Abstract

This short I-D makes a number of partially interrelated proposals how to solve certain problems in the CoRE WG’s main protocol, the Constrained Application Protocol (CoAP). The current version has been resubmitted to keep information about these proposals available; the proposals are not all fleshed out at this point in time.

Status of this Memo

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

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

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

This Internet-Draft will expire on April 5, 2013.

Copyright Notice

Copyright (c) 2012 IETF Trust and the persons identified as the document authors. All rights reserved.

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

1. Introduction ................................................. 3
2. Observing Resources in CoAP .................................. 4
3. References .................................................. 7
   3.1. Normative References .................................... 7
   3.2. Informative References ................................... 8
Appendix A. The Nursery (Things that still need to ripen a bit) .......... 9
   A.1. Envelope Options ......................................... 9
   A.2. Payload-Length Option .................................... 10
   A.3. URI Authorities with Binary Adresses ....................... 10
   A.4. Length-aware number encoding (o256) ....................... 11
   A.5. SMS encoding ........................................... 13
       A.5.1. ASCII-optimized SMS encoding ........................ 14
   A.6. CONNECT .................................................. 16
       A.6.1. Requesting a Tunnel with CONNECT ..................... 17
       A.6.2. Using a CONNECT Tunnel ............................... 17
       A.6.3. Closing down a CONNECT Tunnel ......................... 18
Appendix B. The Museum (Things we did, but maybe not exactly this way) ........ 19
   B.1. Getting rid of artificial limitations ....................... 19
       B.1.1. Beyond 270 bytes in a single option .................. 19
       B.1.2. Beyond 15 options .................................... 20
       B.1.3. Implementing the option delimiter for 15 or more options ........................................... 23
       B.1.4. Option Length encoding beyond 270 bytes .............. 24
   B.2. Registered Option ......................................... 27
       B.2.1. A Separate Suboption Number Space ..................... 27
       B.2.2. Opening Up the Option Number Space .................... 28
   B.3. Enabling Protocol Evolution ................................ 32
       B.3.1. Potential new option number allocation ................ 33
   B.4. Patience, Leisure, and Pledge ................................ 35
       B.4.1. Patience .............................................. 35
       B.4.2. Leisure .............................................. 36
       B.4.3. Pledge .............................................. 36
       B.4.4. Option Formats ....................................... 37
Appendix C. The Cemetery (Things we won’t do) .............................. 38
   C.1. Example envelope option: solving #230 ....................... 38
   C.2. Example envelope option: proxy-elective options .............. 39
   C.3. Stateful URI compression ................................... 39
Appendix D. Experimental Options ......................................... 41
   D.1. Options indicating absolute time ............................ 41
   D.2. Representing Durations ...................................... 42
   D.3. Rationale ................................................ 43
   D.4. Pseudo-Floating Point ...................................... 44
   D.5. A Duration Type for CoAP ................................... 45
Authors’ Addresses ................................................. 52
1. Introduction

The CoRE WG is tasked with standardizing an Application Protocol for Constrained Networks/Nodes, CoAP [I-D.ietf-core-coap]. This protocol is intended to provide RESTful [REST] services not unlike HTTP [RFC2616], while reducing the complexity of implementation as well as the size of packets exchanged in order to make these services useful in a highly constrained network of themselves highly constrained nodes.

This objective requires restraint in a number of sometimes conflicting ways:

- reducing implementation complexity in order to minimize code size,
- reducing message sizes in order to minimize the number of fragments needed for each message (in turn to maximize the probability of delivery of the message), the amount of transmission power needed and the loading of the limited-bandwidth channel,
- reducing requirements on the environment such as stable storage, good sources of randomness or user interaction capabilities.

This draft attempts to address a number of problems not yet adequately solved in [I-D.ietf-core-coap]. The solutions proposed to these problems are somewhat interrelated and are therefore presented in one draft. As of the current version of the draft, the main body is almost empty, since few significant problems remain with CoAP or its satellite specifications.

The appendix contains the "CoAP cemetery" (Appendix C, possibly later to move into its own draft), documenting roads that the WG decided not to take, in order to spare readers from reinventing them in vain. There is also a "CoAP museum" (Appendix B), which documents previous forms of proposals part of which did make it into the main documents in one form or another. Finally, the "CoAP nursery" (Appendix A) contains half- to fully-baked proposals that might become interesting as the basis for future extensions.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

The term "byte" is used in its now customary sense as a synonym for "octet".
2. Observing Resources in CoAP

(Co-Author for this section: Matthias Kovatsch)

There are two open issues related to -observe
[I-D.ietf-core-observe]:

- mixing freshness and observation lifetime, and
- non-cacheable resources.

To solve the first issue, we think that -observe should be clarified as follows:

A server sends at least some notifications as confirmable messages. Each confirmable notification is an opportunity for the server to check if the client is still there. If the client acknowledges the notification, it is assumed to be well and alive and still interested in the resource. If it rejects the message with a reset message or if it doesn’t respond, it is assumed not longer to be interested and is removed from the list of observers. So an observation relationship can potentially go on forever, if the client acknowledges each confirmable notification. If the server doesn’t send a notification for a while and wants to check if the client is still there, it may send a confirmable notification with the current resource state to check that.

So there is no mixing of freshness and lifetime going on.

The other issue is a bit less trivial to solve. The problem is that normal CoAP and -observe actually have very different freshness models:

Normally, when a client wants to know the current state of a resource, it retrieves a representation, uses it and stores it in its cache. Later, when it wants to know the current state again, it can either use the stored representation provided that it’s still fresh, or retrieve a new representation, use it and store it in its cache.

If a server knows when the state of the resource will change the next time, it can set the Max-Age of the representation to an accurate time span. So the change of the resource state will coincide with the expiration of the freshness of the representation stored in the client’s cache (ignoring network latency).

But if the resource changes its state unpredictably at any time, the server can set the Max-Age only to an estimate. If the state then actually changes before the freshness expires, the client wrongly
believes it has fresh information. Conversely, if the freshness expires and the client wants to know the current state, the client wrongly believes it has to make a new request although the representation is actually still fresh (this is defused by ETag validation).

-observe doesn’t have these kinds of problems: the server does not have to predict when the resource will change its state the next time. It just sends a notification when it does. The new representation invalidates the old representation stored in the client’s cache. So the client always has a fresh representation that it can use when it wants to know the current resource state without ever having to make a request. An explicit Max-Age is not needed for determining freshness.

But -observe has a different set of problems:

The first problem is that the resource may change its state more often than there is bandwidth available or the client can handle. Thus, -observe cannot make any guarantee that a client will see every state change. The solution is that -observe guarantees that the client will eventually see the latest state change, and follows a best effort approach to enable the client to see as many state changes as possible.

The second problem is that, when a notification doesn’t arrive for a while, the client does not know if the resource did not change its state or if the server lost its state and forgot that the client is interested in the resource. We propose the following solution: With each notification that the server sends, it makes a promise to send another notification, and that it will send this next notification at latest after a certain time span. This time span is included with each notification. So when no notification arrives for a while and the time span has not expired yet, the client assumes that the resource did not change its state. If the time span has expired, no notification has arrived and the client wants to know the current state of the resource, it has to make a new request.

The third problem is that, when an intermediary is observing a resource and wants to create a response from a representation stored in its cache, it needs to specify a Max-Age. But the intermediary cannot predict when it will receive the next notification, because the next notification can arrive at any time. Unlike the origin server, it also doesn’t have the application-specific knowledge that the origin server has. We propose the following solution: With each notification a server sends, it includes a value that an intermediary should use to calculate the Max-Age.
To summarize:

- A notification doesn’t have a Max-Age; it’s fresh until the next notification arrives. A notification is the promise for another notification that will arrive at latest after Next-Notification-At-Latest. This value is included with every notification. The promise includes that the server attempts to transmit a notification to the client for the promised time span, even if the client does not seem to respond, e.g., due to a temporary network outage.

- A notification also contains another value, called Max-Age-Hint. This value is used by a cache to calculate a Max-Age for the representation if needed. In a cache, the Max-Age-Hint of a representation is counted down like Max-Age. When it reaches zero, however, the representation can be still used to satisfy requests, but is non-cacheable (i.e., Max-Age is 0). The Max-Age-Hint must be less than or equal to Next-Notification-At-Latest.

We see two possible ways to encode Next-Notification-At-Latest and Max-Age-Hint in a message:

- The first way is to require the values of Next-Notification-At-Latest and Max-Age-Hint to be the same, although they are conceptually unrelated. Then, a single option in the message can be used to hold both values.

- The second way is to include two options, one for Next-Notification-At-Latest and one for Max-Age-Hint. Since Next-Notification-At-Latest is less than or equal to Max-Age-Hint, the first option should indicates Max-Age-Hint, and the second option Next-Notification-At-Latest minus Max-Age-Hint with a default value of 0.
3. References

3.1. Normative References

[I-D.ietf-core-coap]
Shelby, Z., Hartke, K., Bormann, C., and B. Frank,
"Constrained Application Protocol (CoAP)",
draft-ietf-core-coap-12 (work in progress), October 2012.

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

[I-D.ietf-httpbis-p1-messaging]
Fielding, R., Lafon, Y., and J. Reschke, "HTTP/1.1, part 1: Message Routing and Syntax",
draft-ietf-httpbis-p1-messaging-20 (work in progress),
July 2012.

[I-D.ietf-httpbis-p4-conditional]
Fielding, R., Lafon, Y., and J. Reschke, "HTTP/1.1, part 4: Conditional Requests",
draft-ietf-httpbis-p4-conditional-20 (work in progress),
July 2012.

[I-D.ietf-httpbis-p6-cache]
Fielding, R., Lafon, Y., Nottingham, M., and J. Reschke,
"HTTP/1.1, part 6: Caching",
draft-ietf-httpbis-p6-cache-20 (work in progress),
July 2012.


3.2. Informative References


Appendix A. The Nursery (Things that still need to ripen a bit)

A.1. Envelope Options

As of [I-D.ietf-core-coap], options can take one of four types, two of which are mostly identical:

- uint: A non-negative integer which is represented in network byte order using a variable number of bytes (see [I-D.ietf-core-coap] Appendix A);
- string: a sequence of bytes that is nominally a Net-Unicode string [RFC5198];
- opaque: a sequence of bytes.
- empty (not explicitly identified as a fourth type in [I-D.ietf-core-coap]).

It turns out some options would benefit from some internal structure. Also, it may be a good idea to be able to bundle multiple options into one, in order to ensure consistency for a set of elective options that need to be processed all or nothing (i.e., the option becomes critical as soon as another option out of the set is processed, too).

In this section, we introduce a fifth CoAP option type: Envelope options.

An envelope option is a sequence of bytes that looks and is interpreted exactly like a CoAP sequence of options. Instead of an option count or an end-of-option marker, the sequence of options is terminated by the end of the envelope option.

The nested options (options inside the envelope option) may come from the same number space as the top-level CoAP options, or the envelope option may define its own number space - this choice needs to be defined for each envelope option.

If the top-level number space is used, the envelope option typically will restrict the set of options that actually can be used in the envelope. In particular, it is unlikely that an envelope option will allow itself inside the envelope (this would be a recursive option).

Envelope options are a general, but simple mechanism. Some of its potential uses are illustrated by two examples in the cemetery: Appendix C.1 and Appendix C.2. (Each of these examples has its own merits and demerits, which led us to decide not to pursue either of
them right now, but this should be discussed separately from the concept of Envelope options employed in the examples.)

A.2. Payload-Length Option

Not all transport mappings may provide an unambiguous length of the CoAP message. For UDP, it may also be desirable to pack more than one CoAP message into one UDP payload (aggregation); in that case, for all but the last message there needs to be a way to delimit the payload of that message.

This can be solved using a new option, the Payload-Length option. If this option is present, the value of this option is an unsigned integer giving the length of the payload of the message (note that this integer can be zero for a zero-length payload, which can in turn be represented by a zero-length option value). (In the UDP aggregation case, what would have been in the payload of this message after "payload-length" bytes is then actually one or more additional messages.)

A.3. URI Authorities with Binary Addresses

One problem with the way URI authorities are represented in the URI syntax is that the authority part can be very bulky if it encodes an IPv6 address in ASCII.

Proposal: Provide an option "Uri-Authority-Binary" that can be an even number of bytes between 2 and 18 except 12 or 14.

- If the number of bytes is 2, the destination IP address of the packet transporting the CoAP message is implied.
- If the number of bytes is 4 or 6, the first four bytes of the option value are an IPv4 address in binary.
- If the number of bytes is 8 or 10, the first eight bytes are the lower 64 bits of an IPv6 address; the upper eight bytes are implied from the destination address of the packet transporting the CoAP message.
- If the number of bytes is 16 or 18, the first 16 bytes are an IPv6 address.
- If two more bytes remain, this is a port number (as always in network byte order).

The resulting authority is (conceptually translated into ASCII and) used in place of an Uri-Authority option, or inserted into a Proxy-
Uri. Examples:

<table>
<thead>
<tr>
<th>Proxy-Uri</th>
<th>Uri-Authority-Binary</th>
<th>Uri-Pattern</th>
<th>URI</th>
</tr>
</thead>
<tbody>
<tr>
<td>(none)</td>
<td>(none)</td>
<td>(none)</td>
<td>&quot;/&quot;</td>
</tr>
<tr>
<td>(none)</td>
<td>(none)</td>
<td>'temp'</td>
<td>&quot;/temp&quot;</td>
</tr>
<tr>
<td>(none)</td>
<td>2 bytes: 61616</td>
<td>'temp'</td>
<td>&quot;coap://[DA]:61616/temp&quot;</td>
</tr>
<tr>
<td>(none)</td>
<td>16 bytes: 2000::1</td>
<td>temp</td>
<td>&quot;coap://[2000::1]/temp&quot;</td>
</tr>
<tr>
<td>'http://'</td>
<td>10 bytes: ::123:45 + 616</td>
<td>(none)</td>
<td>&quot;http://[DA::123:45]:616/16&quot;</td>
</tr>
<tr>
<td>'<a href="http://temp">http://temp</a>'</td>
<td>18 bytes: 2000::1 + 616</td>
<td>(none)</td>
<td>&quot;http://[2000::1]:616/temp&quot;</td>
</tr>
</tbody>
</table>

A.4. Length-aware number encoding (o256)

The number encoding defined in Appendix A of [I-D.ietf-core-coap] has one significant flaw: Every number has an infinite number of representations, which can be derived by adding leading zero bytes. This runs against the principle of minimizing unnecessary choice. The resulting uncertainty in encoding ultimately leads to unnecessary interoperability failures. (It also wastes a small fraction of the encoding space, i.e., it wastes bytes.)

We could solve the first, but not the second, by outlawing leading zeroes, but then we have to cope with error cases caused by illegal values, another source of interoperability problems.

The number encoding "o256" defined in this section avoids this flaw. The suggestion is not to replace CoAP’s "uint" encoding wholesale (CoAP is already too widely implemented for such a change), but to consider this format for new options.

The basic requirements for such an encoding are:

- numbers are encoded as a sequence of zero or more bytes
- each number has exactly one encoding
o for a < b, encoding-size(a) <= encoding-size(b) -- i.e., with larger numbers, the encoding only gets larger, never smaller again.

o within each encoding size (0 bytes, 1 byte, etc.), lexicographical ordering of the bytes is the same as numeric ordering

Obviously, there is only one encoding that satisfies all these requirements. As illustrated by Figure 1, this is unambiguously derived by

1. enumerating all possible byte sequences, ordered by length and within the same length in lexicographic ordering, and,

2. assigning sequential cardinals.

```
0x''  ->  0
0x'00' ->  1
0x'01' ->  2
0x'02' ->  3
...
0x'fe'  ->  255
0x'ff'  ->  256
0x'0000' ->  257
0x'0001' ->  258
...
0x'fefd' ->  65534
0x'ffee' ->  65535
0x'efff' ->  65536
...
0x'ffff' ->  65792
0x'000000' ->  65793
0x'000001' ->  65794
```

Figure 1: Enumerating byte sequences by length and then lexicographic order

This results in an exceedingly simple algorithm: each byte is interpreted in the base-256 place-value system, but stands for a number between 1 and 256 instead of 0 to 255. We therefore call this encoding "o256" (one-to-256). 0 is always encoded in zero bytes; 1 to 256 is one byte, 257 (0x101) to 65792 (0x10100) is two bytes, 65793 (0x10101) to 16843008 (0x1010100) is three bytes, etc.

To further illustrate the algorithmic simplicity, pseudocode for encoding and decoding is given in Figure 2 and Figure 3, respectively (in the encoder, "prepend" stands for adding a byte at the _leading_
edge, the requirement for which is a result of the network byte order). Note that this differs only  in a single subtraction/addition (resp.)  of one from the canonical algorithm for Appendix A uints.

```plaintext
while num > 0
    num -= 1
    prepend(num & 0xFF)
    num >>= 8
end
```

Figure 2: o256 encoder (pseudocode)

```plaintext
num = 0
each_byte do |b|
    num <<= 8
    num += b + 1
end
```

Figure 3: o256 decoder (pseudocode)

On a more philosophical note, it can be observed that o256 solves the inverse problem of Self-Delimiting Numeric Values (SDNV) [RFC6256]: SDNV encodes variable-length numbers together with their length (allowing decoding without knowing their length in advance, deriving delimiting information from the number encoding). o256 encodes variable-length numbers when there is a way to separately convey the length (as in CoAP options), encoding (and later deriving) a small part of the numeric value into/from that size information.

### A.5. SMS encoding

For use in SMS applications, CoAP messages can be transferred using SMS binary mode. However, there is operational experience showing that some environments cannot successfully send a binary mode SMS.

For transferring SMS in character mode (7-bit characters), base64-encoding [RFC4648] is an obvious choice. 3 bytes of message (24 bits) turn into 4 characters, which consume 28 bits. The overall overhead is approximately 17 %; the maximum message size is 120 bytes (160 SMS characters).

If a more compact encoding is desired, base85 encoding can be employed (however, probably not the version defined in [RFC1924] -- instead, the version used in tools such as btoa and PDF should be chosen). However, this requires division operations. Also, the base85 character set includes several characters that cannot be transferred in a single 7-bit unit in SMS and/or are known to cause
operational problems. A modified base85 character set can be defined to solve the latter problem. 4 bytes of message (32 bits) turn into 5 characters, which consume 35 bits. The overall overhead is approximately 9.3 %; the resulting maximum message size is 128 bytes (160 SMS characters).

Base64 and base85 do not make use of the fact that much CoAP data will be ASCII-based. Therefore, we define the following experimental SMS encoding.

A.5.1. ASCII-optimized SMS encoding

Not all 128 theoretically possible SMS characters are operationally free of problems. We therefore define:

Shunned code characters: @ sign, as it maps to 0x00

- LF and CR signs (0x0A, 0x0D)
- Uppercase C cedilla (0x09), as it is often mistranslated in gateways
- ESC (0x1B), as it is used in certain character combinations only

Some ASCII characters cannot be transferred in the base SMS character set, as their code positions are taken by non-ASCII characters. These are simply encoded with their ASCII code positions, e.g., an underscore becomes a section mark (even though underscore has a different code position in the SMS character set).

Equivalently translated input bytes: $, @, [, \, ], ^, _, ', {, |, }, ~, DEL

In other words, bytes 0x20 to 0x7F are encoded into the same code positions in the 7-bit character set.

Out of the remaining code characters, the following SMS characters are available for encoding:

Non-equivalently translated (NET) code characters: 0x01 to 0x08, (8 characters)

- 0x0B, 0x0C, (2 characters)
- 0x0E to 0x1A, (13 characters)
0x1C to 0x1F, (4 characters)

Of the 27 NET code characters, 18 are taken as prefix characters (see below), and 8 are defined as directly translated characters:

Directly translated bytes: Equivalently translated input bytes are represented as themselves

0x00 to 0x07 are represented as 0x01 to 0x08

This leaves 0x08 to 0x1F and 0x80 to 0xFF. Of these, the bytes 0x80 to 0x87 and 0xA0 to 0xFF are represented as the bytes 0x00 to 0x07 (represented by characters 0x01 to 0x08) and 0x20 to 0x7F, with a prefix of 1 (see below). The characters 0x08 to 0x1F are represented as the characters 0x28 to 0x3F with a prefix of 2 (see below). The characters 0x88 to 0x9F are represented as the characters 0x48 to 0x5F with a prefix of 2 (see below). (Characters 0x01 to 0x08, 0x20 to 0x27, 0x40 to 0x47, and 0x60 to 0x7F with a prefix of 2 are reserved for future extensions, which could be used for some backreferencing or run-length compression.)

Bytes that do not need a prefix (directly translated bytes) are sent as is. Any byte that does need a prefix (i.e., 1 or 2) is preceded by a prefix character, which provides a prefix for this and the following two bytes as follows:

```
+------+-----+---+------+-----+
| 0x0B | 100 | 0x15 | 200 |
+------+-----+---+------+-----+
| 0x0C | 101 | 0x16 | 201 |
| 0x0E | 102 | 0x17 | 202 |
| 0x0F | 110 | 0x18 | 210 |
| 0x10 | 111 | 0x19 | 211 |
| 0x11 | 112 | 0x1A | 212 |
| 0x12 | 120 | 0x1C | 220 |
| 0x13 | 121 | 0x1D | 221 |
| 0x14 | 122 | 0x1E | 222 |
```

(This leaves one non-shunned character, 0x1F, for future extension.)
The coding overhead of this encoding for random bytes is similar to Base85, without the need for a division/multiplication. For bytes that are mostly ASCII characters, the overhead can easily become negative. (Conversely, for bytes that are more likely to be non-ASCII than in a random sequence of bytes, the overhead becomes greater.)

So, for instance, for the CoAP message in Figure 4:

```
<table>
<thead>
<tr>
<th>ver</th>
<th>tt</th>
<th>code</th>
<th>mid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ack</td>
<td>2.05</td>
<td>17033</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>content_type</td>
<td>40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>token</td>
<td>sometok</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
3c 2f 3e 3b 74 69 74 6c 65 3d 22 47 65 6e 65 72 61 6c 20 49 6e 66 6f 22 3b 63 74 3d 30 2c 3c 2f 74 69 6d 65 3e 3b 69 66 3d 22 63 6c 6f 63 6b 22 3b 72 74 3d 22 54 69 63 6b 73 22 3b 74 69 74 6c 65 3d 22 49 6e 74 65 72 6e 61 6c 20 43 6c 6f 63 6b 22 3b 63 74 3d 30 2c 3c 2f 61 73 79 6e 63 3e 3b 63 74 3d 30 |

Figure 4: CoAP response message as captured and decoded

The 116 byte unencoded message is shown as ASCII characters in Figure 5 (\xDD stands for the byte with the hex digits DD):

```
bEB\x91\x82\x11\xA7'sometok</;'title="General Info";ct=0</time>;if="clock";rt="Ticks";title="Internal Clock";ct=0</async>;ct=0 |
```

Figure 5: CoAP response message shown as unencoded characters

The equivalent SMS encoding is shown as equivalent-coded SMS characters in Figure 6 (7 bits per character, \x12 is a 220 prefix and \x0B is a 100 prefix, the rest is shown in equivalent encoding), adding two characters of prefix overhead, for a total length of 118 7-bit characters or 104 (103.25 plus padding) bytes:

```
bEB\x12\x0B'sometok</;'title="General Info";ct=0</time>;if="clock";rt="Ticks";title="Internal Clock";ct=0</async>;ct=0 |
```

Figure 6: CoAP response message shown as SMS-encoded characters

A.6. CONNECT

[RFC2817] defines the HTTP CONNECT method to establish a TCP tunnel through a proxy so that end-to-end TLS connections can be made through the proxy. Recently, a requirement for similar functionality has been discussed for CoAP. This section defines a straw-man
CONNECT method and related methods and response codes for CoAP.

(IANA considerations for this section TBD.)

A.6.1. Requesting a Tunnel with CONNECT

CONNECT is allocated as a new method code in the "CoAP Method Codes" registry. When a client makes a CONNECT request to an intermediary, the intermediary evaluates the Uri-Host, Uri-Port, and/or the authority part of the Proxy-Uri Options in a way that is defined by the security policy of the intermediary. If the security policy allows the allocation of a tunnel based on these parameters, the method returns an empty payload and a response code of 2.30 Tunnel Established. Other possible response codes include 4.03 Forbidden.

It may be the case that the intermediary itself can only reach the requested origin server through another intermediary. In this case, the first intermediary SHOULD make a CONNECT request of that next intermediary, requesting a tunnel to the authority. A proxy MUST NOT respond with any 2.xx status code unless it has either a direct or tunnel connection established to the authority.

An origin server which receives a CONNECT request for itself MAY respond with a 2.xx status code to indicate that a tunnel is established to itself.

Code 2.30 "Tunnel Established" is allocated as a new response code in the "CoAP Response Codes" registry.

A.6.2. Using a CONNECT Tunnel

Any successful (2.xx) response to a CONNECT request indicates that the intermediary has established a tunnel to the requested host and port. The tunnel is bound to the requesting end-point and the Token supplied in the request (as always, the default Token is admissible). The tunnel can be used by the client by making a DATAGRAM request.

DATAGRAM is allocated as a new method code in the "CoAP Method Codes" registry. When a client makes a DATAGRAM request to an intermediary, the intermediary looks up the tunnel bound to the client end-point and Token supplied in the DATAGRAM request (no other Options are permitted). If a tunnel is found and the intermediary's security policy permits, the intermediary forwards the payload of the DATAGRAM request as the UDP payload towards the host and port established for the tunnel. No response is defined for this request (note that the request can be given as a CON or NON request; for CON, there will be an ACK on the message layer if the tunnel exists).
The security policy on the intermediary may restrict the allowable payloads based on its security policy, possibly considering host and port. An inadmissible payload SHOULD cause a 4.03 Forbidden response with a diagnostic message as payload.

The UDP payload of any datagram received from the tunnel and admitted by the security policy is forwarded to the client as the payload of a 2.31 "Datagram Received" response. The response does not carry any Option except for Token, which identifies the tunnel towards the client.

Code 2.31 "Datagram Received" is allocated as a new response code in the "CoAP Response Codes" registry.

An origin server that has established a tunnel to itself processes the CoAP payloads of related DATAGRAM requests as it would process an incoming UDP payload, and forwards what would be outgoing UDP payloads in 2.31 "Datagram Received" responses.

A.6.3. Closing down a CONNECT Tunnel

A 2.31 "Datagram Received" response may be replied to with a RST, which closes down the tunnel. Similarly, the Token used in the tunnel may be reused by the client for a different purpose, which also closes down the tunnel.
Appendix B. The Museum (Things we did, but maybe not exactly this way)

B.1. Getting rid of artificial limitations

_Artificial limitations_ are limitations of a protocol or system that are not rooted in limitations of actual capabilities, but in arbitrary design decisions. Proper system design tries to avoid artificial limitations, as these tend to cause complexity in systems that need to work with these limitations.

E.g., the original UNIX filesystem had an artificial limitation of the length of a path name component to 14 bytes. This led to a cascade of workarounds in programs that manipulate file names: E.g., systematically replacing a ".el" extension in a filename with a ".elc" for the compiled file might exceed the limit, so all ".el" files were suddenly limited to 13-byte filenames.

Note that, today, there still is a limitation in most file system implementations, typically at 255. This just happens to be high enough to rarely be of real-world concern; we will refer to this case as a "painless" artificial limitation.

CoAP-08 had two highly recognizable artificial limitations in its protocol encoding

- The number of options in a single message is limited to 15 max.
- The length of an option is limited to 270 max.

It has been argued that the latter limitation causes few problems, just as the 255-byte path name component limitation in filenames today causes few problems. Appendix B.1.1 provided a design to extend this; as a precaution to future extensions of this kind, the current encoding for length 270 (eight ones in the extension byte) could be marked as reserved today. Since, Matthias Kovatsch has proposed a simpler scheme that seems to gain favor in the WG, see Appendix B.1.4.

The former limitation has been solved in CoAP-09. A historical discussion of other approaches for going beyond 15 options is in Appendix B.1.2. Appendix B.1.3 discusses implementation.

B.1.1. Beyond 270 bytes in a single option

The authors would argue that 270 as the maximum length of an option is already beyond the "painless" threshold.

If that is not the consensus of the WG, the scheme can easily be
extended as in Figure 7:

for 15..269:

+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| Option Delta  | 1   1   1   1 |          Length - 15          |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| Option Value ...
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

for 270..65805:

+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| Option Delta  | 1   1   1   1 | 1   1   1   1   1   1   1   1 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| Length - 270 (in network byte order) |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| Option Value ...
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

Figure 7: Ridiculously Long Option Header

The infinite number of obvious variations on this scheme are left as an exercise to the reader.

Again, as a precaution to future extensions, the current encoding for length 270 (eight ones in the extension byte) could be marked as reserved today.

B.1.2. Beyond 15 options

(This section keeps discussion that is no longer needed as we have agreed to do what is documented in Appendix B.1.3).

The limit of 15 options is motivated by the fixed four-bit field "OC" that is used for indicating the number of options in the fixed-length CoAP header (Figure 8).

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
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Ver| T |  OC   |      Code     |          Message ID           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Options (if any) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Payload (if any) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

Figure 8: Four-byte fixed header in a CoAP Message
Note that there is another fixed four-bit field in CoAP: the option length (Figure 9 - note that this figure is not to the same scale as the previous figure):

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

Figure 9: Short Option Header

Since 15 is inacceptable for a maximum option length, the all-ones value (15) was taken out of the set of allowable values for the short header, and a long header was introduced that allows the insertion of an extension byte (Figure 10):

```
for 15..270:
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| Option Delta | 1 1 1 1 | Length - 15 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|       Option Value ...
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

Figure 10: Long Option Header

We might want to use the same technique for the CoAP header as well. There are two obvious places where the extension byte could be placed:

1. right after the byte carrying the OC field, so the structure is the same as for the option header;

2. right after the fixed-size CoAP header.

Both solutions lose the fixed-size-ness of the CoAP header.

Solution 1 has the disadvantage that the CoAP header is also changing in structure: The extension byte is wedged between the first and the second byte of the CoAP header. This is unfortunate, as the number of options only comes into play when the option processing begins, so it is more natural to use solution 2 (Figure 11):
This would allow for up to 270 options in a CoAP message, which is very likely way beyond the "painless" threshold.

B.1.2.1. Implementation considerations

For a message decoder, this extension creates relatively little pain, as the number of options only becomes interesting when the encoding turns to the options part of the message, which is then simply lead in by the extension byte if the four-bit field is 15.

For a message encoder, this extension is not so rosy. If the encoder is constructing the message serially, it may not know in advance whether the number of options will exceed 14. None of the following implementation strategies is particularly savory, but all of them do work:

1. Encode the options serially under the assumption that the number of options will be 14 or less. When the 15th option needs to be encoded, abort the option encoding, and restart it from scratch one byte further to the left.

2. Similar to 1, except that the bytes already encoded are all moved one byte to right, the extension byte is inserted, and the option encoding process is continued.

3. The encoder always leaves space for the extension byte (at least if it can’t prove the number will be less than 14). If the extension byte is not needed, an Option 0 with length 0 is encoded instead (i.e., one byte is wasted – this option is elective and will be ignored by the receiver).

As a minimum, to enable strategy 3, the option 0 should be reserved at least for the case of length=0.
B.1.2.2. What should we do now?

As a minimum proposal for the next version of CoAP, the value 15 for OC should be marked as reserved today.

B.1.2.3. Alternatives

One alternative that has been discussed previously is to have an "Options" Option, which allows the carriage of multiple options in the belly of a single one. This could also be used to carry more than 15 options. However:

- The conditional introduction of an Options option has implementation considerations that are likely to be more severe than the ones listed above;

- Since 270 bytes may not be enough for the encoding of _all_ options, the "Options" option would need to be repeatable. This creates many different ways to encode the same message, leading to combinatorial explosion in test cases for ensuring interoperability.

B.1.2.4. Alternative: Going to a delimiter model

Another alternative is to spend the additional byte not as an extended count, but as an option terminator.

B.1.3. Implementing the option delimiter for 15 or more options

Implementation note: As can be seen from the proof of concept code in Figure 12, the actual implementation cost for a decoder is around 4 lines of code (or about 8-10 machine code instructions).

```plaintext
while numopt > 0
    nextbyte = ... get next byte
    if numopt == 15                  # new
        break if nextbyte == 0xF0    # new
    else                            # new
        numopt -= 1
    end                              # new

    ... decode the delta and length from nextbyte and handle them
end
```

Figure 12: Implementing the Option Terminator
Similarly, creating the option terminator needs about four more lines (not marked "old" in the C code in Figure 13).

```c
b0 = 0x40 + (tt << 4);              /* old */
buffer[0] = b0 + 15;                /* guess first byte */

.... encode options ....           /* old */

if (option_count >= 15 || first_fragment_already_shipped)
  buffer[pos++] = 0xF0;            /* use delimiter */
else                                /* save a byte: */
  buffer[0] = b0 + option_count;   /* old: backpatch */
```

Figure 13: Creating the Option Terminator

B.1.4. Option Length encoding beyond 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" Figure 14. While this causes the length of the option header to grow linearly with the size of the option value, only 0.4 % of that size is used. With a focus on short options, this encoding is justified.
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 invalid option.

In the process, the maximum length of all options that are currently set at 270 should now be set to a carefully chosen value. With the purely encoding-based limit gone, Uri-Proxy should now be restored to be a non-repeatable option.
A first proposal for a new set of per-option length restrictions follows:

<table>
<thead>
<tr>
<th>number</th>
<th>name</th>
<th>min</th>
<th>max</th>
<th>type</th>
<th>repeat</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>content_type</td>
<td>0</td>
<td>2</td>
<td>uint</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>max_age</td>
<td>0</td>
<td>4</td>
<td>uint</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>proxy_uri</td>
<td>1</td>
<td>1023</td>
<td>string</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>etag</td>
<td>1</td>
<td>8</td>
<td>opaque</td>
<td>yes</td>
</tr>
<tr>
<td>5</td>
<td>uri_host</td>
<td>1</td>
<td>255</td>
<td>string</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>location_path</td>
<td>0</td>
<td>255</td>
<td>string</td>
<td>yes</td>
</tr>
<tr>
<td>7</td>
<td>uri_port</td>
<td>0</td>
<td>2</td>
<td>uint</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>location_query</td>
<td>0</td>
<td>255</td>
<td>string</td>
<td>yes</td>
</tr>
<tr>
<td>9</td>
<td>uri_path</td>
<td>0</td>
<td>255</td>
<td>string</td>
<td>yes</td>
</tr>
<tr>
<td>10</td>
<td>observe</td>
<td>0</td>
<td>2</td>
<td>uint</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>token</td>
<td>1</td>
<td>8</td>
<td>opaque</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>accept</td>
<td>0</td>
<td>2</td>
<td>uint</td>
<td>yes</td>
</tr>
<tr>
<td>13</td>
<td>if_match</td>
<td>0</td>
<td>8</td>
<td>opaque</td>
<td>yes</td>
</tr>
<tr>
<td>14</td>
<td>registered_elective</td>
<td>1</td>
<td>1023</td>
<td>opaque</td>
<td>yes</td>
</tr>
<tr>
<td>15</td>
<td>uri_query</td>
<td>1</td>
<td>255</td>
<td>string</td>
<td>yes</td>
</tr>
<tr>
<td>17</td>
<td>block2</td>
<td>0</td>
<td>3</td>
<td>uint</td>
<td>-</td>
</tr>
<tr>
<td>18</td>
<td>size</td>
<td>0</td>
<td>4</td>
<td>uint</td>
<td>-</td>
</tr>
<tr>
<td>19</td>
<td>block1</td>
<td>0</td>
<td>3</td>
<td>uint</td>
<td>-</td>
</tr>
<tr>
<td>21</td>
<td>if_none_match</td>
<td>0</td>
<td>0</td>
<td>empty</td>
<td>-</td>
</tr>
<tr>
<td>25</td>
<td>registered_critical</td>
<td>1</td>
<td>1023</td>
<td>opaque</td>
<td>yes</td>
</tr>
</tbody>
</table>

(Option 14 with a length of 0 is a fencepost only.)
B.2. Registered Option

CoAP’s option encoding is highly efficient, but works best with small option numbers that do not require much fenceposting. The CoAP Option Number Registry therefore has a relatively heavyweight registration requirement: "IETF Review" as described in [RFC5226].

However, there is also considerable benefit in a much looser registry policy, enabling a first-come-first-served policy for a relatively large option number space.

Here, we discuss two solutions that enable such a registry. One is to define a separate mechanism for registered options, discussed in Appendix B.2.1. Alternatively, we could make it easier to use a larger main option number space, discussed in Appendix B.2.2.

B.2.1. A Separate Suboption Number Space

This alternative defines a separate space of suboption numbers, with an expert review [RFC5226] (or even first-come-first-served) registration policy. If expert review is selected for this registry, it would be with a relatively loose policy delegated to the expert. This draft proposes leaving the registered suboption numbers 0-127 to expert review with a policy that mainly focuses on the availability of a specification, and 128-16383 for first-come-first-served where essentially only a name is defined.

The "registered" options are used in conjunction with this suboption number registry. They use two normal CoAP option numbers, one for options with elective semantics (Registered-Elective) and one for options with critical semantics (Registered-Critical). The suboption numbers are not separate, i.e. one registered suboption number might have some elective semantics and some other critical semantics (e.g., for the request and the response leg of an exchange). The option value starts with an SDNV [RFC6256] of the registered suboption number. (Note that there is no need for an implementation to understand SDNVs, it can treat the prefixes as opaque. One could consider the SDNVs as a suboption prefix allocation guideline for IANA as opposed to a number encoding.)

```
+-----------------------------+-----------------------------+
| 1 0 0 0 0 0 1 | 0 1 1 1 0 0 1 |
|           value...           |
\___SDNV of registered number___/
```

Figure 15: Example option value for registered option

Note that a Registered Option cannot be empty, because there would be
no space for the SDNV. Also, the empty option 14 is reserved for fenceposting ([I-D.ietf-core-coap], section 3.2). (Obviously, once a Registered-Elective Option is in use, there is never a need for a fence-post for option number 14.)

The Registered-Elective and Registered-Critical Options are repeatable.

<table>
<thead>
<tr>
<th>No.</th>
<th>C/E</th>
<th>Name</th>
<th>Format</th>
<th>Length</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>Elective</td>
<td>Registered-Elective</td>
<td>(see above)</td>
<td>1-1023</td>
<td>(none)</td>
</tr>
<tr>
<td>25</td>
<td>Critical</td>
<td>Registered-Critical</td>
<td>(see above)</td>
<td>1-1023</td>
<td>(none)</td>
</tr>
</tbody>
</table>

This solves CoRE issue #214 [CoRE214]. (How many options we need will depend on the resolution of #241 [CoRE241].)

**B.2.2. Opening Up the Option Number Space**

The disadvantage of the registered-... options is that there is a significant syntactic difference between options making use of this space and the usual standard options. This creates a problem not unlike that decried in [RFC6648].

The alternative discussed in this section reduces the distance by opening up the main Option number space instead.

There is still a significant incentive to use low-numbered Options. However, the proposal reduces the penalty for using a high-numbered Option to two or three bytes. More importantly, using a cluster of related high-numbered options only carries a total penalty of two or three bytes.

The main reason high-numbered options are expensive to use and thus the total space is relatively limited is that the option delta mechanism only allows increasing the current option number by up to 14 per one-byte fencepost. To use, e.g., Option number 1234 together with the usual set of low-numbered Options, one needs to insert 88 fence-post bytes. This is prohibitive.

Enabling first-come-first-served probably requires easily addressing a 16-bit option number space, with some potential increase later in the lifetime of the protocol (say, 10 to 15 years from now).
To enable the use of large option numbers, one needs a way to advance the Option number in bigger steps than possible by the Option Delta. So we propose a new construct, the Long Jump construct, to move the Option number forward.

### B.2.2.1. Long Jump construct

The following construct can occur in front of any Option:

```
+---+---+---+---+---+---+---+---+
| 1   1   1   1 | 0   0   0   1 |  0xf1 (Delta = 15) |
+---+---+---+---+---+---+---+---+
```

```
+---+---+---+---+---+---+---+---+
| 1   1   1   1 | 0   0   1   0 |  0xf2 |
```

```
| Long Jump Value | (Delta/8)-2 |
```

```
+---+---+---+---+---+---+---+---+
| 1   1   1   1 | 0   0   1   1 |  0xf3 |
```

```
| Long Jump Value, MSB | |
```

```
| Long Jump Value, LSB | (Delta/8)-258 |
```

Figure 16: Long 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 in multiples of eight. More specifically, the actual addition to the current Option number is computed as follows:

\[
\text{Delta} = ((\text{Long Jump Value}) + N) \times 8
\]

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

A Long Jump MUST be followed by an actual Option, i.e., it MUST NOT be followed by another Long Jump or an end-of-options indicator. A message violating this MUST be rejected as malformed.

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

B.2.2.2. Discussion

Adding a mechanism at this late stage creates concerns of backwards compatibility. A message sender never needs to implement long-jumps unless it wants to make use of a high-numbered option. So this mechanism can be added once a high-numbered option is added. A message receiver, though, would more or less unconditionally have to implement the mechanism, leading to unconditional additional complexity. There are good reasons to minimize this, as follows:

- The increase in multiples of eight allows looking at an option and finding out whether it is critical or not even if the Long Jump value has just been skipped (as opposed to having been processed fully). (It also allows accessing up to approximately 2048 options with a two-byte Long Jump.) This allows a basic implementation that does not implement any high-numbered options to simply ignore long jumps and any elective options behind them, while still properly reacting to critical options.

- There is probably a good reason to disallow long-jumps that lead to an option number of 42 and less, enabling simple receivers to do the above simplification.

- It might seem obvious to remove the fenceposting mechanism altogether in favor of long jumps. This is not advisable: Fenceposting already has zero implementation effort at the receiver, and the overhead at the sender is very limited (it is just a third kind of jump, at one byte per jump). Beyond 42, senders can ignore the existence of fenceposts if they want (possibly obviating the need for more complex base-14 arithmetic).

There is no need for a finer granularity than 8, as the Option construct following can also specify a Delta of 0..14. (A granularity of 16 will require additional fenceposting where an option delta of 15 would happen to be required otherwise, which we have reserved. It can be argued that 16 is still the better choice, as fenceposting is already in the code path.)

The Long Jump construct takes 0xf1 and 0xf2 from the space available for initial bytes of Options. (Note that we previously took 0xf0 to indicate end-of-options for OC=15.)

Varying N with the length as defined above makes it unambiguous whether a one- or two-byte Long Jump is to be used. Setting N=2 for the one-byte version makes it clear that a Delta of 8 is to be handled the usual way (i.e., by Option Delta itself and/or
fenceposting). If the delta is not small and not 7 modulo 8, there is still a choice between using the smaller multiple of 8 and a larger Delta in the actual Option or v.v., this biases the choice towards a larger Long Jump and a smaller following Delta, which is also easier to implement as it reduces the number of choice points.

B.2.2.3. Example

The following sequence of bytes would encode a Uri-Path Option of "foo" followed by Options 1357 (value "bar") and 1360 (value "baz"):

```
93 65 6f 6f       Option 9 (0 + 9, "foo")
f1 a6             Long Jump by 1344
43 62 61 72       Option 1357 (9 + 1344 + 4, "bar")
33 62 61 7a       Option 1360 (1357 + 3, "baz")
```

Figure 17: Example using a Long Jump construct

where f1 a6 is the long jump forward by (0xa6+2)*8=1344 option numbers. The total option count (OC) for the CoAP header is 3. Note that even if f1 a6 is skipped, the 1357 (which then appears as an Option number 13) is clearly visible as Critical.

B.2.2.4. IANA considerations

With the scheme proposed above, we could have three tiers of Option Numbers:

<table>
<thead>
<tr>
<th>Option Number</th>
<th>Policy [RFC5226]</th>
</tr>
</thead>
<tbody>
<tr>
<td>0..255</td>
<td>Standards Action</td>
</tr>
<tr>
<td>256..2047</td>
<td>Designated Expert</td>
</tr>
<tr>
<td>2048..65535</td>
<td>First Come First Served</td>
</tr>
</tbody>
</table>

For the inventor of a new option, this would provide a small incentive to go through the designated expert for some minimal cross-checking in order to be able to use the two-byte long-jump.

This draft adds option numbers to Table 2 of [I-D.ietf-core-coap]:

Bormann & Hartke Expires April 5, 2013 [Page 31]
Table 1: New CoAP Option Numbers

This draft adds a suboption registry, initially empty.

| Number | Name                | Reference |
|--------|---------------------+-----------|
| 14     | Registered-Elective | [RFCXXXX] |
| 25     | Registered-Critical | [RFCXXXX] |

Table 2: CoAP Suboption Numbers

B.3. Enabling Protocol Evolution

To enable a protocol to evolve, it is critical that new capabilities can be introduced without requiring changes in components that don’t really care about the capability. One such problem is exhibited by CoAP options: If a proxy does not understand an elective option in a request, it will not be able to forward it to the origin server, rendering the new option ineffectual. Worse, if a proxy does not understand a critical option in a request, it will not be able to operate on the request, rendering the new option damaging.

As a conclusion to the Ticket #230 discussion in the June 4th interim call, we decided to solve the identification of options that a proxy can safely forward even if not understood (previously called Proxy-Elective).

The proposal is to encode this information in the option number, just like the way the information that an option is critical is encoded now. This leads to two bits with semantics: the lowest bit continues to be the critical bit, and the next higher bit is now the "unsafe" bit (i.e., this option is not safe to forward unless understood by the proxy).

Another consideration (for options that are not unsafe to forward) is whether the option should serve as a cache key in a request. HTTP has a vary header that indicates in the response which header fields
were considered by the origin server to be cache keys. In order to avoid this complexity, we should be able to indicate this information right in the option number. However, reserving another bit is wasteful, in particular as there are few safe-to-forward options that are not cache-keys.

Therefore, we propose the following bit allocation in an option number:

xxx nnn UC

Figure 18

(where xxx is a variable length prefix, as option numbers are not bounded upwards). UC is the unsafe and critical bits. For U=0 only, if nnn is equal to 111 binary, the option does not serve as a cache key (for U=1, the proxy has to know the option to act on it, so there is no point in indicating whether it is a cache key). There is no semantic meaning of xxx.

Note that clients and servers are generally not interested in this information. A proxy may use an equivalent of the following C code to derive the characteristics of an option number "onum":

```c
Critical = (onum & 1);
Unsafe = (onum & 2);
NoCache = ((onum & 0x1e) == 0x1c);
```

Figure 19

Discussion: This requires a renumbering of all options.

This renumbering may also be considered as an opportunity to make the numbering straight again shortly before nailing down the protocol.

In particular, Content-Type is now probably better considered to be elective.

### B.3.1. Potential new option number allocation

We want to give one example for a revised allocation of option numbers. Option numbers are given as decimal numbers, one each for xxx, nnn, and UC, with the UC values as follows
<table>
<thead>
<tr>
<th>UC binary</th>
<th>UC decimal</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>(safe, elective, 111=no-cache-key)</td>
</tr>
<tr>
<td>01</td>
<td>1</td>
<td>(safe, critical, 111=no-cache-key)</td>
</tr>
<tr>
<td>10</td>
<td>2</td>
<td>(unsafe, elective)</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>(unsafe, critical)</td>
</tr>
</tbody>
</table>

The table is:

<table>
<thead>
<tr>
<th>New</th>
<th>xx nnn</th>
<th>Old</th>
<th>Name</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>UC</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-----</td>
<td>--------</td>
<td>-----</td>
<td>-------------------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>4</td>
<td>0 1 0</td>
<td>1</td>
<td>Content-Type</td>
<td>category change</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(elective)</td>
</tr>
<tr>
<td>8</td>
<td>0 2 0</td>
<td>4</td>
<td>ETag</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>0 3 0</td>
<td>12</td>
<td>Accept</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>0 4 0</td>
<td>6</td>
<td>Location-Path</td>
<td></td>
</tr>
<tr>
<td>20</td>
<td>0 5 0</td>
<td>8</td>
<td>Location-Query</td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>0 6 0</td>
<td>-</td>
<td>(unused)</td>
<td></td>
</tr>
<tr>
<td>28</td>
<td>0 7 0</td>
<td>18</td>
<td>Size</td>
<td>needs nnn=111</td>
</tr>
<tr>
<td>32</td>
<td>1 0 0</td>
<td>20/22</td>
<td>Patience</td>
<td></td>
</tr>
<tr>
<td>64</td>
<td>2 x 0</td>
<td>-</td>
<td>Location-reserved</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>(nnn = 0..3, 4 reserved numbers)</td>
</tr>
<tr>
<td>1</td>
<td>0 0 1</td>
<td>13</td>
<td>If-Match</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0 1 1</td>
<td>21</td>
<td>If-None-Match</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0 0 2</td>
<td>2</td>
<td>Max-Age</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>0 1 2</td>
<td>10</td>
<td>Observe</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>0 2 2</td>
<td>xx</td>
<td>Observe-2</td>
<td></td>
</tr>
</tbody>
</table>
B.4. Patience, Leisure, and Pledge

A number of options might be useful for controlling the timing of interactions.

(This section also addresses core-coap ticket #177.)

B.4.1. Patience

A client may have a limited time period in which it can actually make use of the response for a request. Using the Patience option, it can provide an (elective) indication how much time it is willing to wait for the response from the server, giving the server license to ignore or reject the request if it cannot fulfill it in this period.

If the server knows early that it cannot fulfill the request in the time requested, it MAY indicate this with a 5.04 "Timeout" response. For non-safe methods (such as PUT, POST, DELETE), the server SHOULD indicate whether it has fulfilled the request by either responding with 5.04 "Timeout" (and not further processing the request) or by processing the request normally.

Note that the value of the Patience option should be chosen such that the client will be able to make use of the result even in the presence of the expected network delays for the request and the response. Similarly, when a proxy receives a request with a Patience option and cannot fulfill that request from its cache, it may want to adjust the value of the option before forwarding it to an upstream...
The Patience option is elective. Hence, a client MUST be prepared to receive a normal response even after the chosen Patience period (plus an allowance for network delays) has elapsed.

B.4.2. Leisure

Servers generally will compute an internal value that we will call Leisure, which indicates the period of time that will be used for responding to a request. A Patience option, if present, can be used as an upper bound for the Leisure. Leisure may be non-zero for congestion control reasons, in particular for responses to multicast requests. For these, the server should have a group size estimate G, a target 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 follows:

\[ \text{lb}_{\text{Leisure}} = \frac{S \times G}{R} \]

Figure 20: Computing a lower bound for the Leisure

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 8 kbit/s = 1 kB/s. The resulting lower bound for the Leisure is 10 seconds.

To avoid response implosion, responses to multicast requests SHOULD be dithered within a Leisure period chosen by the server to fall between these two bounds.

Currently, we don’t foresee a need to signal a value for Leisure from client to server (beyond the signalling provided by Patience) or from server to client, but an appropriate Option might be added later.

B.4.3. Pledge

In a basic observation relationship [I-D.ietf-core-observe], the server makes a pledge to keep the client in the observation relationship for a resource at least until the max-age for the resource is reached.

To save the client some effort in re-establishing observation relationships each time max-age is reached, the server MAY want to
extend its pledge beyond the end of max-age by signalling in a response/notification an additional time period using the Pledge Option, in parallel to the Observe Option.

The Pledge Option MUST NOT be used unless the server can make a reasonable promise not to lose the observation relationship in this time frame.

Currently, we don’t foresee a need to signal a value for Pledge from client to server, but an appropriate behavior might be added later for this option when sent in a request.

### B.4.4. Option Formats

<table>
<thead>
<tr>
<th>No.</th>
<th>C/E</th>
<th>Name</th>
<th>Format</th>
<th>Length</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>Elective</td>
<td>Patience</td>
<td>Duration in mis</td>
<td>1 B</td>
<td>(none)</td>
</tr>
<tr>
<td>24</td>
<td>Elective</td>
<td>Pledge</td>
<td>Duration in s</td>
<td>1 B</td>
<td>0</td>
</tr>
</tbody>
</table>

All timing options use the Duration data type (see Appendix D.2), however Patience (and Leisure, if that ever becomes an option) uses a timebase of milliseconds (mis = 1/1024 s) instead of seconds. (This reduces the range of the Duration from ~ 91 days to 128 minutes.)

Implementation note: As there are no strong accuracy requirements on the clocks employed, making use of any existing time base of milliseconds is a valid implementation approach (2.4 % off).

None of the options may be repeated.
Appendix C. The Cemetery (Things we won’t do)

This annex documents roads that the WG decided not to take, in order to spare readers from reinventing them in vain.

C.1. Example envelope option: solving #230

Ticket #230 [CoRE230] points out a design flaw of [I-D.ietf-core-coap]: When we split the elective Location option of draft -01 into multiple elective options, we made it possible that an implementation might process some of these and ignore others, leading to an incorrect interpretation of the Location expressed by the server.

There are several more or less savory solutions to #230.

Each of the elective options that together make up the Location could be defined in such a way that it makes a requirement on the processing of the related option (essentially revoking their elective status once the option under consideration is actually processed). This falls flat as soon as another option is defined that would also become part of the Location: existing implementations would not know that the new option is also part of the cluster that is re-interpreted as critical. The potential future addition of Location-Host and Location-Port makes this a valid consideration.

A better solution would be to define an elective Envelope Option called Location. Within a Location Option, the following top-level options might be allowed (now or in the future):

- Uri-Host
- Uri-Port
- Uri-Path
- Uri-Query

This would unify the code for interpreting the top-level request options that indicate the request URI with the code that interprets the Location URI.

The four options listed are all critical, while the envelope is elective. This gives exactly the desired semantics: If the envelope is processed at all (which is elective), the nested options are critical and all need to be processed.
C.2. Example envelope option: proxy-elective options

Another potential application of envelope options is motivated by the observation that new critical options might not be implemented by all proxies on the CoAP path to an origin server. So that this does not become an obstacle to introducing new critical options that are of interest only to client and origin server, the client might want to mark some critical options proxy-elective, i.e. elective for a proxy but still critical for the origin server.

One way to do this would be an Envelope option, the Proxy-Elective Option. A client might bundle a number of critical options into a critical Proxy-Elective Option. A proxy that processes the message is obliged to process the envelope (or reject the message), where processing means passing on the nested options towards the origin server (preferably again within a Proxy-Elective option). It can pass on the nested options, even ones unknown to the proxy, knowing that the client is happy with proxies not processing all of them.

(The assumption here is that the Proxy-Elective option becomes part of the base standard, so all but the most basic proxies would know how to handle it.)

C.3. Stateful URI compression

Is the approximately 25% average saving achievable with Huffman-based URI compression schemes worth the complexity? Probably not, because much higher average savings can be achieved by introducing state.

Henning Schulzrinne has proposed for a server to be able to supply a shortened URI once a resource has been requested using the full-length URI. Let’s call such a shortened referent a Temporary Resource Identifier, _TeRI_ for short. This could be expressed by a response option as shown in Figure 21.

```
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-
| duration | TeRI...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 21: Option for offering a TeRI in a response

The TeRI offer option indicates that the server promises to offer this resources under the TeRI given for at least the time given as the duration. Another TeRI offer can be made later to extend the duration.
Once a TeRI for a URI is known (and still within its lifetime), the client can supply a TeRI instead of a URI in its requests. The same option format as an offer could be used to allow the client to indicate how long it believes the TeRI will still be valid (so that the server can decide when to update the lifetime duration). TeRIs in requests could be distinguished from URIs e.g. by using a different option number.

Proposal: Add a TeRI option that can be used in CoAP requests and responses.

Add a way to indicate a TeRI and its duration in a link-value.

Do not add any form of stateless URI encoding.

Benefits: Much higher reduction of message size than any stateless URI encoding could achieve.

As the use of TeRIs is entirely optional, minimal complexity nodes can get by without implementing them.

Drawbacks: Adds considerable state and complexity to the protocol.

It turns out that real CoAP URIs are short enough that TeRIs are not needed.

(Discuss the security implications of TeRIs.)
Appendix D.  Experimental Options

This annex documents proposals that need significant additional discussion before they can become part of (or go back to) the main CoAP specification. They are not dead, but might die if there turns out to be no good way to solve the problem.

D.1. Options indicating absolute time

HTTP has a number of headers that may indicate absolute time:

- "Date", defined in Section 14.18 in [RFC2616] (Section 9.3 in [I-D.ietf-httpbis-p1-messaging]), giving the absolute time a response was generated;

- "Last-Modified", defined in Section 14.29 in [RFC2616], (Section 6.6 in [I-D.ietf-httpbis-p4-conditional], giving the absolute time of when the origin server believes the resource representation was last modified;

- "If-Modified-Since", defined in Section 14.25 in [RFC2616], "If-Unmodified-Since", defined in Section 14.28 in [RFC2616], and "If-Range", defined in Section 14.27 in [RFC2616] can be used to supply absolute time to gate a conditional request;

- "Expires", defined in Section 14.21 in [RFC2616] (Section 3.3 in [I-D.ietf-httpbis-p6-cache]), giving the absolute time after which a response is considered stale.

- The more obscure headers "Retry-After", defined in Section 14.37 in [RFC2616], and "Warning", defined in section 14.46 in [RFC2616], also may employ absolute time.

[I-D.ietf-core-coap] defines a single "Date" option, which however "indicates the creation time and date of a given resource representation", i.e., is closer to a "Last-Modified" HTTP header. HTTP’s caching rules [I-D.ietf-httpbis-p6-cache] make use of both "Date" and "Last-Modified", combined with "Expires". The specific semantics required for CoAP needs further consideration.

In addition to the definition of the semantics, an encoding for absolute times needs to be specified.

In UNIX-related systems, it is customary to indicate absolute time as an integer number of seconds, after midnight UTC, January 1, 1970. Unless negative numbers are employed, this time format cannot represent time values prior to January 1, 1970, which probably is not required for the uses of absolute time in CoAP.
If a 32-bit integer is used and allowance is made for a sign-bit in a local implementation, the latest UTC time value that can be represented by the resulting 31 bit integer value is 03:14:07 on January 19, 2038. If the 32-bit integer is used as an unsigned value, the last date is 2106-02-07, 06:28:15.

The reach can be extended by: - moving the epoch forward, e.g. by 40 years (= 1262304000 seconds) to 2010-01-01. This makes it impossible to represent Last-Modified times in that past (such as could be gatewayed in from HTTP). - extending the number of bits, e.g. by one more byte, either always or as one of two formats, keeping the 32-bit variant as well.

Also, the resolution can be extended by expressing time in milliseconds etc., requiring even more bits (e.g., a 48-bit unsigned integer of milliseconds would last well after year 9999.)

For experiments, an experimental "Date" option is defined with the semantics of HTTP’s "Last-Modified". It can carry an unsigned integer of 32, 40, or 48 bits; 32- and 40-bit integers indicate the absolute time in seconds since 1970-01-01 00:00 UTC, while 48-bit integers indicate the absolute time in milliseconds since 1970-01-01 00:00 UTC.

However, that option is not really that useful until there is a "If-Modified-Since" option as well.

(Also: Discuss nodes without clocks.)

**D.2. Representing Durations**

Various message types used in CoAP need the representation of *durations*, i.e. of the length of a timespan. In SI units, these are measured in seconds. CoAP durations represent integer numbers of seconds, but instead of representing these numbers as integers, a more compact single-byte pseudo-floating-point (pseudo-FP) representation is used (Figure 22).

```
  0   1   2   3   4   5   6   7
+---+---+---+---+---+---+---+---+
| 0...          value           |
+---+---+---+---+---+---+---+---+
| 1... mantissa |    exponent   |
+---+---+---+---+---+---+---+---+
```

Bormann & Hartke  Expires April 5, 2013  [Page 42]
Figure 22: Duration in (8,4) pseudo-FP representation

If the high bit is clear, the entire n-bit value (including the high bit) is the decoded value. If the high bit is set, the mantissa (including the high bit, with the exponent field cleared out but still present) is shifted left by the exponent to yield the decoded value.

The (n,e)-pseudo-FP format can be decoded with a single line of code (plus a couple of constant definitions), as demonstrated in Figure 23.

```c
#define N 8
#define E 4
#define HIBIT (1 << (N - 1))
#define EMASK ((1 << E) - 1)
#define MMASK ((1 << N) - 1 - EMASK)

#define DECODE_8_4(r) (r < HIBIT ? r : (r & MMASK) << (r & EMASK))
```

Figure 23: Decoding an (8,4) pseudo-FP value

Note that a pseudo-FP encoder needs to consider rounding; different applications of durations may favor rounding up or rounding down the value encoded in the message.

The highest pseudo-FP value, represented by an all-ones byte (0xFF), is reserved to indicate an indefinite duration. The next lower value (0xEF) is thus the highest representable value and is decoded as 7340032 seconds, a little more than 12 weeks.

D.3. Rationale

Where CPU power and memory is abundant, a duration can almost always be adequately represented by a non-negative floating-point number representing that number of seconds. Historically, many APIs have also used an integer representation, which limits both the resolution (e.g., if the integer represents the duration in seconds) and often the range (integer machine types have range limits that may become relevant). UNIX’s "time_t" (which is used for both absolute time and durations) originally was a signed 32-bit value of seconds, but was later complemented by an additional integer to add microsecond ("struct timeval") and then later nanosecond ("struct timespec") resolution.

Three decisions need to be made for each application of the concept of duration:
o the *resolution*. What rounding error is acceptable?

- the *range*. What is the maximum duration that needs to be represented?

- the *number of bits* that can be expended.

Obviously, these decisions are interrelated. Typically, a large range needs a large number of bits, unless resolution is traded. For most applications, the actual requirement for resolution are limited for longer durations, but can be more acute for shorter durations.

### D.4. Pseudo-Floating Point

Constrained systems typically avoid the use of floating-point (FP) values, as

- simple CPUs often don’t have support for floating-point datatypes
- software floating-point libraries are expensive in code size and slow.

In addition, floating-point datatypes used to be a significant element of market differentiation in CPU design; it has taken the industry a long time to agree on a standard floating point representation.

These issues have led to protocols that try to constrain themselves to integer representation even where the ability of a floating point representation to trade range for resolution would be beneficial.

The idea of introducing _pseudo-FP_ is to obtain the increased range provided by embedding an exponent, without necessarily getting stuck with hardware datatypes or inefficient software floating-point libraries.

For the purposes of this draft, we define an \((n,e)\)-pseudo-FP as a fixed-length value of \(n\) bits, \(e\) of which may be used for an exponent. Figure 22 illustrates an \((8,4)\)-pseudo-FP value.

If the high bit is clear, the entire \(n\)-bit value (including the high bit) is the decoded value. If the high bit is set, the mantissa (including the high bit, but with the exponent field cleared out) is shifted left by the exponent to yield the decoded value.

The \((n,e)\)-pseudo-FP format can be decoded with a single line of code (plus a couple of constant definition), as demonstrated in Figure 23.
Only non-negative numbers can be represented by this format. It is designed to provide full integer resolution for values from 0 to $2^{n-1}-1$, i.e., 0 to 127 in the (8,4) case, and a mantissa of $n-e$ bits from $2^{n-1}$ to $(2^n-2^e)*2^{2^e-1}$, i.e., 128 to 7864320 in the (8,4) case. By choosing $e$ carefully, resolution can be traded against range.

Note that a pseudo-FP encoder needs to consider rounding; different applications of durations may favor rounding up or rounding down the value encoded in the message. This requires a little more than a single line of code (which is left as an exercise to the reader, as the most efficient expression depends on hardware details).

D.5. A Duration Type for CoAP

CoAP needs durations in a number of places. In [I-D.ietf-core-coap], durations occur in the option "Subscription-lifetime" as well as in the option "Max-age". (Note that the option "Date" is not a duration, but a point in time.) Other durations of this kind may be added later.

Most durations relevant to CoAP are best expressed with a minimum resolution of one second. More detailed resolutions are unlikely to provide much benefit.

The range of lifetimes and caching ages are probably best kept below the order of magnitude of months. An (8,4)-pseudo-FP has the maximum value of 7864320, which is about 91 days; this appears to be adequate for a subscription lifetime and probably even for a maximum cache age. Figure 24 shows the values that can be expressed. (If a larger range for the latter is indeed desired, an (8,5)-pseudo-FP could be used; this would last 15 milleniums, at the cost of having only 3 bits of accuracy for values larger than 127 seconds.)

Proposal: A single duration type is used throughout CoAP, based on an (8,4)-pseudo-FP giving a duration in seconds.

Benefits: Implementations can use a single piece of code for managing all CoAP-related durations.

In addition, length information never needs to be managed for durations that are embedded in other data structures: All durations are expressed by a single byte.

It might be worthwhile to reserve one duration value, e.g. 0xFF, for an indefinite duration.

<table>
<thead>
<tr>
<th>Duration</th>
<th>Seconds</th>
<th>Encoded</th>
</tr>
</thead>
</table>

Bormann & Hartke Expires April 5, 2013 [Page 45]
22d 18:08:00 0x001e0000 0xfd
24d 06:32:32 0x00200000 0x8e
27d 07:21:36 0x00240000 0x9e
30d 08:10:40 0x00280000 0xae
33d 08:59:44 0x002c0000 0xbe
36d 09:48:48 0x00300000 0xce
39d 10:37:52 0x00340000 0xde
42d 11:26:56 0x00380000 0xe6
45d 12:16:00 0x003c0000 0xfe
48d 13:05:04 0x00400000 0x8f
54d 14:43:12 0x00480000 0x9f
60d 16:21:20 0x00500000 0xaf
66d 17:59:28 0x00580000 0xbf
72d 19:37:36 0x00600000 0xcf
78d 21:15:44 0x00680000 0xdf
84d 22:53:52 0x00700000 0xef
91d 00:32:00 0x00780000 0xff (reserved)

Figure 24
Authors’ Addresses

Carsten Bormann
Universitaet Bremen TZI
Postfach 330440
Bremen  D-28359
Germany
Phone: +49-421-218-63921
Email: cabo@tzi.org

Klaus Hartke
Universitaet Bremen TZI
Postfach 330440
Bremen  D-28359
Germany
Phone: +49-421-218-63905
Email: hartke@tzi.org