discovery of resources offered by a CoAP endpoint is extremely important in machine-to-machine applications where there are no humans in the loop and static interfaces result in fragility. A CoAP endpoint SHOULD support the CoRE Link Format of discoverable resources as described in [RFC6690]. It is up to the server which resources are made discoverable (if any).

7.2.1. 'ct' Attribute

This section defines a new Web Linking [RFC5988] attribute for use with [RFC6690]. The Content-Format code "ct" attribute provides a hint about the Content-Formats this resource returns. Note that this is only a hint, and does not override the Content-Format Option of a CoAP response obtained by actually requesting the representation of the resource. The value is in the CoAP identifier code format as a decimal ASCII integer and MUST be in the range of 0-65535 (16-bit unsigned integer). For example application/xml would be indicated as "ct=41". If no Content-Format code attribute is present then nothing about the type can be assumed. The Content-Format code attribute MAY include a space-separated sequence of Content-Format codes, indicating that multiple content-formats are available. The syntax of the attribute value is summarized in the production ct-value in Figure 13, where cardinal, SP and DQUOTE are defined as in [RFC6690].

```
ct-value =  cardinal
         /  DQUOTE cardinal *( 1*SP cardinal ) DQUOTE
```

Figure 13

8. Multicast CoAP

CoAP supports making requests to a IP multicast group. This is defined by a series of deltas to Unicast CoAP.

CoAP endpoints that offer services that they want other endpoints to be able to find using multicast service discovery, join one or more of the appropriate all-CoAP-nodes multicast addresses Section 12.8 and listen on the default CoAP port. Note that an endpoint might receive multicast requests on other multicast addresses, including the all-nodes IPv6 address (or via broadcast on IPv4); an endpoint MUST therefore be prepared to receive such messages but MAY ignore them if multicast service discovery is not desired.

8.1. Messaging Layer

A multicast request is characterized by being transported in a CoAP message that is addressed to an IP multicast address instead of a CoAP endpoint. Such multicast requests MUST be Non-Confirmable.

A server SHOULD be aware that a request arrived via multicast, e.g. by making use of modern APIs such as IPV6_RECVPKTINFO [RFC3542], if available.

When a server is aware that a request arrived via multicast, it MUST NOT return a RST in reply to NON. If it is not aware, it MAY return a RST in reply to NON as usual. Because such a Reset message will look identical to an RST for a unicast message from the sender, the sender MUST avoid using a Message ID that is also still active from this endpoint with any unicast endpoint that might receive the multicast message.

8.2. Request/Response Layer

When a server is aware that a request arrived via multicast, the server MAY always pretend it did not receive the request, in particular if it doesn’t have anything useful to respond (e.g., if it only has an empty payload or an error response). The decision for this may depend on the application. (For example, in [RFC6690] query filtering, a server should not respond to a multicast request if the filter does not match.)

If a server does decide to respond to a multicast request, it should not respond immediately. Instead, it should pick a duration for the period of time during which it intends to respond. For purposes of this exposition, we call the length of this period the Leisure. The specific value of this Leisure may depend on the application, or MAY be derived as described below. The server SHOULD then pick a random point of time within the chosen Leisure period to send back the unicast response to the multicast request. If further responses need to be sent based on the same multicast address membership, a new leisure period starts at the earliest after the previous one finishes.

To compute a value for Leisure, the server should have a group size estimate G, a target data transfer rate R (which both should be chosen conservatively) and an estimated response size S; a rough lower bound for Leisure can then be computed as

$$lb\_Leisure = S * G / R$$

E.g., for a multicast request with link-local scope on an 2.4 GHz IEEE 802.15.4 (6LoWPAN) network, G could be (relatively conservatively) set to 100, S to 100 bytes, and the target rate to a conservative 8 kbit/s = 1 kB/s. The resulting lower bound for the Leisure is 10 seconds.

If a CoAP endpoint does not have suitable data to compute a value for Leisure, it MAY resort to DEFAULT_LEISURE.

When matching a response to a multicast request, only the token MUST match; the source endpoint of the response does not need to (and will not) be the same as the destination endpoint of the original request.

8.2.1. Caching

When a client makes a multicast request, it always makes a new request to the multicast group (since there may be new group members that joined meanwhile or ones that did not get the previous request). It MAY update the cache with the received responses. Then it uses both cached-still-fresh and 'new' responses as the result of the request.

A response received in reply to a GET request to a multicast group MAY be used to satisfy a subsequent request on the related unicast request URI. The unicast request URI is obtained by replacing the authority part of the request URI with the transport layer source address of the response message.

A cache MAY revalidate a response by making a GET request on the related unicast request URI.

A GET request to a multicast group MUST NOT contain an ETag option. A mechanism to suppress responses the client already has is left for further study.

8.2.2. Proxying

When a forward-proxy receives a request with a Proxy-Uri that indicates a multicast address, the proxy obtains a set of responses as described above and sends all responses (both cached-still-fresh and new) back to the original client.

9. Securing CoAP

This section defines the DTLS binding for CoAP, and the alternative use of IPsec.

During the provisioning phase, a CoAP device is provided with the security information that it needs, including keying materials and access control lists. This specification defines provisioning for the RawPublicKey mode in Section 9.1.3.2.1. At the end of the provisioning phase, the device will be in one of four security modes with the following information for the given mode. The NoSec and RawPublicKey modes are mandatory to implement for this specification.

NoSec: There is no protocol level security (DTLS is disabled). Alternative techniques to provide lower layer security SHOULD be used when appropriate. The use of IPsec is discussed in Section 9.2.

PreSharedKey: DTLS is enabled and there is a list of pre-shared keys [RFC4279] and each key includes a list of which nodes it can be used to communicate with as described in Section 9.1.3.1. At the extreme there may be one key for each node this CoAP node needs to communicate with (1:1 node/key ratio).

RawPublicKey: DTLS is enabled and the device has an asymmetric key pair without a certificate (a raw public key) that is validated using an out-of-band mechanism [I-D.ietf-tls-oob-pubkey] as described in Section 9.1.3.2. The device also has an identity calculated from the public key and a list of identities of the nodes it can communicate with.

Certificate: DTLS is enabled and the device has an asymmetric key pair with an X.509 certificate [RFC5280] that binds it to its Authority Name and is signed by some common trust root as described in Section 9.1.3.3. The device also has a list of root trust anchors that can be used for validating a certificate.

In the "NoSec" mode, the system simply sends the packets over normal UDP over IP and is indicated by the "coap" scheme and the CoAP default port. The system is secured only by keeping attackers from being able to send or receive packets from the network with the CoAP nodes; see Section 11.5 for an additional complication with this approach.

The other three security modes are achieved using DTLS and are indicated by the "coaps" scheme and DTLS-secured CoAP default port. The result is a security association that can be used to authenticate (within the limits of the security model) and, based on this authentication, authorize the communication partner. CoAP itself does not provide protocol primitives for authentication or authorization; where this is required, it can either be provided by communication security (i.e., IPsec or DTLS) or by object security (within the payload). Devices that require authorization for certain operations are expected to require one of these two forms of security. Necessarily, where an intermediary is involved, communication security only works when that intermediary is part of the trust relationships; CoAP does not provide a way to forward different levels of authorization that clients may have with an intermediary to further intermediaries or origin servers -- it therefore may be required to perform all authorization at the first intermediary.

9.1. DTLS-secured CoAP

Just as HTTP is secured using Transport Layer Security (TLS) over TCP, CoAP is secured using Datagram TLS (DTLS) [RFC6347] over UDP (see Figure 14). This section defines the CoAP binding to DTLS, along with the minimal mandatory-to-implement configurations appropriate for constrained environments. The binding is defined by a series of deltas to Unicast CoAP. DTLS is in practice TLS with added features to deal with the unreliable nature of the UDP transport.

```
+----------------------+
|      Application     |
+----------------------+
+----------------------+
|  Requests/Responses  |
|----------------------|  CoAP
|       Messages       |
+----------------------+
+----------------------+
|         DTLS         |
+----------------------+
+----------------------+
|          UDP         |
+----------------------+
```

Figure 14: Abstract layering of DTLS-secured CoAP

In some constrained nodes (limited flash and/or RAM) and networks (limited bandwidth or high scalability requirements), and depending on the specific cipher suites in use, all modes of DTLS may not be applicable. Some DTLS cipher suites can add significant implementation complexity as well as some initial handshake overhead needed when setting up the security association. Once the initial handshake is completed, DTLS adds a limited per-datagram overhead of approximately 13 bytes, not including any initialization vectors/ nonces (e.g., 8 bytes with TLS_PSK_WITH_AES_128_CCM_8 [RFC6655]), integrity check values (e.g., 8 bytes with TLS_PSK_WITH_AES_128_CCM_8 [RFC6655]) and padding required by the cipher suite. Whether and which mode of using DTLS is applicable for a CoAP-based application should be carefully weighed considering the specific cipher suites that may be applicable, and whether the session maintenance makes it compatible with application flows and sufficient resources are available on the constrained nodes and for the added network overhead. DTLS is not applicable to group keying (multicast communication); however, it may be a component in a future group key management protocol.

9.1.1. Messaging Layer

The endpoint acting as the CoAP client should also act as the DTLS client. It should initiate a session to the server on the appropriate port. When the DTLS handshake has finished, the client may initiate the first CoAP request. All CoAP messages MUST be sent as DTLS "application data".

The following rules are added for matching an ACK or RST to a CON message or a RST to a NON message: The DTLS session MUST be the same and the epoch MUST be the same.

A message is the same when it is sent within the same DTLS session and same epoch and has the same Message ID.

Note: When a confirmable message is retransmitted, a new DTLS sequence_number is used for each attempt, even though the CoAP Message ID stays the same. So a recipient still has to perform deduplication as described in Section 4.5. Retransmissions MUST NOT be performed across epochs.

DTLS connections in RawPublicKey and Certificate mode are set up using mutual authentication so they can remain up and be reused for future message exchanges in either direction. Devices can close a DTLS connection when they need to recover resources but in general they should keep the connection up for as long as possible. Closing the DTLS connection after every CoAP message exchange is very inefficient.

9.1.2. Request/Response Layer

The following rules are added for matching a response to a request: The DTLS session MUST be the same and the epoch MUST be the same.

9.1.3. Endpoint Identity

Devices SHOULD support the Server Name Indication (SNI) to indicate their Authority Name in the SNI HostName field as defined in Section 3 of [RFC6066]. This is needed so that when a host that acts as a virtual server for multiple Authorities receives a new DTLS connection, it knows which keys to use for the DTLS session.

9.1.3.1. Pre-Shared Keys

When forming a connection to a new node, the system selects an appropriate key based on which nodes it is trying to reach and then forms a DTLS session using a PSK (Pre-Shared Key) mode of DTLS. Implementations in these modes MUST support the mandatory to implement cipher suite TLS_PSK_WITH_AES_128_CCM_8 as specified in [RFC6655].

The security considerations of [RFC4279] (Section 7) apply. In particular, applications should carefully weigh whether they need Perfect Forward Secrecy (PFS) or not and select an appropriate cipher suite (7.1). The entropy of the PSK must be sufficient to mitigate against brute-force and (where the PSK is not chosen randomly but by a human) dictionary attacks (7.2). The cleartext communication of client identities may leak data or compromise privacy (7.3).

9.1.3.2. Raw Public Key Certificates

In this mode the device has an asymmetric key pair but without an X.509 certificate (called a raw public key). A device MAY be configured with multiple raw public keys. The type and length of the raw public key depends on the cipher suite used. Implementations in RawPublicKey mode MUST support the mandatory to implement cipher suite TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8 as specified in [I-D.mcgrew-tls-aes-ccm-ecc], [RFC5246], [RFC4492]. The mechanism for using raw public keys with TLS is specified in [I-D.ietf-tls-oob-pubkey].

9.1.3.2.1. Provisioning

The RawPublicKey mode was designed to be easily provisioned in M2M deployments. It is assumed that each device has an appropriate asymmetric public key pair installed. An identifier is calculated from the public key as described in Section 2 of [I-D.farrell-decade-ni]. All implementations that support checking RawPublicKey identities MUST support at least the sha-256-120 mode (SHA-256 truncated to 120 bits). Implementations SHOULD support also longer length identifiers and MAY support shorter lengths. Note that the shorter lengths provide less security against attacks and their use is NOT RECOMMENDED.

Depending on how identifiers are given to the system that verifies them, support for URI, binary, and/or human-speakable format [I-D.farrell-decade-ni] needs to be implemented. All implementations SHOULD support the binary mode and implementations that have a user interface SHOULD also support the human-speakable format.

During provisioning, the identifier of each node is collected, for example by reading a barcode on the outside of the device or by obtaining a pre-compiled list of the identifiers. These identifiers are then installed in the corresponding endpoint, for example an M2M data collection server. The identifier is used for two purposes, to associate the endpoint with further device information and to perform access control. During provisioning, an access control list of identifiers the device may start DTLS sessions with SHOULD also be installed.

9.1.3.3. X.509 Certificates

Implementations in Certificate Mode MUST support the mandatory to implement cipher suite TLS_ECDHE_ECDSA_WITH_AES_128_CCM_8 as specified in [RFC5246].

The Authority Name in the certificate is the name that would be used in the Authority part of a CoAP URI. It is worth noting that this would typically not be either an IP address or DNS name but would instead be a long term unique identifier for the device such as the EUI-64 [EUI64]. The discovery process used in the system would build up the mapping between IP addresses of the given devices and the Authority Name for each device. Some devices could have more than one Authority and would need more than a single certificate.

When a new connection is formed, the certificate from the remote device needs to be verified. If the CoAP node has a source of absolute time, then the node SHOULD check that the validity dates of the certificate are within range. The certificate MUST also be signed by an appropriate chain of trust. If the certificate contains a SubjectAltName, then the Authority Name MUST match at least one of the authority names of any CoAP URI found in a URI type field in the SubjectAltName set. If there is no SubjectAltName in the certificate, then the Authoritative Name must match the CN found in the certificate using the matching rules defined in [RFC2818] with the exception that certificates with wildcards are not allowed.

If the system has a shared key in addition to the certificate, then a cipher suite that includes the shared key such as TLS_RSA_PSK_WITH_AES_128_CBC_SHA [RFC4279] SHOULD be used.

9.2. Using CoAP with IPsec

One mechanism to secure CoAP in constrained environments is the IPsec Encapsulating Security Payload (ESP) [RFC4303] when CoAP is used without DTLS in NoSec Mode. Using IPsec ESP with the appropriate configuration, it is possible for many constrained devices to support encryption with built-in link-layer encryption hardware. For example, some IEEE 802.15.4 radio chips are compatible with AES-CBC (with 128-bit keys) [RFC3602] as defined for use with IPsec in [RFC4835]. Alternatively, particularly on more common IEEE 802.15.4 hardware that supports AES encryption but not decryption, and to avoid the need for padding, nodes could directly use the more widely supported AES-CCM as defined for use with IPsec in [RFC4309], if the security considerations in Section 9 of that specification can be fulfilled.

Necessarily for AES-CCM, but much preferably also for AES-CBC, static keying should be avoided and the initial keying material be derived into transient session keys, e.g. using a low-overhead mode of IKEv2 [RFC5996] as described in [I-D.kivinen-ipsecme-ikev2-minimal]; such a protocol for managing keys and sequence numbers is also the only way to achieve anti-replay capabilities. However, no recommendation can be made at this point on how to manage group keys (i.e., for multicast) in a constrained environment. Once any initial setup is completed, IPsec ESP adds a limited overhead of approximately 10 bytes per packet, not including initialization vectors, integrity check values and padding required by the cipher suite.

When using IPsec to secure CoAP, both authentication and confidentiality SHOULD be applied as recommended in [RFC4303]. The use of IPsec between CoAP endpoints is transparent to the application layer and does not require special consideration for a CoAP implementation.

IPsec may not be appropriate for all environments. For example, IPsec support is not available for many embedded IP stacks and even in full PC operating systems or on back-end web servers, application developers may not have sufficient access to configure or enable IPsec or to add a security gateway to the infrastructure. Problems with firewalls and NATs may furthermore limit the use of IPsec.

10. Cross-Protocol Proxying between CoAP and HTTP

CoAP supports a limited subset of HTTP functionality, and thus cross-protocol proxying to HTTP is straightforward. There might be several reasons for proxying between CoAP and HTTP, for example when designing a web interface for use over either protocol or when realizing a CoAP-HTTP proxy. Likewise, CoAP could equally be proxied to other protocols such as XMPP [RFC6120] or SIP [RFC3264]; the definition of these mechanisms is out of scope of this specification.

There are two possible directions to access a resource via a forward-proxy:

CoAP-HTTP Proxying: Enables CoAP clients to access resources on HTTP servers through an intermediary. This is initiated by including the Proxy-Uri Option with an "http" or "https" URI in a CoAP request to a CoAP-HTTP proxy.

HTTP-CoAP Proxying: Enables HTTP clients to access resources on CoAP servers through an intermediary. This is initiated by specifying a "coap" or "coaps" URI in the Request-Line of an HTTP request to an HTTP-CoAP proxy.

Either way, only the Request/Response model of CoAP is mapped to HTTP. The underlying model of confirmable or non-confirmable messages, etc., is invisible and MUST have no effect on a proxy function. The following sections describe the handling of requests to a forward-proxy. Reverse proxies are not specified as the proxy function is transparent to the client with the proxy acting as if it was the origin server.

10.1. CoAP-HTTP Proxying

If a request contains a Proxy-URI Option with an 'http' or 'https' URI [RFC2616], then the receiving CoAP endpoint (called "the proxy" henceforth) is requested to perform the operation specified by the request method on the indicated HTTP resource and return the result to the client.

This section specifies for any CoAP request the CoAP response that the proxy should return to the client. How the proxy actually satisfies the request is an implementation detail, although the typical case is expected to be the proxy translating and forwarding the request to an HTTP origin server.

Since HTTP and CoAP share the basic set of request methods, performing a CoAP request on an HTTP resource is not so different from performing it on a CoAP resource. The meanings of the individual CoAP methods when performed on HTTP resources are explained below.

If the proxy is unable or unwilling to service a request with an HTTP URI, a 5.05 (Proxying Not Supported) response is returned to the client. If the proxy services the request by interacting with a third party (such as the HTTP origin server) and is unable to obtain a result within a reasonable time frame, a 5.04 (Gateway Timeout) response is returned; if a result can be obtained but is not understood, a 5.02 (Bad Gateway) response is returned.

10.1.1. GET

The GET method requests the proxy to return a representation of the HTTP resource identified by the request URI.

Upon success, a 2.05 (Content) response code SHOULD be returned. The payload of the response MUST be a representation of the target HTTP resource, and the Content-Format Option be set accordingly. The response MUST indicate a Max-Age value that is no greater than the remaining time the representation can be considered fresh. If the HTTP entity has an entity tag, the proxy SHOULD include an ETag Option in the response and process ETag Options in requests as described below.

A client can influence the processing of a GET request by including the following option:

Accept: The request MAY include one or more Accept Options, identifying the preferred response content-format.

ETag: The request MAY include one or more ETag Options, identifying responses that the client has stored. This requests the proxy to send a 2.03 (Valid) response whenever it would send a 2.05 (Content) response with an entity tag in the requested set otherwise. Note that CoAP ETags are always strong ETags in the HTTP sense; CoAP does not have the equivalent of HTTP weak ETags, and there is no good way to make use of these in a cross-proxy.

10.1.2. PUT

The PUT method requests the proxy to update or create the HTTP resource identified by the request URI with the enclosed representation.

If a new resource is created at the request URI, a 2.01 (Created) response MUST be returned to the client. If an existing resource is modified, a 2.04 (Changed) response MUST be returned to indicate successful completion of the request.

10.1.3. DELETE

The DELETE method requests the proxy to delete the HTTP resource identified by the request URI at the HTTP origin server.

A 2.02 (Deleted) response MUST be returned to client upon success or if the resource does not exist at the time of the request.

10.1.4. POST

The POST method requests the proxy to have the representation enclosed in the request be processed by the HTTP origin server. The actual function performed by the POST method is determined by the origin server and dependent on the resource identified by the request URI.

If the action performed by the POST method does not result in a resource that can be identified by a URI, a 2.04 (Changed) response MUST be returned to the client. If a resource has been created on the origin server, a 2.01 (Created) response MUST be returned.

10.2. HTTP-CoAP Proxying

If an HTTP request contains a Request-URI with a 'coap' or 'coaps' URI, then the receiving HTTP endpoint (called "the proxy" henceforth) is requested to perform the operation specified by the request method on the indicated CoAP resource and return the result to the client.

This section specifies for any HTTP request the HTTP response that the proxy should return to the client. How the proxy actually satisfies the request is an implementation detail, although the typical case is expected to be the proxy translating and forwarding the request to a CoAP origin server. The meanings of the individual HTTP methods when performed on CoAP resources are explained below.

If the proxy is unable or unwilling to service a request with a CoAP URI, a 501 (Not Implemented) response SHOULD be returned to the client. If the proxy services the request by interacting with a third party (such as the CoAP origin server) and is unable to obtain a result within a reasonable time frame, a 504 (Gateway Timeout) response SHOULD be returned; if a result can be obtained but is not understood, a 502 (Bad Gateway) response SHOULD be returned.

10.2.1. OPTIONS and TRACE

As the OPTIONS and TRACE methods are not supported in CoAP a 501 (Not Implemented) error MUST be returned to the client.

10.2.2. GET

The GET method requests the proxy to return a representation of the CoAP resource identified by the Request-URI.

Upon success, a 200 (OK) response SHOULD be returned. The payload of the response MUST be a representation of the target CoAP resource, and the Content-Type and Content-Encoding header fields be set accordingly. The response MUST indicate a max-age directive that indicates a value no greater than the remaining time the representation can be considered fresh. If the CoAP response has an ETag option, the proxy SHOULD include an ETag header field in the response.

A client can influence the processing of a GET request by including the following options:

Accept: Each individual Media-type of the HTTP Accept header in a request is mapped to a CoAP Accept option. HTTP Accept Media-type ranges, parameters and extensions are not supported by the CoAP Accept option. If the proxy cannot send a response which is acceptable according to the combined Accept field value, then the proxy SHOULD send a 406 (not acceptable) response.

Conditional GETs: Conditional HTTP GET requests that include an "If-Match" or "If-None-Match" request-header field can be mapped to a corresponding CoAP request. The "If-Modified-Since" and "If-Unmodified-Since" request-header fields are not directly supported by CoAP, but SHOULD be implemented locally by a caching proxy.

10.2.3. HEAD

The HEAD method is identical to GET except that the server MUST NOT return a message-body in the response.

Although there is no direct equivalent of HTTP’s HEAD method in CoAP, an HTTP-CoAP proxy responds to HEAD requests for CoAP resources, and the HTTP headers are returned without a message-body.

Implementation Note: An HTTP-CoAP proxy may want to try using a block-wise transfer [I-D.ietf-core-block] option to minimize the amount of data actually transferred, but needs to be prepared for the case that the origin server does not support block-wise transfers.

10.2.4. POST

The POST method requests the proxy to have the representation enclosed in the request be processed by the CoAP origin server. The actual function performed by the POST method is determined by the origin server and dependent on the resource identified by the request URI.

If the action performed by the POST method does not result in a resource that can be identified by a URI, a 200 (OK) or 204 (No Content) response MUST be returned to the client. If a resource has been created on the origin server, a 201 (Created) response MUST be returned.

If any of the Location-* Options are present in the CoAP response, a Location header field constructed from the values of these options SHOULD be returned.

10.2.5. PUT

The PUT method requests the proxy to update or create the CoAP resource identified by the Request-URI with the enclosed representation.

If a new resource is created at the Request-URI, a 201 (Created) response MUST be returned to the client. If an existing resource is modified, either the 200 (OK) or 204 (No Content) response codes SHOULD be sent to indicate successful completion of the request.

10.2.6. DELETE

The DELETE method requests the proxy to delete the CoAP resource identified by the Request-URI at the CoAP origin server.

A successful response SHOULD be 200 (OK) if the response includes an entity describing the status or 204 (No Content) if the action has been enacted but the response does not include an entity.

10.2.7. CONNECT

This method can not currently be satisfied by an HTTP-CoAP proxy function as TLS to DTLS tunneling has not yet been specified. For now, a 501 (Not Implemented) error SHOULD be returned to the client.

11. Security Considerations

This section analyzes the possible threats to the protocol. It is meant to inform protocol and application developers about the security limitations of CoAP as described in this document. As CoAP realizes a subset of the features in HTTP/1.1, the security considerations in Section 15 of [RFC2616] are also pertinent to CoAP. This section concentrates on describing limitations specific to CoAP.

11.1. Protocol Parsing, Processing URIs

A network-facing application can exhibit vulnerabilities in its processing logic for incoming packets. Complex parsers are well-known as a likely source of such vulnerabilities, such as the ability to remotely crash a node, or even remotely execute arbitrary code on it. CoAP attempts to narrow the opportunities for introducing such vulnerabilities by reducing parser complexity, by giving the entire range of encodable values a meaning where possible, and by aggressively reducing complexity that is often caused by unnecessary choice between multiple representations that mean the same thing. Much of the URI processing has been moved to the clients, further reducing the opportunities for introducing vulnerabilities into the servers. Even so, the URI processing code in CoAP implementations is likely to be a large source of remaining vulnerabilities and should be implemented with special care. The most complex parser remaining could be the one for the CoRE Link Format, although this also has been designed with a goal of reduced implementation complexity [RFC6690]. (See also section 15.2 of [RFC2616].)

11.2. Proxying and Caching

As mentioned in 15.7 of [RFC2616], proxies are by their very nature men-in-the-middle, breaking any IPsec or DTLS protection that a direct CoAP message exchange might have. They are therefore interesting targets for breaking confidentiality or integrity of CoAP message exchanges. As noted in [RFC2616], they are also interesting targets for breaking availability.

The threat to confidentiality and integrity of request/response data is amplified where proxies also cache. Note that CoAP does not define any of the cache-suppressing Cache-Control options that HTTP/1.1 provides to better protect sensitive data.

For a caching implementation, any access control considerations that would apply to making the request that generated the cache entry also need to be applied to the value in the cache. This is relevant for clients that implement multiple security domains, as well as for proxies that may serve multiple clients. Also, a caching proxy MUST NOT make cached values available to requests that have lesser transport security properties than to which it would make available the process of forwarding the request in the first place.

Finally, a proxy that fans out Separate Responses (as opposed to Piggy-backed Responses) to multiple original requesters may provide additional amplification (see below).

11.3. Risk of amplification

CoAP servers generally reply to a request packet with a response packet. This response packet may be significantly larger than the request packet. An attacker might use CoAP nodes to turn a small attack packet into a larger attack packet, an approach known as amplification. There is therefore a danger that CoAP nodes could become implicated in denial of service (DoS) attacks by using the amplifying properties of the protocol: An attacker that is attempting to overload a victim but is limited in the amount of traffic it can generate, can use amplification to generate a larger amount of traffic.

This is particularly a problem in nodes that enable NoSec access, that are accessible from an attacker and can access potential victims (e.g. on the general Internet), as the UDP protocol provides no way to verify the source address given in the request packet. An attacker need only place the IP address of the victim in the source address of a suitable request packet to generate a larger packet directed at the victim.

As a mitigating factor, many constrained networks will only be able to generate a small amount of traffic, which may make CoAP nodes less attractive for this attack. However, the limited capacity of the constrained network makes the network itself a likely victim of an amplification attack.

A CoAP server can reduce the amount of amplification it provides to an attacker by using slicing/blocking modes of CoAP [I-D.ietf-core-block] and offering large resource representations only in relatively small slices. E.g., for a 1000 byte resource, a 10-byte request might result in an 80-byte response (with a 64-byte block) instead of a 1016-byte response, considerably reducing the amplification provided.

CoAP also supports the use of multicast IP addresses in requests, an important requirement for M2M. Multicast CoAP requests may be the source of accidental or deliberate denial of service attacks, especially over constrained networks. This specification attempts to reduce the amplification effects of multicast requests by limiting when a response is returned. To limit the possibility of malicious use, CoAP servers SHOULD NOT accept multicast requests that can not be authenticated. If possible a CoAP server SHOULD limit the support for multicast requests to specific resources where the feature is required.

On some general purpose operating systems providing a Posix-style API, it is not straightforward to find out whether a packet received was addressed to a multicast address. While many implementations will know whether they have joined a multicast group, this creates a problem for packets addressed to multicast addresses of the form FF0x::1, which are received by every IPv6 node. Implementations SHOULD make use of modern APIs such as IPV6_RECVPKTINFO [RFC3542], if available, to make this determination.

11.4. IP Address Spoofing Attacks

Due to the lack of a handshake in UDP, a rogue endpoint which is free to read and write messages carried by the constrained network (i.e. NoSec or PreSharedKey deployments with nodes/key ratio > 1:1), may easily attack a single endpoint, a group of endpoints, as well as the whole network e.g. by:

1. spoofing RST in response to a CON or NON message, thus making an endpoint "deaf"; or

2. spoofing the entire response with forged payload/options (this has different levels of impact: from single response disruption, to much bolder attacks on the supporting infrastructure, e.g. poisoning proxy caches, or tricking validation / lookup interfaces in resource directories and, more generally, any component that stores global network state and uses CoAP as the messaging facility to handle state set/update's is a potential target.); or

3. spoofing a multicast request for a target node which may result in both network congestion/collapse and victim DoS'ing / forced wakeup from sleeping; or

4. spoofing observe messages, etc.

In principle, spoofing can be detected by CoAP only in case CON semantics is used, because of unexpected ACK/RSTs coming from the deceived endpoint. But this imposes keeping track of the used Message IDs which is not always possible, and moreover detection becomes available usually after the damage is already done. This kind of attack can be prevented using security modes other than NoSec.

11.5. Cross-Protocol Attacks

The ability to incite a CoAP endpoint to send packets to a fake source address can be used not only for amplification, but also for cross-protocol attacks against a victim listening to UDP packets at a given address (IP address and port):

- the attacker sends a message to a CoAP endpoint with the given address as the fake source address,

- the CoAP endpoint replies with a message to the given source address,

- the victim at the given address receives a UDP packet that it interprets according to the rules of a different protocol.

This may be used to circumvent firewall rules that prevent direct communication from the attacker to the victim, but happen to allow communication from the CoAP endpoint (which may also host a valid role in the other protocol) to the victim.

Also, CoAP endpoints may be the victim of a cross-protocol attack generated through an endpoint of another UDP-based protocol such as DNS. In both cases, attacks are possible if the security properties of the endpoints rely on checking IP addresses (and firewalling off direct attacks sent from outside using fake IP addresses). In general, because of their lack of context, UDP-based protocols are relatively easy targets for cross-protocol attacks.

Finally, CoAP URIs transported by other means could be used to incite clients to send messages to endpoints of other protocols.

One mitigation against cross-protocol attacks is strict checking of the syntax of packets received, combined with sufficient difference in syntax. As an example, it might help if it were difficult to incite a DNS server to send a DNS response that would pass the checks of a CoAP endpoint. Unfortunately, the first two bytes of a DNS reply are an ID that can be chosen by the attacker, which map into the interesting part of the CoAP header, and the next two bytes are then interpreted as CoAP's Message ID (i.e., any value is acceptable). The DNS count words may be interpreted as multiple instances of a (non-existent, but elective) CoAP option 0. The echoed query finally may be manufactured by the attacker to achieve a desired effect on the CoAP endpoint; the response added by the server (if any) might then just be interpreted as added payload.

```
                                 1  1  1  1  1  1
   0  1  2  3  4  5  6  7  8  9  0  1  2  3  4  5
 +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
 |                      ID                       | T, OC, code
 +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
 |QR|   Opcode  |AA|TC|RD|RA|   Z    |   RCODE   | message id
 +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
 |                    QDCOUNT                    | (options 0)
 +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
 |                    ANCOUNT                    | (options 0)
 +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
 |                    NSCOUNT                    | (options 0)
 +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
 |                    ARCOUNT                    | (options 0)
 +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

Figure 15: DNS Header vs. CoAP Message

In general, for any pair of protocols, one of the protocols can very well have been designed in a way that enables an attacker to cause the generation of replies that look like messages of the other protocol. It is often much harder to ensure or prove the absence of viable attacks than to generate examples that may not yet completely enable an attack but might be further developed by more creative minds. Cross-protocol attacks can therefore only be completely mitigated if endpoints don’t authorize actions desired by an attacker just based on trusting the source IP address of a packet. Conversely, a NoSec environment that completely relies on a firewall for CoAP security not only needs to firewall off the CoAP endpoints but also all other endpoints that might be incited to send UDP messages to CoAP endpoints using some other UDP-based protocol.

In addition to the considerations above, the security considerations for DTLS with respect to cross-protocol attacks apply. E.g., if the same DTLS security association ("connection") is used to carry data of multiple protocols, DTLS no longer provides protection against cross-protocol attacks between these protocols.

12. IANA Considerations

12.1. CoAP Code Registry

This document defines a registry for the values of the Code field in the CoAP header. The name of the registry is "CoAP Codes".

All values are assigned by sub-registries according to the following ranges:

0 Indicates an empty message (see Section 4.1).

1-31 Indicates a request. Values in this range are assigned by the "CoAP Method Codes" sub-registry (see Section 12.1.1).

32-63 Reserved

64-191 Indicates a response. Values in this range are assigned by the "CoAP Response Codes" sub-registry (see Section 12.1.2).

192-255 Reserved

12.1.1. Method Codes

The name of the sub-registry is "CoAP Method Codes".

Each entry in the sub-registry must include the Method Code in the range 1-31, the name of the method, and a reference to the method’s documentation.

Initial entries in this sub-registry are as follows:

Code	Name	Reference
1	GET	[RFCXXXX]
2	POST	[RFCXXXX]
3	PUT	[RFCXXXX]
4	DELETE	[RFCXXXX]

Table 2: CoAP Method Codes

All other Method Codes are Unassigned.

The IANA policy for future additions to this registry is "IETF Review" as described in [RFC5226].

The documentation of a method code should specify the semantics of a request with that code, including the following properties:

- o The response codes the method returns in the success case.
- o Whether the method is idempotent, safe, or both.

12.1.2. Response Codes

The name of the sub-registry is "CoAP Response Codes".

Each entry in the sub-registry must include the Response Code in the range 64-191, a description of the Response Code, and a reference to the Response Code's documentation.

Initial entries in this sub-registry are as follows:

Code	Description	Reference
65	2.01 Created	[RFCXXXX]
66	2.02 Deleted	[RFCXXXX]
67	2.03 Valid	[RFCXXXX]
68	2.04 Changed	[RFCXXXX]
69	2.05 Content	[RFCXXXX]
128	4.00 Bad Request	[RFCXXXX]
129	4.01 Unauthorized	[RFCXXXX]
130	4.02 Bad Option	[RFCXXXX]
131	4.03 Forbidden	[RFCXXXX]
132	4.04 Not Found	[RFCXXXX]
133	4.05 Method Not Allowed	[RFCXXXX]
134	4.06 Not Acceptable	[RFCXXXX]
140	4.12 Precondition Failed	[RFCXXXX]
141	4.13 Request Entity Too Large	[RFCXXXX]
143	4.15 Unsupported Content-Format	[RFCXXXX]
160	5.00 Internal Server Error	[RFCXXXX]
161	5.01 Not Implemented	[RFCXXXX]
162	5.02 Bad Gateway	[RFCXXXX]
163	5.03 Service Unavailable	[RFCXXXX]
164	5.04 Gateway Timeout	[RFCXXXX]
165	5.05 Proxying Not Supported	[RFCXXXX]

Table 3: CoAP Response Codes

The Response Codes 96-127 are Reserved for future use. All other Response Codes are Unassigned.

The IANA policy for future additions to this registry is "IETF Review" as described in [RFC5226].

The documentation of a response code should specify the semantics of a response with that code, including the following properties:

- The methods the response code applies to.

- Whether payload is required, optional or not allowed.

- The semantics of the payload. For example, the payload of a 2.05 (Content) response is a representation of the target resource; the payload in an error response is a human-readable diagnostic message.

- The format of the payload. For example, the format in a 2.05 (Content) response is indicated by the Content-Format Option; the format of the payload in an error response is always Net-Unicode text.

- Whether the response is cacheable according to the freshness model.

- Whether the response is validatable according to the validation model.

- Whether the response causes a cache to mark responses stored for the request URI as not fresh.

12.2. Option Number Registry

This document defines a registry for the Option Numbers used in CoAP options. The name of the registry is "CoAP Option Numbers".

Each entry in the registry must include the Option Number, the name of the option and a reference to the option’s documentation.

Initial entries in this registry are as follows:

Number	Name	Reference
0	(Reserved)	
1	If-Match	[RFCXXXX]
3	Uri-Host	[RFCXXXX]
4	ETag	[RFCXXXX]
5	If-None-Match	[RFCXXXX]
7	Uri-Port	[RFCXXXX]
8	Location-Path	[RFCXXXX]
11	Uri-Path	[RFCXXXX]
12	Content-Format	[RFCXXXX]
14	Max-Age	[RFCXXXX]
15	Uri-Query	[RFCXXXX]
16	Accept	[RFCXXXX]
19	Token	[RFCXXXX]
20	Location-Query	[RFCXXXX]
35	Proxy-Uri	[RFCXXXX]
128	(Reserved)	[RFCXXXX]
132	(Reserved)	[RFCXXXX]
136	(Reserved)	[RFCXXXX]

Table 4: CoAP Option Numbers

The IANA policy for future additions to this registry is split into three tiers as follows. The range of 0..255 is reserved for options defined by the IETF (IETF Review). The range of 256..2047 is reserved for commonly used options with public specifications (Specification Required). The range of 2048..65535 is for all other options including private or vendor specific ones, which undergo a Designated Expert review to help ensure that the option semantics are defined correctly.

Option Number	Policy [RFC5226]
0..255	IETF Review
256..2047	Specification Required
2048..65535	Designated Expert

The documentation of an Option Number should specify the semantics of an option with that number, including the following properties:

o The meaning of the option in a request.

- The meaning of the option in a response.
- Whether the option is critical or elective, as determined by the Option Number.
- Whether the option is Safe, and whether it is part of the Cache-Key, as determined by the Option Number (see Section 5.4.2).
- The format and length of the option's value.
- Whether the option must occur at most once or whether it can occur multiple times.
- The default value, if any. For a critical option with a default value, a discussion on how the default value enables processing by implementations not implementing the critical option (Section 5.4.4).

12.3. Content-Format Registry

Internet media types are identified by a string, such as "application/xml" [RFC2046]. In order to minimize the overhead of using these media types to indicate the format of payloads, this document defines a registry for a subset of Internet media types to be used in CoAP and assigns each, in combination with a content-coding, a numeric identifier. The name of the registry is "CoAP Content-Formats".

Each entry in the registry must include the media type registered with IANA, the numeric identifier in the range 0-65535 to be used for that media type in CoAP, the content-coding associated with this identifier, and a reference to a document describing what a payload with that media type means semantically.

CoAP does not include a way to convey content-encoding information with a request or response, and for that reason the content-encoding is also specified for each identifier (if any). If multiple content-encodings will be used with a media type, then a separate Content-Format identifier for each is to be registered. Similarly, other parameters related to an Internet media type, such as level, can be defined for a CoAP Content-Format entry.

Initial entries in this registry are as follows:

Media type	Encoding	Id.	Reference
text/plain; charset=utf-8	-	0	[RFC2046][RFC3676][RFC5147]
application/ link-format	-	40	[RFC6690]
application/xml	-	41	[RFC3023]
application/ octet-stream	-	42	[RFC2045][RFC2046]
application/exi	-	47	[EXIMIME]
application/json	-	50	[RFC4627]

Table 5: CoAP Content-Formats

The identifiers between 201 and 255 inclusive are reserved for Private Use. All other identifiers are Unassigned.

Because the name space of single-byte identifiers is so small, the IANA policy for future additions in the range 0-200 inclusive to the registry is "Expert Review" as described in [RFC5226]. The IANA policy for additions in the range 256-65535 inclusive is "First Come First Served" as described in [RFC5226].

In machine to machine applications, it is not expected that generic Internet media types such as text/plain, application/xml or application/octet-stream are useful for real applications in the long term. It is recommended that M2M applications making use of CoAP will request new Internet media types from IANA indicating semantic information about how to create or parse a payload. For example, a Smart Energy application payload carried as XML might request a more specific type like application/se+xml or application/se-exi.

12.4. URI Scheme Registration

This document requests the registration of the Uniform Resource Identifier (URI) scheme "coap". The registration request complies with [RFC4395].

URI scheme name.
   coap

Status.
   Permanent.

URI scheme syntax.
Defined in Section 6.1 of [RFCXXXX].

URI scheme semantics.
The "coap" URI scheme provides a way to identify resources that are potentially accessible over the Constrained Application Protocol (CoAP). The resources can be located by contacting the governing CoAP server and operated on by sending CoAP requests to the server. This scheme can thus be compared to the "http" URI scheme [RFC2616]. See Section 6 of [RFCXXXX] for the details of operation.

Encoding considerations.
The scheme encoding conforms to the encoding rules established for URIs in [RFC3986], i.e. internationalized and reserved characters are expressed using UTF-8-based percent-encoding.

Applications/protocols that use this URI scheme name.
The scheme is used by CoAP endpoints to access CoAP resources.

Interoperability considerations.
None.

Security considerations.
See Section 11.1 of [RFCXXXX].

Contact.
IETF Chair <chair@ietf.org>

Author/Change controller.
IESG <iesg@ietf.org>

References.
[RFCXXXX]

12.5. Secure URI Scheme Registration

This document requests the registration of the Uniform Resource Identifier (URI) scheme "coaps". The registration request complies with [RFC4395].

URI scheme name.
coaps

Status.
Permanent.

URI scheme syntax.
Defined in Section 6.2 of [RFCXXXX].

URI scheme semantics.
The "coaps" URI scheme provides a way to identify resources that are potentially accessible over the Constrained Application Protocol (CoAP) using Datagram Transport Layer Security (DTLS) for transport security. The resources can be located by contacting the governing CoAP server and operated on by sending CoAP requests to the server. This scheme can thus be compared to the "https" URI scheme [RFC2616]. See Section 6 of [RFCXXXX] for the details of operation.

Encoding considerations.
The scheme encoding conforms to the encoding rules established for URIs in [RFC3986], i.e. internationalized and reserved characters are expressed using UTF-8-based percent-encoding.

Applications/protocols that use this URI scheme name.
The scheme is used by CoAP endpoints to access CoAP resources using DTLS.

Interoperability considerations.
None.

Security considerations.
See Section 11.1 of [RFCXXXX].

Contact.
IETF Chair <chair@ietf.org>

Author/Change controller.
IESG <iesg@ietf.org>

References.
[RFCXXXX]

12.6. Service Name and Port Number Registration

One of the functions of CoAP is resource discovery: a CoAP client can ask a CoAP server about the resources offered by it (see Section 7). To enable resource discovery just based on the knowledge of an IP address, the CoAP port for resource discovery needs to be standardized.

IANA has assigned the port number 5683 and the service name "coap", in accordance with [RFC6335].

Besides unicast, CoAP can be used with both multicast and anycast.

Service Name.
coap

Transport Protocol.
UDP

Assignee.
IESG <iesg@ietf.org>

Contact.
IETF Chair <chair@ietf.org>

Description.
Constrained Application Protocol (CoAP)

Reference.
[RFCXXXX]

Port Number.
5683

12.7. Secure Service Name and Port Number Registration

CoAP resource discovery may also be provided using the DTLS-secured CoAP "coaps" scheme. Thus the CoAP port for secure resource discovery needs to be standardized.

This document requests the assignment of the port number [IANA_TBD_PORT] and the service name "coaps", in accordance with [RFC6335].

Besides unicast, DTLS-secured CoAP can be used with anycast.

Service Name.
coaps

Transport Protocol.
UDP

Assignee.
IESG <iesg@ietf.org>

Contact.
IETF Chair <chair@ietf.org>

Description.
DTLS-secured CoAP

Reference.
[RFCXXXX]

Port Number.
[IANA_TBD_PORT]

12.8. Multicast Address Registration

Section 8, "Multicast CoAP", defines the use of multicast. This document requests the assignment of the following multicast addresses for use by CoAP nodes:

IPv4 -- "All CoAP Nodes" address [TBD1], from the IPv4 Multicast Address Space Registry. As the address is used for discovery that may span beyond a single network, it should come from the Internetwork Control Block (224.0.1.x, RFC 5771).

IPv6 -- "All CoAP Nodes" address [TBD2], from the IPv6 Multicast Address Space Registry, in the Variable Scope Multicast Addresses space (RFC3307). Note that there is a distinct multicast address for each scope that interested CoAP nodes should listen to.

[The explanatory text to be removed upon allocation of the addresses, except for the note about the distinct multicast addresses.]

13. Acknowledgements

Special thanks to Peter Bigot, Esko Dijk and Cullen Jennings for substantial contributions to the ideas and text in the document, along with countless detailed reviews and discussions.

Thanks to Ed Beroset, Angelo P. Castellani, Gilbert Clark, Robert Cragie, Esko Dijk, Lisa Dussealt, Thomas Fossati, Tom Herbst, Richard Kelsey, Ari Keranen, Matthias Kovatsch, Salvatore Loreto, Kerry Lynn, Alexey Melnikov, Guido Moritz, Petri Mutka, Colin O’Flynn, Charles Palmer, Adriano Pezzuto, Robert Quattlebaum, Akbar Rahman, Eric Rescorla, David Ryan, Szymon Sasin, Michael Scharf, Dale Seed, Robby Simpson, Peter van der Stok, Michael Stuber, Linyi Tian, Gilman Tolle, Matthieu Vial and Alper Yegin for helpful comments and discussions that have shaped the document.

Some of the text has been borrowed from the working documents of the IETF httpbis working group.

14. References

14.1. Normative References

[I-D.farrell-decade-ni]
Farrell, S., Kutscher, D., Dannewitz, C., Ohlman, B., Keranen, A., and P. Hallam-Baker, "Naming Things with Hashes", draft-farrell-decade-ni-10 (work in progress), August 2012.

[I-D.ietf-tls-oob-pubkey]
Wouters, P., Gilmore, J., Weiler, S., Kivinen, T., and H. Tschofenig, "Out-of-Band Public Key Validation for Transport Layer Security", draft-ietf-tls-oob-pubkey-04 (work in progress), July 2012.

[RFC2045] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, November 1996.

[RFC2046] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types", RFC 2046, November 1996.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

[RFC3023] Murata, M., St. Laurent, S., and D. Kohn, "XML Media Types", RFC 3023, January 2001.

[RFC3602] Frankel, S., Glenn, R., and S. Kelly, "The AES-CBC Cipher Algorithm and Its Use with IPsec", RFC 3602, September 2003.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, November 2003.

[RFC3676] Gellens, R., "The Text/Plain Format and DelSp Parameters", RFC 3676, February 2004.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, January 2005.

[RFC4279] Eronen, P. and H. Tschofenig, "Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)", RFC 4279, December 2005.

[RFC4303] Kent, S., "IP Encapsulating Security Payload (ESP)", RFC 4303, December 2005.

[RFC4309] Housley, R., "Using Advanced Encryption Standard (AES) CCM Mode with IPsec Encapsulating Security Payload (ESP)", RFC 4309, December 2005.

[RFC4395] Hansen, T., Hardie, T., and L. Masinter, "Guidelines and Registration Procedures for New URI Schemes", BCP 35, RFC 4395, February 2006.

[RFC4835] Manral, V., "Cryptographic Algorithm Implementation Requirements for Encapsulating Security Payload (ESP) and Authentication Header (AH)", RFC 4835, April 2007.

[RFC5147] Wilde, E. and M. Duerst, "URI Fragment Identifiers for the text/plain Media Type", RFC 5147, April 2008.

[RFC5198] Klensin, J. and M. Padlipsky, "Unicode Format for Network Interchange", RFC 5198, March 2008.

[RFC5226] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 5226, May 2008.

[RFC5234] Crocker, D. and P. Overell, "Augmented BNF for Syntax Specifications: ABNF", STD 68, RFC 5234, January 2008.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, May 2008.

[RFC5785] Nottingham, M. and E. Hammer-Lahav, "Defining Well-Known Uniform Resource Identifiers (URIs)", RFC 5785, April 2010.

[RFC5952] Kawamura, S. and M. Kawashima, "A Recommendation for IPv6 Address Text Representation", RFC 5952, August 2010.

[RFC5988] Nottingham, M., "Web Linking", RFC 5988, October 2010.

[RFC5996] Kaufman, C., Hoffman, P., Nir, Y., and P. Eronen, "Internet Key Exchange Protocol Version 2 (IKEv2)", RFC 5996, September 2010.

[RFC6066] Eastlake, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", RFC 6066, January 2011.

[RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, January 2012.

[RFC6690] Shelby, Z., "Constrained RESTful Environments (CoRE) Link Format", RFC 6690, August 2012.

14.2. Informative References

[EUI64] "GUIDELINES FOR 64-BIT GLOBAL IDENTIFIER (EUI-64) REGISTRATION AUTHORITY", April 2010, <http://standards.ieee.org/regauth/oui/tutorials/EUI64.html>.

[EXIMIME] "Efficient XML Interchange (EXI) Format 1.0", December 2009, <http://www.w3.org/TR/2009/CR-exi-20091208/#mediaTypeRegistration>.

[I-D.allman-tcpm-rto-consider] Allman, M., "Retransmission Timeout Considerations", draft-allman-tcpm-rto-consider-01 (work in progress), May 2012.

[I-D.ietf-core-block] Bormann, C. and Z. Shelby, "Blockwise transfers in CoAP", draft-ietf-core-block-09 (work in progress), August 2012.

[I-D.ietf-core-observe] Hartke, K., "Observing Resources in CoAP", draft-ietf-core-observe-06 (work in progress), September 2012.

[I-D.kivinen-ipsecme-ikev2-minimal] Kivinen, T., "Minimal IKEv2", draft-kivinen-ipsecme-ikev2-minimal-00 (work in progress), February 2011.

[I-D.mcgrew-tls-aes-ccm-ecc] McGrew, D., Bailey, D., Campagna, M., and R. Dugal, "AES-CCM ECC Cipher Suites for TLS", draft-mcgrew-tls-aes-ccm-ecc-05 (work in progress), July 2012.

[REST]   Fielding, R., "Architectural Styles and the Design of Network-based Software Architectures", Ph.D. Dissertation, University of California, Irvine, 2000, <http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf>.

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

[RFC2818] Rescorla, E., "HTTP Over TLS", RFC 2818, May 2000.

[RFC3264] Rosenberg, J. and H. Schulzrinne, "An Offer/Answer Model with Session Description Protocol (SDP)", RFC 3264, June 2002.

[RFC3542] Stevens, W., Thomas, M., Nordmark, E., and T. Jinmei, "Advanced Sockets Application Program Interface (API) for IPv6", RFC 3542, May 2003.

[RFC4492] Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., and B. Moeller, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)", RFC 4492, May 2006.

[RFC4627] Crockford, D., "The application/json Media Type for JavaScript Object Notation (JSON)", RFC 4627, July 2006.

[RFC4944] Montenegro, G., Kushalnagar, N., Hui, J., and D. Culler, "Transmission of IPv6 Packets over IEEE 802.15.4 Networks", RFC 4944, September 2007.

[RFC5405] Eggert, L. and G. Fairhurst, "Unicast UDP Usage Guidelines for Application Designers", BCP 145, RFC 5405, November 2008.

[RFC6120] Saint-Andre, P., "Extensible Messaging and Presence Protocol (XMPP): Core", RFC 6120, March 2011.

[RFC6335] Cotton, M., Eggert, L., Touch, J., Westerlund, M., and S. Cheshire, "Internet Assigned Numbers Authority (IANA) Procedures for the Management of the Service Name and Transport Protocol Port Number Registry", BCP 165, RFC 6335, August 2011.

[RFC6655] McGrew, D. and D. Bailey, "AES-CCM Cipher Suites for Transport Layer Security (TLS)", RFC 6655, July 2012.

Appendix A. Examples

This section gives a number of short examples with message flows for GET requests. These examples demonstrate the basic operation, the operation in the presence of retransmissions, and multicast.

Figure 16 shows a basic GET request causing a piggy-backed response: The client sends a Confirmable GET request for the resource coap://server/temperature to the server with a Message ID of 0x7d34. The request includes one Uri-Path Option (Delta 0 + 11 = 11, Length 11, Value "temperature"); the Token is left at its default value (empty). This request is a total of 16 bytes long. A 2.05 (Content) response is returned in the Acknowledgement message that acknowledges the Confirmable request, echoing both the Message ID 0x7d34 and the (implicitly empty) Token value. The response includes a Payload of "22.3 C" and is 10 bytes long.

```
Client  Server
   |      |
   |      |
   +----->|     Header: GET (T=CON, Code=1, MID=0x7d34)
   | GET  |   Uri-Path: "temperature"
   |      |
   |      |
   |<-----+     Header: 2.05 Content (T=ACK, Code=69, MID=0x7d34)
   | 2.05 |    Payload: "22.3 C"
   |      |


 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 1 | 0 |   1   |     GET=1     |          MID=0x7d34           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  11   |  11   |      "temperature" (11 B) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 1 | 2 |   0   |    2.05=69    |          MID=0x7d34           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|      "22.3 C" (6 B) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 16: Confirmable request; piggy-backed response

Figure 17 shows a similar example, but with the inclusion of an explicit Token Option (Delta 11 + 8 = 19, Length 1, Value 0x20) in the request and (Jump 15 + 4 = 19) in the response, increasing the sizes to 18 and 12 bytes, respectively.

```
Client  Server
   |      |
   |      |
   +----->|     Header: GET (T=CON, Code=1, MID=0x7d35)
   | GET  |      Token: 0x20
   |      |   Uri-Path: "temperature"
   |      |
   |      |
   |<-----+     Header: 2.05 Content (T=ACK, Code=69, MID=0x7d35)
   | 2.05 |      Token: 0x20
   |      |    Payload: "22.3 C"
   |      |

 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 1 | 0 |   2   |     GET=1     |          MID=0x7d35           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   11  |  11   |      "temperature" (11 B) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   8   |   1   |     0x20      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| 1 | 2 |   1   |    2.05=69    |          MID=0x7d35           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Jump 15 = 0xf1 |  4    |   1   |     0x20      | "22.3 C" (6 B) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 17: Confirmable request; piggy-backed response

In Figure 18, the Confirmable GET request is lost. After ACK_TIMEOUT seconds, the client retransmits the request, resulting in a piggy-backed response as in the previous example.

```
Client  Server
   |       |
   |       |
   +----X |     Header: GET (T=CON, Code=1, MID=0x7d36)
   | GET  |      Token: 0x31
   |       |   Uri-Path: "temperature"
TIMEOUT    |
   |       |
   +----->|     Header: GET (T=CON, Code=1, MID=0x7d36)
   | GET  |      Token: 0x31
   |       |   Uri-Path: "temperature"
   |       |
   |       |
   |<-----+     Header: 2.05 Content (T=ACK, Code=69, MID=0x7d36)
   | 2.05 |      Token: 0x31
   |       |    Payload: "22.3 C"
   |       |
```

Figure 18: Confirmable request (retransmitted); piggy-backed response

In Figure 19, the first Acknowledgement message from the server to the client is lost. After ACK_TIMEOUT seconds, the client retransmits the request.

```
Client  Server
   |       |
   |       |
   +----->|     Header: GET (T=CON, Code=1, MID=0x7d37)
   | GET  |      Token: 0x42
   |       |   Uri-Path: "temperature"
   |       |
   |       |
   | X----+     Header: 2.05 Content (T=ACK, Code=69, MID=0x7d37)
   | 2.05 |      Token: 0x42
   |       |    Payload: "22.3 C"
TIMEOUT    |
   |       |
   +----->|     Header: GET (T=CON, Code=1, MID=0x7d37)
   | GET  |      Token: 0x42
   |       |   Uri-Path: "temperature"
   |       |
   |       |
   |<-----+     Header: 2.05 Content (T=ACK, Code=69, MID=0x7d37)
   | 2.05 |      Token: 0x42
   |       |    Payload: "22.3 C"
   |       |
```

Figure 19: Confirmable request; piggy-backed response (retransmitted)

In Figure 20, the server acknowledges the Confirmable request and sends a 2.05 (Content) response separately in a Confirmable message. Note that the Acknowledgement message and the Confirmable response do not necessarily arrive in the same order as they were sent. The client acknowledges the Confirmable response.

```
Client  Server
   |      |
   |      |
   +----->|     Header: GET (T=CON, Code=1, MID=0x7d38)
   | GET  |      Token: 0x53
   |      |   Uri-Path: "temperature"
   |      |
   |      |
   |<- - -+     Header: (T=ACK, Code=0, MID=0x7d38)
   |      |
   |      |
   |<-----+     Header: 2.05 Content (T=CON, Code=69, MID=0xad7b)
   | 2.05 |      Token: 0x53
   |      |    Payload: "22.3 C"
   |      |
   |      |
   +- - ->|     Header: (T=ACK, Code=0, MID=0xad7b)
   |      |
```

Figure 20: Confirmable request; separate response

Figure 21 shows an example where the client loses its state (e.g., crashes and is rebooted) right after sending a Confirmable request, so the separate response arriving some time later comes unexpected. In this case, the client rejects the Confirmable response with a Reset message. Note that the unexpected ACK is silently ignored.

```
Client  Server
   |       |
   |       |
   +------>|     Header: GET (T=CON, Code=1, MID=0x7d39)
   | GET   |      Token: 0x64
   |       |   Uri-Path: "temperature"
 CRASH     |
   |       |
   |<- - --+     Header: (T=ACK, Code=0, MID=0x7d39)
   |       |
   |       |
   |<------+     Header: 2.05 Content (T=CON, Code=69, MID=0xad7c)
   | 2.05  |      Token: 0x64
   |       |    Payload: "22.3 C"
   |       |
   |       |
   +- - -->|     Header: (T=RST, Code=0, MID=0xad7c)
   |       |
```

Figure 21: Confirmable request; separate response (unexpected)

Figure 22 shows a basic GET request where the request and the response are non-confirmable, so both may be lost without notice.

```
Client  Server
   |       |
   |       |
   +------>|     Header: GET (T=NON, Code=1, MID=0x7d40)
   | GET   |      Token: 0x75
   |       |   Uri-Path: "temperature"
   |       |
   |       |
   |<------+     Header: 2.05 Content (T=NON, Code=69, MID=0xad7d)
   | 2.05  |      Token: 0x75
   |       |    Payload: "22.3 C"
   |       |
```

Figure 22: Non-confirmable request; Non-confirmable response

In Figure 23, the client sends a Non-confirmable GET request to a multicast address: all nodes in link-local scope. There are 3 servers on the link: A, B and C. Servers A and B have a matching resource, therefore they send back a Non-confirmable 2.05 (Content) response. The response sent by B is lost. C does not have matching response, therefore it sends a Non-confirmable 4.04 (Not Found) response.

```
   Client  ff02::1  A  B  C
      |       |     |  |  |
      |       |     |  |  |
      +------>|     |  |  |   Header: GET (T=NON, Code=1, MID=0x7d41)
      |  GET  |     |  |  |    Token: 0x86
      |             |  |  |    Uri-Path: "temperature"
      |             |  |  |
      |             |  |  |
      |<------------+  |  |   Header: 2.05 (T=NON, Code=69, MID=0x60b1)
      |     2.05    |  |  |    Token: 0x86
      |             |  |  |    Payload: "22.3 C"
      |             |  |  |
      |             |  |  |
      |   X------------+  |   Header: 2.05 (T=NON, Code=69, MID=0x01a0)
      |     2.05    |  |  |    Token: 0x86
      |             |  |  |    Payload: "20.9 C"
      |             |  |  |
      |             |  |  |
      |<------------------+   Header: 4.04 (T=NON, Code=132, MID=0x952a)
      |     4.04    |  |  |    Token: 0x86
      |             |  |  |
```

Figure 23: Non-confirmable request (multicast); Non-confirmable response

Appendix B. URI Examples

The following examples demonstrate different sets of Uri options, and the result after constructing an URI from them.

- coap://[2001:db8::2:1]/

  Destination IP Address = [2001:db8::2:1]

  Destination UDP Port = 5683

- coap://example.net/

  Destination IP Address = [2001:db8::2:1]

  Destination UDP Port = 5683

  Uri-Host = "example.net"

- coap://example.net/.well-known/core

Destination IP Address = [2001:db8::2:1]

Destination UDP Port = 5683

Uri-Host = "example.net"

Uri-Path = ".well-known"

Uri-Path = "core"

- coap://
  xn--18j4d.example/%E3%81%93%E3%82%93%E3%81%AB%E3%81%A1%E3%81%AF

  Destination IP Address = [2001:db8::2:1]

  Destination UDP Port = 5683

  Uri-Host = "xn--18j4d.example"

  Uri-Path = the string composed of the Unicode characters U+3053 U+3093 U+306b U+3061 U+306f, usually represented in UTF-8 as E38193E38293E381ABE381A1E381AF hexadecimal

- coap://198.51.100.1:61616//%2F//?%2F%2F&?%26

  Destination IP Address = 198.51.100.1

  Destination UDP Port = 61616

  Uri-Path = ""

  Uri-Path = "/"

  Uri-Path = ""

  Uri-Path = ""

  Uri-Query = "//"

  Uri-Query = "?&"

Appendix C. Changelog

Changed from ietf-11 to ietf-12:

- Extended options to support lengths of up to 1034 bytes (#202).

- Added new Jump mechanism for options and removed Fenceposting (#214).
- Added new IANA option number registration policy (#214).
- Added Proxy Unsafe/Safe and Cache-Key masking to option numbers (#241).
- Re-numbered option numbers to use Unsafe/Safe and Cache-Key compliant numbers (#241).
- Defined NSTART and restricted the value to 1 with a MUST (#215).
- Defined PROBING_RATE and set it to 1 Byte/second (#215).
- Defined DEFAULT_LEISURE (#246).
- Renamed Content-Type into Content-Format, and Media Type registry into Content-Format registry.
- A large number of small editorial changes, clarifications and improvements have been made.

Changed from ietf-10 to ietf-11:

- Expanded section 4.8 on Transmission Parameters, and used the derived values defined there (#201). Changed parameter names to be shorter and more to the point.
- Several more small editorial changes, clarifications and improvements have been made.

Changed from ietf-09 to ietf-10:

- Option deltas are restricted to 0 to 14; the option delta 15 is used exclusively for the end-of-options marker (#239).
- Option numbers that are a multiple of 14 are not reserved, but are required to have an empty default value (#212).
- Fixed misleading language that was introduced in 5.10.2 in coap-07 re Uri-Host and Uri-Port (#208).
- Segments and arguments can have a length of zero characters (#213).
- The Location-* options describe together describe one location. The location is a relative URI, not an "absolute path URI" (#218).

- The value of the Location-Path Option must not be '.' or '..' (#218).

- Added a sentence on constructing URIs from Location-* options (#231).

- Reserved option numbers for future Location-* options (#230).

- Fixed response codes with payload inconsistency (#233).

- Added advice on default values for critical options (#207).

- Clarified use of identifiers in RawPublicKey Mode Provisioning (#222).

- Moved "Securing CoAP" out of the "Security Considerations" (#229).

- Added "All CoAP Nodes" multicast addresses to "IANA Considerations" (#216).

- Over 100 small editorial changes, clarifications and improvements have been made.

Changed from ietf-08 to ietf-09:

- Improved consistency of statements about RST on NON: RST is a valid response to a NON message (#183).

- Clarified that the protocol constants can be configured for specific application environments.

- Added implementation note recommending piggy-backing whenever possible (#182).

- Added a content-encoding column to the media type registry (#181).

- Minor improvements to Appendix D.

- Added text about multicast response suppression (#177).

- Included the new End-of-options Marker (#176).

- Added a reference to draft-ietf-tls-oob-pubkey and updated the RPK text accordingly.

Changed from ietf-07 to ietf-08:

- o Clarified matching rules for messages (#175)
- o Fixed a bug in Section 8.2.2 on Etags (#168)
- o Added an IP address spoofing threat analysis contribution (#167)
- o Re-focused the security section on raw public keys (#166)
- o Added an 4.06 error to Accept (#165)

Changed from ietf-06 to ietf-07:

- o application/link-format added to Media types registration (#160)
- o Moved content-type attribute to the document from link-format.
- o Added coaps scheme and DTLS-secured CoAP default port (#154)
- o Allowed 0-length Content-type options (#150)
- o Added congestion control recommendations (#153)
- o Improved text on PUT/POST response payloads (#149)
- o Added an Accept option for content-negotiation (#163)
- o Added If-Match and If-None-Match options (#155)
- o Improved Token Option explanation (#147)
- o Clarified mandatory to implement security (#156)
- o Added first come first server policy for 2-byte Media type codes (#161)
- o Clarify matching rules for messages and tokens (#151)
- o Changed OPTIONS and TRACE to always return 501 in HTTP-CoAP mapping (#164)

Changed from ietf-05 to ietf-06:

- o HTTP mapping section improved with the minimal protocol standard text for CoAP-HTTP and HTTP-CoAP forward proxying (#137).
- o Eradicated percent-encoding by including one Uri-Query Option per &-delimited argument in a query.

- Allowed RST message in reply to a NON message with unexpected token (#135).

- Cache Invalidation only happens upon successful responses (#134).

- 50% jitter added to the initial retransmit timer (#142).

- DTLS cipher suites aligned with ZigBee IP, DTLS clarified as default CoAP security mechanism (#138, #139)

- Added a minimal reference to draft-kivinen-ipsecme-ikev2-minimal (#140).

- Clarified the comparison of UTF-8s (#136).

- Minimized the initial media type registry (#101).

Changed from ietf-04 to ietf-05:

- Renamed Immediate into Piggy-backed and Deferred into Separate -- should finally end the confusion on what this is about.

- GET requests now return a 2.05 (Content) response instead of 2.00 (OK) response (#104).

- Added text to allow 2.02 (Deleted) responses in reply to POST requests (#105).

- Improved message deduplication rules (#106).

- Section added on message size implementation considerations (#103).

- Clarification made on human readable error payloads (#109).

- Definition of CoAP methods improved (#108).

- Max-Age removed from requests (#107).

- Clarified uniqueness of tokens (#112).

- Location-Query Option added (#113).

- ETag length set to 1-8 bytes (#123).

- Clarified relation between elective/critical and option numbers (#110).

- o Defined when to update Version header field (#111).
- o URI scheme registration improved (#102).
- o Added review guidelines for new CoAP codes and numbers.

Changes from ietf-03 to ietf-04:

- o Major document reorganization (#51, #63, #71, #81).
- o Max-age length set to 0-4 bytes (#30).
- o Added variable unsigned integer definition (#31).
- o Clarification made on human readable error payloads (#50).
- o Definition of POST improved (#52).
- o Token length changed to 0-8 bytes (#53).
- o Section added on multiplexing CoAP, DTLS and STUN (#56).
- o Added cross-protocol attack considerations (#61).
- o Used new Immediate/Deferred response definitions (#73).
- o Improved request/response matching rules (#74).
- o Removed unnecessary media types and added recommendations for their use in M2M (#76).
- o Response codes changed to base 32 coding, new Y.XX naming (#77).
- o References updated as per AD review (#79).
- o IANA section completed (#80).
- o Proxy-Uri Option added to disambiguate between proxy and non-proxy requests (#82).
- o Added text on critical options in cached states (#83).
- o HTTP mapping sections improved (#88).
- o Added text on reverse proxies (#72).
- o Some security text on multicast added (#54).

- Trust model text added to introduction (#58, #60).
- AES-CCM vs. AES-CCB text added (#55).
- Text added about device capabilities (#59).
- DTLS section improvements (#87).
- Caching semantics aligned with RFC2616 (#78).
- Uri-Path Option split into multiple path segments.
- MAX_RETRANSMIT changed to 4 to adjust for RESPONSE_TIME = 2.

Changes from ietf-02 to ietf-03:

- Token Option and related use in asynchronous requests added (#25).
- CoAP specific error codes added (#26).
- Erroring out on unknown critical options changed to a MUST (#27).
- Uri-Query Option added.
- Terminology and definitions of URIs improved.
- Security section completed (#22).

Changes from ietf-01 to ietf-02:

- Sending an error on a critical option clarified (#18).
- Clarification on behavior of PUT and idempotent operations (#19).
- Use of Uri-Authority clarified along with server processing rules; Uri-Scheme Option removed (#20, #23).
- Resource discovery section removed to a separate CoRE Link Format draft (#21).
- Initial security section outline added.

Changes from ietf-00 to ietf-01:

- New cleaner transaction message model and header (#5).
- Removed subscription while being designed (#1).

- o Section 2 re-written (#3).
- o Text added about use of short URIs (#4).
- o Improved header option scheme (#5, #14).
- o Date option removed whiled being designed (#6).
- o New text for CoAP default port (#7).
- o Completed proxying section (#8).
- o Completed resource discovery section (#9).
- o Completed HTTP mapping section (#10).
- o Several new examples added (#11).
- o URI split into 3 options (#12).
- o MIME type defined for link-format (#13, #16).
- o New text on maximum message size (#15).
- o Location Option added.

Changes from shelby-01 to ietf-00:

- o Removed the TCP binding section, left open for the future.
- o Fixed a bug in the example.
- o Marked current Sub/Notify as (Experimental) while under WG discussion.
- o Fixed maximum datagram size to 1280 for both IPv4 and IPv6 (for CoAP-CoAP proxying to work).
- o Temporarily removed the Magic Byte header as TCP is no longer included as a binding.
- o Removed the Uri-code Option as different URI encoding schemes are being discussed.
- o Changed the rel= field to desc= for resource discovery.
- o Changed the maximum message size to 1024 bytes to allow for IP/UDP headers.

- o Made the URI slash optimization and method idempotence MUSTs
- o Minor editing and bug fixing.

Changes from shelby-00 to shelby-01:

- o Unified the message header and added a notify message type.
- o Renamed methods with HTTP names and removed the NOTIFY method.
- o Added a number of options field to the header.
- o Combines the Option Type and Length into an 8-bit field.
- o Added the magic byte header.
- o Added new ETag Option.
- o Added new Date Option.
- o Added new Subscription Option.
- o Completed the HTTP Code - CoAP Code mapping table appendix.
- o Completed the Content-type Identifier appendix and tables.
- o Added more simplifications for URI support.
- o Initial subscription and discovery sections.
- o A Flag requirements simplified.

Authors' Addresses

Zach Shelby
Sensinode
Kidekuja 2
Vuokatti 88600
Finland

Phone: +358407796297
Email: zach@sensinode.com

Klaus Hartke
Universitaet Bremen TZI
Postfach 330440
Bremen D-28359
Germany

Phone: +49-421-218-63905
Email: hartke@tzi.org

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

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

Brian Frank
SkyFoundry
Richmond, VA
USA

Phone:
Email: brian@skyfoundry.com