   Block-Wise Transfers in the Constrained Application Protocol (CoAP)

Abstract

   The Constrained Application Protocol (CoAP) is a RESTful transfer
   protocol for constrained nodes and networks. Basic CoAP messages
   work well for small payloads from sensors and actuators; however,
   applications will need to transfer larger payloads occasionally --
   for instance, for firmware updates. In contrast to HTTP, where TCP
   does the grunt work of segmenting and resequencing, CoAP is based on
   datagram transports such as UDP or Datagram Transport Layer Security
   (DTLS). These transports only offer fragmentation, which is even
   more problematic in constrained nodes and networks, limiting the
   maximum size of resource representations that can practically be
   transferred.

   Instead of relying on IP fragmentation, this specification extends
   basic CoAP with a pair of "Block" options for transferring multiple
   blocks of information from a resource representation in multiple
   request-response pairs. In many important cases, the Block options
   enable a server to be truly stateless: the server can handle each
   block transfer separately, with no need for a connection setup or
   other server-side memory of previous block transfers. Essentially,
   the Block options provide a minimal way to transfer larger
   representations in a block-wise fashion.

   A CoAP implementation that does not support these options generally
   is limited in the size of the representations that can be exchanged,
   so there is an expectation that the Block options will be widely used
   in CoAP implementations. Therefore, this specification updates
   RFC 7252.

Status of This Memo

This is an Internet Standards Track document.

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

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc7959.

Copyright Notice

Table of Contents

1. Introduction

   The work on Constrained RESTful Environments (CoRE) aims at realizing
   the Representational State Transfer (REST) architecture in a suitable
   form for the most constrained nodes (such as microcontrollers with
   limited RAM and ROM [RFC7228]) and networks (such as IPv6 over Low-
   Power Wireless Personal Area Networks (6LoWPANs) [RFC4944])
   [RFC7252]. The CoAP protocol is intended to provide RESTful [REST]
   services not unlike HTTP [RFC7230], 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 highly
   constrained nodes.

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

   o reducing implementation complexity in order to minimize code size,

   o reducing message sizes in order to minimize the number of
      fragments needed for each message (to maximize the probability of
      delivery of the message), the amount of transmission power needed,
      and the loading of the limited-bandwidth channel,

   o reducing requirements on the environment such as stable storage,
      good sources of randomness, or user-interaction capabilities.

   Because CoAP is based on datagram transports such as UDP or Datagram
   Transport Layer Security (DTLS), the maximum size of resource
   representations that can be transferred without too much
   fragmentation is limited. In addition, not all resource
   representations will fit into a single link-layer packet of a
   constrained network, which may cause adaptation layer fragmentation
   even if IP-layer fragmentation is not required. Using fragmentation
   (either at the adaptation layer or at the IP layer) for the transport
   of larger representations would be possible up to the maximum size of
   the underlying datagram protocol (such as UDP), but the
   fragmentation/reassembly process burdens the lower layers with
   conversation state that is better managed in the application layer.

   The present specification defines a pair of CoAP options to enable
   block-wise access to resource representations. The Block options
   provide a minimal way to transfer larger resource representations in
   a block-wise fashion. The overriding objective is to avoid the need
   for creating conversation state at the server for block-wise GET
   requests. (It is impossible to fully avoid creating conversation
   state for POST/PUT, if the creation/replacement of resources is to be
   atomic; where that property is not needed, there is no need to create
   server conversation state in this case, either.)

Block-wise transfers are realized as combinations of exchanges, each
of which is performed according to the CoAP base protocol [RFC7252].
Each exchange in such a combination is governed by the specifications
in [RFC7252], including the congestion control specifications
(Section 4.7 of [RFC7252]) and the security considerations
(Section 11 of [RFC7252]; additional security considerations then
apply to the transfers as a whole, see Section 7). The present
specification minimizes the constraints it adds to those base
exchanges; however, not all variants of using CoAP are very useful
inside a block-wise transfer (e.g., using Non-confirmable requests
within block-wise transfers outside the use case of Section 2.8 would
escalate the overall non-delivery probability). To be perfectly
clear, the present specification also does not remove any of the
constraints posed by the base specification it is strictly layered on
top of. For example, back-to-back packets are limited by the
congestion control described in Section 4.7 of [RFC7252] (NSTART as a
limit for initiating exchanges, PROBING_RATE as a limit for sending
with no response); block-wise transfers cannot send/solicit more
traffic than a client could be sending to / soliciting from the same
server without the block-wise mode.

In some cases, the present specification will RECOMMEND that a client
perform a sequence of block-wise transfers "without undue delay".
This cannot be phrased as an interoperability requirement, but is an
expectation on implementation quality. Conversely, the expectation
is that servers will not have to go out of their way to accommodate
clients that take considerable time to finish a block-wise transfer.
For example, for a block-wise GET, if the resource changes while this
proceeds, the entity-tag (ETag) for a further block obtained may be
different. To avoid this happening all the time for a fast-changing
resource, a server MAY try to keep a cache around for a specific
client for a short amount of time. The expectation here is that the
lifetime for such a cache can be kept short, on the order of a few
expected round-trip times, counting from the previous block
transferred.

In summary, this specification adds a pair of Block options to CoAP
that can be used for block-wise transfers. Benefits of using these
options include:

o Transfers larger than what can be accommodated in constrained-
   network link-layer packets can be performed in smaller blocks.

o No hard-to-manage conversation state is created at the adaptation
   layer or IP layer for fragmentation.

o The transfer of each block is acknowledged, enabling individual
   retransmission if required.

o Both sides have a say in the block size that actually will be
   used.

o The resulting exchanges are easy to understand using packet
   analyzer tools, and thus quite accessible to debugging.

o If needed, the Block options can also be used (without changes) to
   provide random access to power-of-two sized blocks within a
   resource representation.

A CoAP implementation that does not support these options generally
is limited in the size of the representations that can be exchanged,
see Section 4.6 of [RFC7252]. Even though the options are Critical,
a server may decide to start using them in an unsolicited way in a
response. No effort was expended to provide a capability indication
mechanism supporting that decision: since the block-wise transfer
mechanisms are so fundamental to the use of CoAP for representations
larger than about a kilobyte, there is an expectation that they are
very widely implemented.

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 RFC
2119, BCP 14 [RFC2119] and indicate requirement levels for compliant
CoAP implementations.

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

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

2. Block-Wise Transfers

As discussed in the introduction, there are good reasons to limit the
size of datagrams in constrained networks:

o by the maximum datagram size (~ 64 KiB for UDP)

o by the desire to avoid IP fragmentation (MTU of 1280 for IPv6)

o by the desire to avoid adaptation-layer fragmentation (60-80 bytes
   for 6LoWPAN [RFC4919])

When a resource representation is larger than can be comfortably
transferred in the payload of a single CoAP datagram, a Block option
can be used to indicate a block-wise transfer. As payloads can be

sent both with requests and with responses, this specification
provides two separate options for each direction of payload transfer.
In naming these options (for block-wise transfers as well as in
Section 4), we use the number 1 ("Block1", "Size1") to refer to the
transfer of the resource representation that pertains to the request,
and the number 2 ("Block2", "Size2") to refer to the transfer of the
resource representation for the response.

In the following, the term "payload" will be used for the actual
content of a single CoAP message, i.e., a single block being
transferred, while the term "body" will be used for the entire
resource representation that is being transferred in a block-wise
fashion. The Content-Format Option applies to the body, not to the
payload; in particular, the boundaries between the blocks may be in
places that are not separating whole units in terms of the structure,
encoding, or content-coding used by the Content-Format. (Similarly,
the ETag Option defined in Section 5.10.6 of [RFC7252] applies to the
whole representation of the resource, and thus to the body of the
response.)

In most cases, all blocks being transferred for a body (except for
the last one) will be of the same size. (If the first request uses a
bigger block size than the receiver prefers, subsequent requests will
use the preferred block size.) The block size is not fixed by the
protocol. To keep the implementation as simple as possible, the
Block options support only a small range of power-of-two block sizes,
from $2^{*}{*}4$ (16) to $2^{*}{*}10$ (1024) bytes. As bodies often will not
evenly divide into the power-of-two block size chosen, the size need
not be reached in the final block (but even for the final block, the
chosen power-of-two size will still be indicated in the block size
field of the Block option).

2.1. The Block2 and Block1 Options

No.	C	U	N	R	Name	Format	Length	Default
23	C	U	-	-	Block2	uint	0-3	(none)
27	C	U	-	-	Block1	uint	0-3	(none)

Table 1: Block Option Numbers

Both Block1 and Block2 Options can be present in both the request and
response messages. In either case, the Block1 Option pertains to the
request payload, and the Block2 Option pertains to the response
payload.

Hence, for the methods defined in [RFC7252], Block1 is useful with
the payload-bearing POST and PUT requests and their responses.
Block2 is useful with GET, POST, and PUT requests and their payload-
bearing responses (2.01, 2.02, 2.04, and 2.05 -- see Section 5.5 of
[RFC7252]).

Where Block1 is present in a request or Block2 in a response (i.e.,
in that message to the payload of which it pertains) it indicates a
block-wise transfer and describes how this specific block-wise
payload forms part of the entire body being transferred ("descriptive
usage"). Where it is present in the opposite direction, it provides
additional control on how that payload will be formed or was
processed ("control usage").

Implementation of either Block option is intended to be optional.
However, when it is present in a CoAP message, it MUST be processed
(or the message rejected); therefore, it is identified as a Critical
option. Either Block option MUST NOT occur more than once in a
single message.

2.2. Structure of a Block Option

Three items of information may need to be transferred in a Block
(Block1 or Block2) option:

o the size of the block (SZX);

o whether more blocks are following (M);

o the relative number of the block (NUM) within a sequence of blocks
   with the given size.

The value of the Block option is a variable-size (0 to 3 byte)
unsigned integer (uint, see Section 3.2 of [RFC7252]). This integer
value encodes these three fields, see Figure 1. (Due to the CoAP
uint-encoding rules, when all of NUM, M, and SZX happen to be zero, a
zero-byte integer will be sent.)

```
         0
         0 1 2 3 4 5 6 7
        +-+-+-+-+-+-+-+-+
        |  NUM  |M| SZX |
        +-+-+-+-+-+-+-+-+

         0                   1
         0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |          NUM          |M| SZX |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

         0                   1                   2
         0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
        |                  NUM                  |M| SZX |
        +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```
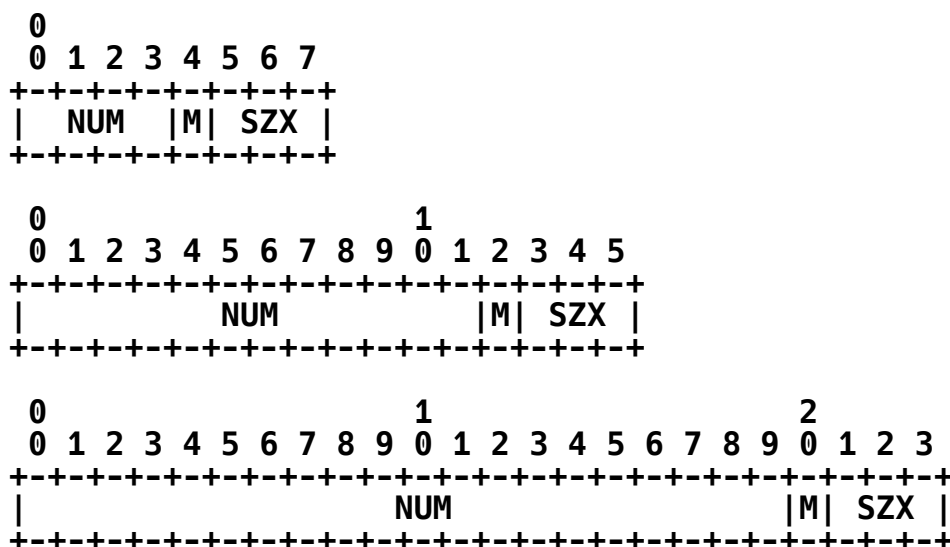
                    Figure 1: Block Option Value

   The block size is encoded using a three-bit unsigned integer (0 for
   2**4 bytes to 6 for 2**10 bytes), which we call the "SZX" ("size
   exponent"); the actual block size is then "2**(SZX + 4)". SZX is
   transferred in the three least significant bits of the option value
   (i.e., "val & 7" where "val" is the value of the option).

   The fourth least significant bit, the M or "more" bit ("val & 8"),
   indicates whether more blocks are following or if the current block-
   wise transfer is the last block being transferred.

   The option value divided by sixteen (the NUM field) is the sequence
   number of the block currently being transferred, starting from zero.
   The current transfer is, therefore, about the "size" bytes starting
   at byte "NUM << (SZX + 4)".

   Implementation note: As an implementation convenience, "(val & ~0xF)
      << (val & 7)", i.e., the option value with the last 4 bits masked
      out, shifted to the left by the value of SZX, gives the byte
      position of the first byte of the block being transferred.

   More specifically, within the option value of a Block1 or Block2
   Option, the meaning of the option fields is defined as follows:

   NUM: Block Number, indicating the block number being requested or
      provided. Block number 0 indicates the first block of a body
      (i.e., starting with the first byte of the body).

M: More Flag ("not last block"). For descriptive usage, this flag,
   if unset, indicates that the payload in this message is the last
   block in the body; when set, it indicates that there are one or
   more additional blocks available. When a Block2 Option is used in
   a request to retrieve a specific block number ("control usage"),
   the M bit MUST be sent as zero and ignored on reception. (In a
   Block1 Option in a response, the M flag is used to indicate
   atomicity, see below.)

SZX: Block Size. The block size is represented as a three-bit
   unsigned integer indicating the size of a block to the power of
   two. Thus, block size = $2**(SZX + 4)$. The allowed values of SZX
   are 0 to 6, i.e., the minimum block size is $2**(0+4) = 16$ and the
   maximum is $2**(6+4) = 1024$. The value 7 for SZX (which would
   indicate a block size of 2048) is reserved, i.e., MUST NOT be sent
   and MUST lead to a 4.00 Bad Request response code upon reception
   in a request.

There is no default value for the Block1 and Block2 Options. Absence
of one of these options is equivalent to an option value of 0 with
respect to the value of NUM and M that could be given in the option,
i.e., it indicates that the current block is the first and only block
of the transfer (block number 0, M bit not set). However, in
contrast to the explicit value 0, which would indicate an SZX of 0
and thus a size value of 16 bytes, there is no specific explicit size
implied by the absence of the option -- the size is left unspecified.
(As for any uint, the explicit value 0 is efficiently indicated by a
zero-length option; this, therefore, is different in semantics from
the absence of the option.)

2.3. Block Options in Requests and Responses

The Block options are used in one of three roles:

o In descriptive usage, i.e., a Block2 Option in a response (such as
   a 2.05 response for GET), or a Block1 Option in a request (a PUT
   or POST):

   * The NUM field in the option value describes what block number
      is contained in the payload of this message.

   * The M bit indicates whether further blocks need to be
      transferred to complete the transfer of that body.

   * The block size implied by SZX MUST match the size of the
      payload in bytes, if the M bit is set. (SZX does not govern
      the payload size if M is unset). For Block2, if the request
      suggested a larger value of SZX, the next request MUST move SZX

down to the size given in the response. (The effect is that,
if the server uses the smaller of (1) its preferred block size
and (2) the block size requested, all blocks for a body use the
same block size.)

o A Block2 Option in control usage in a request (e.g., GET):

   * The NUM field in the Block2 Option gives the block number of
      the payload that is being requested to be returned in the
      response.

   * In this case, the M bit has no function and MUST be set to
      zero.

   * The block size given (SZX) suggests a block size (in the case
      of block number 0) or repeats the block size of previous blocks
      received (in the case of a non-zero block number).

o A Block1 Option in control usage in a response (e.g., a 2.xx
   response for a PUT or POST request):

   * The NUM field of the Block1 Option indicates what block number
      is being acknowledged.

   * If the M bit was set in the request, the server can choose
      whether to act on each block separately, with no memory, or
      whether to handle the request for the entire body atomically,
      or any mix of the two.

      + If the M bit is also set in the response, it indicates that
         this response does not carry the final response code to the
         request, i.e., the server collects further blocks from the
         same endpoint and plans to implement the request atomically
         (e.g., acts only upon reception of the last block of
         payload). In this case, the response MUST NOT carry a
         Block2 Option.

      + Conversely, if the M bit is unset even though it was set in
         the request, it indicates the block-wise request was enacted
         now specifically for this block, and the response carries
         the final response to this request (and to any previous ones
         with the M bit set in the response's Block1 Option in this
         sequence of block-wise transfers); the client is still
         expected to continue sending further blocks, the request
         method for which may or may not also be enacted per-block.
         (Note that the resource is now in a partially updated state;
         this approach is only appropriate where exposing such an

       intermediate state is acceptable. The client can reduce the
       window by quickly continuing to update the resource, or, in
       case of failure, restarting the update.)

   * Finally, the SZX block size given in a control Block1 Option
      indicates the largest block size preferred by the server for
      transfers toward the resource that is the same or smaller than
      the one used in the initial exchange; the client SHOULD use
      this block size or a smaller one in all further requests in the
      transfer sequence, even if that means changing the block size
      (and possibly scaling the block number accordingly) from now
      on.

   Using one or both Block options, a single REST operation can be split
   into multiple CoAP message exchanges. As specified in [RFC7252],
   each of these message exchanges uses their own CoAP Message ID.

   The Content-Format Option sent with the requests or responses MUST
   reflect the Content-Format of the entire body. If blocks of a
   response body arrive with different Content-Format Options, it is up
   to the client how to handle this error (it will typically abort any
   ongoing block-wise transfer). If blocks of a request arrive at a
   server with mismatching Content-Format Options, the server MUST NOT
   assemble them into a single request; this usually leads to a 4.08
   (Request Entity Incomplete, Section 2.9.2) error response on the
   mismatching block.

2.4. Using the Block2 Option

   When a request is answered with a response carrying a Block2 Option
   with the M bit set, the requester may retrieve additional blocks of
   the resource representation by sending further requests with the same
   options as the initial request and a Block2 Option giving the block
   number and block size desired. In a request, the client MUST set the
   M bit of a Block2 Option to zero and the server MUST ignore it on
   reception.

   To influence the block size used in a response, the requester MAY
   also use the Block2 Option on the initial request, giving the desired
   size, a block number of zero and an M bit of zero. A server MUST use
   the block size indicated or a smaller size. Any further block-wise
   requests for blocks beyond the first one MUST indicate the same block
   size that was used by the server in the response for the first
   request that gave a desired size using a Block2 Option.

   Once the Block2 Option is used by the requester and a first response
   has been received with a possibly adjusted block size, all further
   requests in a single block-wise transfer will ultimately converge on

using the same size, except that there may not be enough content to
fill the last block (the one returned with the M bit not set). (Note
that the client may start using the Block2 Option in a second request
after a first request without a Block2 Option resulted in a Block2
Option in the response.) The server uses the block size indicated in
the request option or a smaller size, but the requester MUST take
note of the actual block size used in the response it receives to its
initial request and proceed to use it in subsequent requests. The
server behavior MUST ensure that this client behavior results in the
same block size for all responses in a sequence (except for the last
one with the M bit not set, and possibly the first one if the initial
request did not contain a Block2 Option).

Block-wise transfers can be used to GET resources whose
representations are entirely static (not changing over time at all,
such as in a schema describing a device), or for dynamically changing
resources. In the latter case, the Block2 Option SHOULD be used in
conjunction with the ETag Option ([RFC7252], Section 5.10.6), to
ensure that the blocks being reassembled are from the same version of
the representation: The server SHOULD include an ETag Option in each
response. If an ETag Option is available, the client, when
reassembling the representation from the blocks being exchanged, MUST
compare ETag Options. If the ETag Options do not match in a GET
transfer, the requester has the option of attempting to retrieve
fresh values for the blocks it retrieved first. To minimize the
resulting inefficiency, the server MAY cache the current value of a
representation for an ongoing sequence of requests. (The server may
identify the sequence by the combination of the requesting endpoint
and the URI being the same in each block-wise request.) Note well
that this specification makes no requirement for the server to
establish any state; however, servers that offer quickly changing
resources may thereby make it impossible for a client to ever
retrieve a consistent set of blocks. Clients that want to retrieve
all blocks of a resource SHOULD strive to do so without undue delay.
Servers can fully expect to be free to discard any cached state after
a period of EXCHANGE_LIFETIME ([RFC7252], Section 4.8.2) after the
last access to the state, however, there is no requirement to always
keep the state for as long.

The Block2 Option provides no way for a single endpoint to perform
multiple concurrently proceeding block-wise response payload transfer
(e.g., GET) operations to the same resource. This is rarely a
requirement, but as a workaround, a client may vary the cache key
(e.g., by using one of several URIs accessing resources with the same
semantics, or by varying a proxy-safe elective option).

2.5. Using the Block1 Option

In a request with a request payload (e.g., PUT or POST), the Block1
Option refers to the payload in the request (descriptive usage).

In response to a request with a payload (e.g., a PUT or POST
transfer), the block size given in the Block1 Option indicates the
block size preference of the server for this resource (control
usage). Obviously, at this point the first block has already been
transferred by the client without benefit of this knowledge. Still,
the client SHOULD heed the preference indicated and, for all further
blocks, use the block size preferred by the server or a smaller one.
Note that any reduction in the block size may mean that the second
request starts with a block number larger than one, as the first
request already transferred multiple blocks as counted in the smaller
size.

To counter the effects of adaptation-layer fragmentation on packet-
delivery probability, a client may want to give up retransmitting a
request with a relatively large payload even before MAX_RETRANSMIT
has been reached, and try restating the request as a block-wise
transfer with a smaller payload. Note that this new attempt is then
a new message-layer transaction and requires a new Message ID.
(Because of the uncertainty about whether the request or the
acknowledgement was lost, this strategy is useful mostly for
idempotent requests.)

In a block-wise transfer of a request payload (e.g., a PUT or POST)
that is intended to be implemented in an atomic fashion at the
server, the actual creation/replacement takes place at the time the
final block, i.e., a block with the M bit unset in the Block1 Option,
is received. In this case, all success responses to non-final blocks
carry the response code 2.31 (Continue, Section 2.9.1). If not all
previous blocks are available at the server at the time of processing
the final block, the transfer fails and error code 4.08 (Request
Entity Incomplete, Section 2.9.2) MUST be returned. A server MAY
also return a 4.08 error code for any (final or non-final) Block1
transfer that is not in sequence; therefore, clients that do not have
specific mechanisms to handle this case SHOULD always start with
block zero and send the following blocks in order.

One reason that a client might encounter a 4.08 error code is that
the server has already timed out and discarded the partial request
body being assembled. Clients SHOULD strive to send all blocks of a
request without undue delay. Servers can fully expect to be free to
discard any partial request body when a period of EXCHANGE_LIFETIME

([RFC7252], Section 4.8.2) has elapsed after the most recent block was transferred; however, there is no requirement on a server to always keep the partial request body for as long.

The error code 4.13 (Request Entity Too Large) can be returned at any time by a server that does not currently have the resources to store blocks for a block-wise request payload transfer that it would intend to implement in an atomic fashion. (Note that a 4.13 response to a request that does not employ Block1 is a hint for the client to try sending Block1, and a 4.13 response with a smaller SZX in its Block1 Option than requested is a hint to try a smaller SZX.)

A block-wise transfer of a request payload that is implemented in a stateless fashion at the server is likely to leave the resource being operated on in an inconsistent state while the transfer is still ongoing or when the client does not complete the transfer. This characteristic is closer to that of remote file systems than to that of HTTP, where state is always kept on the server during a transfer. Techniques well known from shared file access (e.g., client-specific temporary resources) can be used to mitigate this difference from HTTP.

The Block1 Option provides no way for a single endpoint to perform multiple concurrently proceeding block-wise request payload transfer (e.g., PUT or POST) operations to the same resource. Starting a new block-wise sequence of requests to the same resource (before an old sequence from the same endpoint was finished) simply overwrites the context the server may still be keeping. (This is probably exactly what one wants in this case -- the client may simply have restarted and lost its knowledge of the previous sequence.)

2.6. Combining Block-Wise Transfers with the Observe Option

The Observe option provides a way for a client to be notified about changes over time of a resource [RFC7641]. Resources observed by clients may be larger than can be comfortably processed or transferred in one CoAP message. The following rules apply to the combination of block-wise transfers with notifications.

Observation relationships always apply to an entire resource; the Block2 Option does not provide a way to observe a single block of a resource.

As with basic GET transfers, the client can indicate its desired block size in a Block2 Option in the GET request establishing or renewing the observation relationship. If the server supports block-wise transfers, it SHOULD take note of the block size and apply it as a maximum size to all notifications/responses resulting from the GET

request (until the client is removed from the list of observers or
the entry in that list is updated by the server receiving a new GET
request for the resource from the client).

When sending a 2.05 (Content) notification, the server only sends the
first block of the representation. The client retrieves the rest of
the representation as if it had caused this first response by a GET
request, i.e., by using additional GET requests with Block2 Options
containing NUM values greater than zero. (This results in the
transfer of the entire representation, even if only some of the
blocks have changed with respect to a previous notification.)

As with other dynamically changing resources, to ensure that the
blocks being reassembled are from the same version of the
representation, the server SHOULD include an ETag Option in each
response, and the reassembling client MUST compare the ETag Options
(Section 2.4). Even more so than for the general case of Block2,
clients that want to retrieve all blocks of a resource they have been
notified about with a first block SHOULD strive to do so without
undue delay.

See Section 3.4 for examples.

2.7. Combining Block1 and Block2

In PUT and particularly in POST exchanges, both the request body and
the response body may be large enough to require the use of block-
wise transfers. First, the Block1 transfer of the request body
proceeds as usual. In the exchange of the last slice of this block-
wise transfer, the response carries the first slice of the Block2
transfer (NUM is zero). To continue this Block2 transfer, the client
continues to send requests similar to the requests in the Block1
phase, but leaves out the Block1 Options and includes a Block2
request option with non-zero NUM.

Block2 transfers that retrieve the response body for a request that
used Block1 MUST be performed in sequential order.

2.8. Combining Block2 with Multicast

A client can use the Block2 Option in a multicast GET request with
NUM = 0 to aid in limiting the size of the response.

Similarly, a response to a multicast GET request can use a Block2
Option with NUM = 0 if the representation is large, or to further
limit the size of the response.

In both cases, the client retrieves any further blocks using unicast exchanges; in the unicast requests, the client SHOULD heed any block size preferences indicated by the server in the response to the multicast request.

Other uses of the Block options in conjunction with multicast messages are for further study.

2.9. Response Codes

Beyond the response codes defined in [RFC7252], this specification defines two response codes and extends the meaning of one.

2.9.1. 2.31 Continue

This new success status code indicates that the transfer of this block of the request body was successful and that the server encourages sending further blocks, but that a final outcome of the whole block-wise request cannot yet be determined. No payload is returned with this response code.

2.9.2. 4.08 Request Entity Incomplete

This new client error status code indicates that the server has not received the blocks of the request body that it needs to proceed. The client has not sent all blocks, not sent them in the order required by the server, or has sent them long enough ago that the server has already discarded them.

(Note that one reason for not having the necessary blocks at hand may be a Content-Format mismatch, see Section 2.3. Implementation note: A server can reject a Block1 transfer with this code when NUM != 0 and a different Content-Format is indicated than expected from the current state of the resource. If it implements the transfer in a stateless fashion, it can match up the Content-Format of the block against that of the existing resource. If it implements the transfer in an atomic fashion, it can match up the block against the partially reassembled piece of representation that is going to replace the state of the resource.)

2.9.3. 4.13 Request Entity Too Large

In Section 5.9.2.9 of [RFC7252], the response code 4.13 (Request Entity Too Large) is defined to be like HTTP 413 "Request Entity Too Large". [RFC7252] also recommends that this response SHOULD include a Size1 Option (Section 4) to indicate the maximum size of request entity the server is able and willing to handle, unless the server is not in a position to make this information available.

The present specification allows the server to return this response
code at any time during a Block1 transfer to indicate that it does
not currently have the resources to store blocks for a transfer that
it would intend to implement in an atomic fashion. It also allows
the server to return a 4.13 response to a request that does not
employ Block1 as a hint for the client to try sending Block1.
Finally, a 4.13 response to a request with a Block1 Option (control
usage, see Section 2.3) where the response carries a smaller SZX in
its Block1 Option is a hint to try that smaller SZX.

2.10. Caching Considerations

This specification attempts to leave a variety of implementation
strategies open for caches, in particular those in caching proxies.
For example, a cache is free to cache blocks individually, but also
could wait to obtain the complete representation before it serves
parts of it. Partial caching may be more efficient in a cross-proxy
(equivalent to a streaming HTTP proxy). A cached block (partial
cached response) can be used in place of a complete response to
satisfy a block-wise request that is presented to a cache. Note that
different blocks can have different Max-Age values, as they are
transferred at different times. A response with a block updates the
freshness of the complete representation. Individual blocks can be
validated, and validating a single block validates the complete
representation. A response with a Block1 Option in control usage
with the M bit set invalidates cached responses for the target URI.

A cache or proxy that combines responses (e.g., to split blocks in a
request or increase the block size in a response, or a cross-proxy)
may need to combine 2.31 and 2.01/2.04 responses; a stateless server
may be responding with 2.01 only on the first Block1 block
transferred, which dominates any 2.04 responses for later blocks.

If-None-Match only works correctly on Block1 requests with (NUM=0)
and MUST NOT be used on Block1 requests with NUM != 0.

3. Examples

This section gives a number of short examples with message flows for
a block-wise GET, and for a PUT or POST. These examples demonstrate
the basic operation, the operation in the presence of
retransmissions, and examples for the operation of the block size
negotiation.

In all these examples, a Block option is shown in a decomposed way
indicating the kind of Block option (1 or 2) followed by a colon, and
then the block number (NUM), more bit (M), and block size exponent
(2**(SZX+4)) separated by slashes. For example, a Block2 Option
value of 33 would be shown as 2:2/0/32) and a Block1 Option value of
59 would be shown as 1:3/1/128.

As in [RFC7252], "MID" is used as an abbreviation for "Message ID".

3.1. Block2 Examples

The first example (Figure 2) shows a GET request that is split into
three blocks. The server proposes a block size of 128, and the
client agrees. The first two ACKs contain a payload of 128 bytes
each, and the third ACK contains a payload between 1 and 128 bytes.

```
CLIENT                                                       SERVER
  |                                                            |
  | CON [MID=1234], GET, /status                       ------> |
  |                                                            |
  | <------   ACK [MID=1234], 2.05 Content, 2:0/1/128          |
  |                                                            |
  | CON [MID=1235], GET, /status, 2:1/0/128            ------> |
  |                                                            |
  | <------   ACK [MID=1235], 2.05 Content, 2:1/1/128          |
  |                                                            |
  | CON [MID=1236], GET, /status, 2:2/0/128            ------> |
  |                                                            |
  | <------   ACK [MID=1236], 2.05 Content, 2:2/0/128          |
```

Figure 2: Simple Block-Wise GET

In the second example (Figure 3), the client anticipates the block-
wise transfer (e.g., because of a size indication in the link-format
description [RFC6690]) and sends a block size proposal. All ACK
messages except for the last carry 64 bytes of payload; the last one
carries between 1 and 64 bytes.

```
CLIENT                                                       SERVER
  |                                                            |
  |  CON [MID=1234], GET, /status, 2:0/0/64         ------>    |
  |                                                            |
  |  <------     ACK [MID=1234], 2.05 Content, 2:0/1/64        |
  |                                                            |
  |  CON [MID=1235], GET, /status, 2:1/0/64         ------>    |
  |                                                            |
  |  <------     ACK [MID=1235], 2.05 Content, 2:1/1/64        |
  :                                                            :
  :                            ...                             :
  :                                                            :
  |  CON [MID=1238], GET, /status, 2:4/0/64         ------>    |
  |                                                            |
  |  <------     ACK [MID=1238], 2.05 Content, 2:4/1/64        |
  |                                                            |
  |  CON [MID=1239], GET, /status, 2:5/0/64         ------>    |
  |                                                            |
  |  <------     ACK [MID=1239], 2.05 Content, 2:5/0/64        |
```
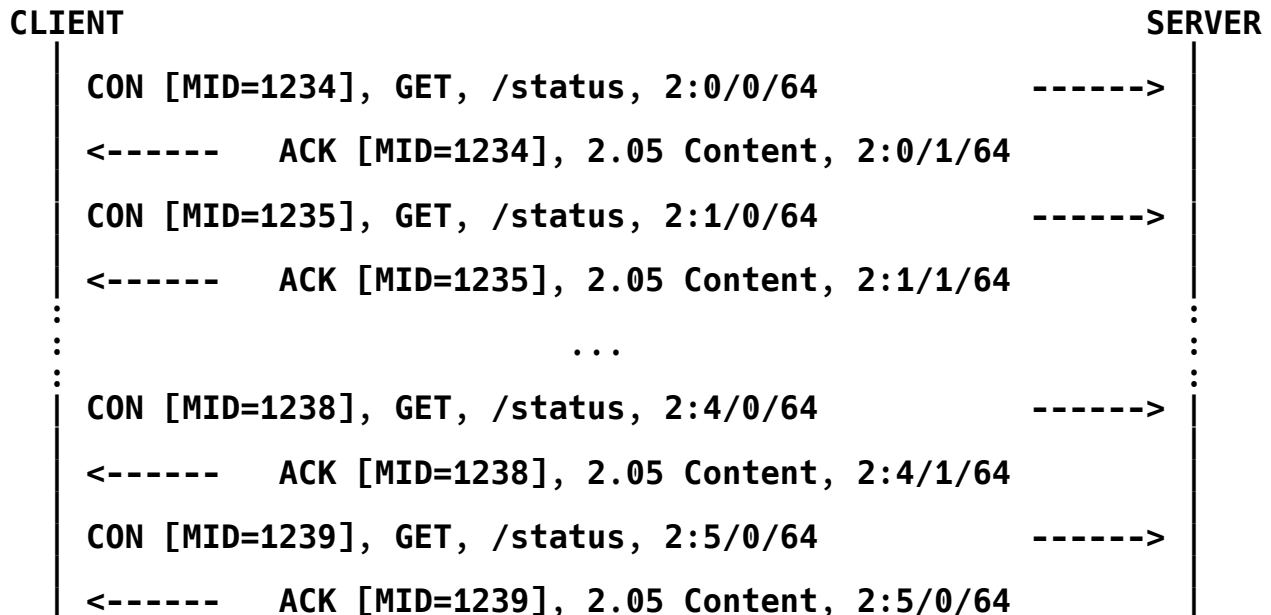
            Figure 3: Block-Wise GET with Early Negotiation

In the third example (Figure 4), the client is surprised by the need
for a block-wise transfer, and unhappy with the size chosen
unilaterally by the server. As it did not send a size proposal
initially, the negotiation only influences the size from the second
message exchange onward. Since the client already obtained both the
first and second 64-byte block in the first 128-byte exchange, it
goes on requesting the third 64-byte block ("2/0/64"). None of this
is (or needs to be) understood by the server, which simply responds
to the requests as it best can.

```
CLIENT                                                    SERVER
   |                                                         |
   | CON [MID=1234], GET, /status                    ------> |
   |                                                         |
   | <------     ACK [MID=1234], 2.05 Content, 2:0/1/128     |
   |                                                         |
   | CON [MID=1235], GET, /status, 2:2/0/64          ------> |
   |                                                         |
   | <------     ACK [MID=1235], 2.05 Content, 2:2/1/64      |
   |                                                         |
   | CON [MID=1236], GET, /status, 2:3/0/64          ------> |
   |                                                         |
   | <------     ACK [MID=1236], 2.05 Content, 2:3/1/64      |
   |                                                         |
   | CON [MID=1237], GET, /status, 2:4/0/64          ------> |
   |                                                         |
   | <------     ACK [MID=1237], 2.05 Content, 2:4/1/64      |
   |                                                         |
   | CON [MID=1238], GET, /status, 2:5/0/64          ------> |
   |                                                         |
   | <------     ACK [MID=1238], 2.05 Content, 2:5/0/64      |
```

              Figure 4: Block-Wise GET with Late Negotiation

In all these (and the following) cases, retransmissions are handled
by the CoAP message exchange layer, so they don't influence the block
operations (Figures 5 and 6).

```
 CLIENT                                                      SERVER
   |                                                           |
   |  CON [MID=1234], GET, /status                    ------>  |
   |                                                           |
   |  <------   ACK [MID=1234], 2.05 Content, 2:0/1/128        |
   |                                                           |
   |  CON [MID=1235], GE///////////////////////               |
   |                                                           |
   |  (timeout)                                                |
   |                                                           |
   |  CON [MID=1235], GET, /status, 2:2/0/64          ------>  |
   |                                                           |
   |  <------   ACK [MID=1235], 2.05 Content, 2:2/1/64         |
   :                                                           :
   :                        ...                                :
   :                                                           :
   |  CON [MID=1238], GET, /status, 2:5/0/64          ------>  |
   |                                                           |
   |  <------   ACK [MID=1238], 2.05 Content, 2:5/0/64         |
```
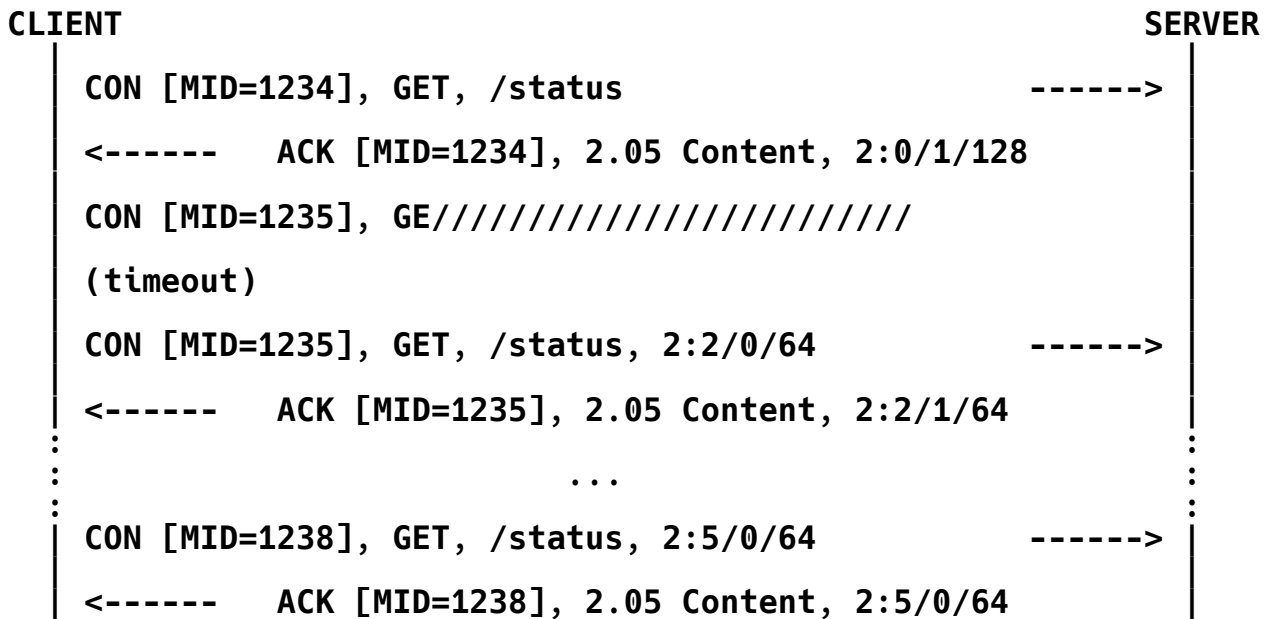
       Figure 5: Block-Wise GET with Late Negotiation and Lost CON

```
 CLIENT                                                      SERVER
   |                                                           |
   |  CON [MID=1234], GET, /status                    ------>  |
   |                                                           |
   |  <------   ACK [MID=1234], 2.05 Content, 2:0/1/128        |
   |                                                           |
   |  CON [MID=1235], GET, /status, 2:2/0/64          ------>  |
   |                                                           |
   |  ////////////////////////////////tent, 2:2/1/64          |
   |                                                           |
   |  (timeout)                                                |
   |                                                           |
   |  CON [MID=1235], GET, /status, 2:2/0/64          ------>  |
   |                                                           |
   |  <------   ACK [MID=1235], 2.05 Content, 2:2/1/64         |
   :                                                           :
   :                        ...                                :
   :                                                           :
   |  CON [MID=1238], GET, /status, 2:5/0/64          ------>  |
   |                                                           |
   |  <------   ACK [MID=1238], 2.05 Content, 2:5/0/64         |
```
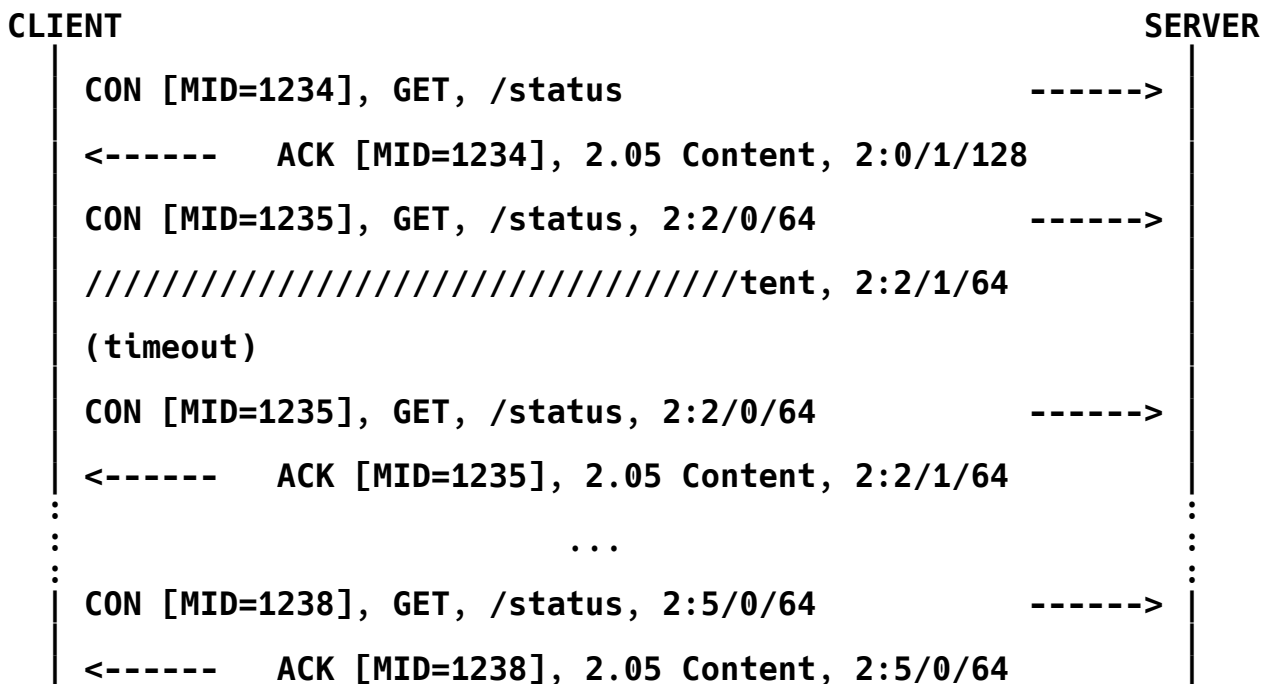
       Figure 6: Block-Wise GET with Late Negotiation and Lost ACK

3.2. Block1 Examples

   The following examples demonstrate a PUT exchange; a POST exchange
   looks the same, with different requirements on atomicity/idempotence.
   Note that, similar to GET, the responses to the requests that have a
   more bit in the request Block1 Option are provisional and carry the
   response code 2.31 (Continue); only the final response tells the
   client that the PUT succeeded.

```
CLIENT                                                          SERVER
   |                                                               |
   | CON [MID=1234], PUT, /options, 1:0/1/128      ------>         |
   |                                                               |
   | <------     ACK [MID=1234], 2.31 Continue, 1:0/1/128          |
   |                                                               |
   | CON [MID=1235], PUT, /options, 1:1/1/128      ------>         |
   |                                                               |
   | <------     ACK [MID=1235], 2.31 Continue, 1:1/1/128          |
   |                                                               |
   | CON [MID=1236], PUT, /options, 1:2/0/128      ------>         |
   |                                                               |
   | <------     ACK [MID=1236], 2.04 Changed, 1:2/0/128           |
```

                 Figure 7: Simple Atomic Block-Wise PUT

   A stateless server that simply builds/updates the resource in place
   (statelessly) may indicate this by not setting the more bit in the
   response (Figure 8); in this case, the response codes are valid
   separately for each block being updated. This is of course only an
   acceptable behavior of the server if the potential inconsistency
   present during the run of the message exchange sequence does not lead
   to problems, e.g., because the resource being created or changed is
   not yet or not currently in use.

```
CLIENT                                                          SERVER
  |                                                                |
  |  CON [MID=1234], PUT, /options, 1:0/1/128    ------>           |
  |                                                                |
  |  <------    ACK [MID=1234], 2.04 Changed, 1:0/0/128           |
  |                                                                |
  |  CON [MID=1235], PUT, /options, 1:1/1/128    ------>           |
  |                                                                |
  |  <------    ACK [MID=1235], 2.04 Changed, 1:1/0/128           |
  |                                                                |
  |  CON [MID=1236], PUT, /options, 1:2/0/128    ------>           |
  |                                                                |
  |  <------    ACK [MID=1236], 2.04 Changed, 1:2/0/128           |
```

Figure 8: Simple Stateless Block-Wise PUT

Finally, a server receiving a block-wise PUT or POST may want to
indicate a smaller block size preference (Figure 9). In this case,
the client SHOULD continue with a smaller block size; if it does, it
MUST adjust the block number to properly count in that smaller size.

```
CLIENT                                                          SERVER
  |                                                                |
  |  CON [MID=1234], PUT, /options, 1:0/1/128    ------>           |
  |                                                                |
  |  <------    ACK [MID=1234], 2.31 Continue, 1:0/1/32           |
  |                                                                |
  |  CON [MID=1235], PUT, /options, 1:4/1/32     ------>           |
  |                                                                |
  |  <------    ACK [MID=1235], 2.31 Continue, 1:4/1/32           |
  |                                                                |
  |  CON [MID=1236], PUT, /options, 1:5/1/32     ------>           |
  |                                                                |
  |  <------    ACK [MID=1235], 2.31 Continue, 1:5/1/32           |
  |                                                                |
  |  CON [MID=1237], PUT, /options, 1:6/0/32     ------>           |
  |                                                                |
  |  <------    ACK [MID=1236], 2.04 Changed, 1:6/0/32           |
```

Figure 9: Simple Atomic Block-Wise PUT with Negotiation

3.3. Combining Block1 and Block2

Block options may be used in both directions of a single exchange.
The following example demonstrates a block-wise POST request,
resulting in a separate block-wise response.

```
CLIENT                                                        SERVER
  |                                                             |
  |  CON [MID=1234], POST, /soap, 1:0/1/128      ------>        |
  |                                                             |
  |  <------    ACK [MID=1234], 2.31 Continue, 1:0/1/128        |
  |                                                             |
  |  CON [MID=1235], POST, /soap, 1:1/1/128      ------>        |
  |                                                             |
  |  <------    ACK [MID=1235], 2.31 Continue, 1:1/1/128        |
  |                                                             |
  |  CON [MID=1236], POST, /soap, 1:2/0/128      ------>        |
  |                                                             |
  |  <------    ACK [MID=1236], 2.04 Changed, 2:0/1/128, 1:2/0/128
  |                                                             |
  |  CON [MID=1237], POST, /soap, 2:1/0/128      ------>        |
  |  (no payload for requests with Block2 with NUM != 0)        |
  |  (could also do late negotiation by requesting,             |
  |   e.g., 2:2/0/64)                                           |
  |                                                             |
  |  <------    ACK [MID=1237], 2.04 Changed, 2:1/1/128         |
  |                                                             |
  |  CON [MID=1238], POST, /soap, 2:2/0/128      ------>        |
  |                                                             |
  |  <------    ACK [MID=1238], 2.04 Changed, 2:2/1/128         |
  |                                                             |
  |  CON [MID=1239], POST, /soap, 2:3/0/128      ------>        |
  |                                                             |
  |  <------    ACK [MID=1239], 2.04 Changed, 2:3/0/128         |
```

Figure 10: Atomic Block-Wise POST with Block-Wise Response
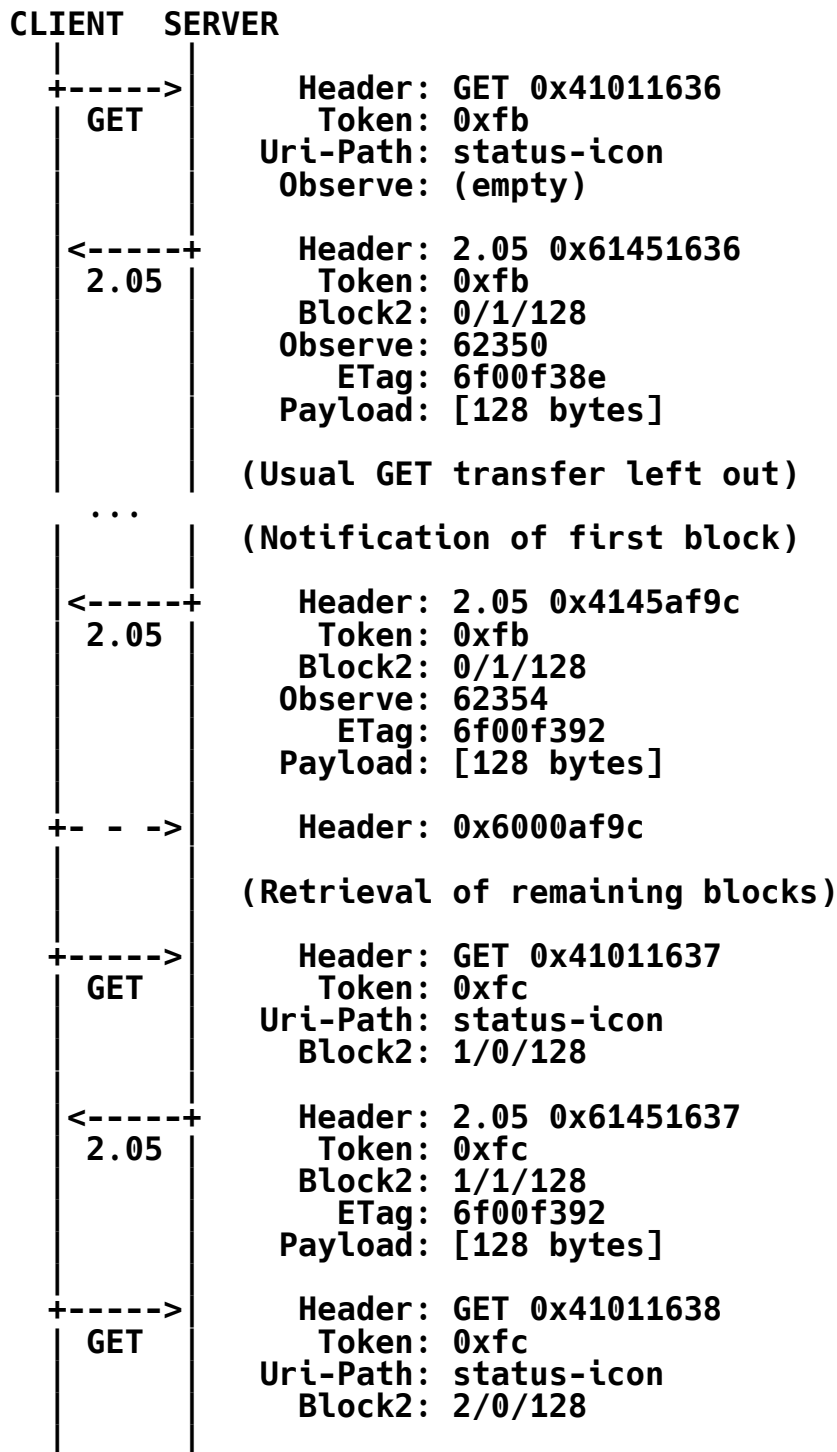
This model does provide for early negotiation input to the Block2
block-wise transfer, as shown below.

```
CLIENT                                                        SERVER
  |                                                              |
  | CON [MID=1234], POST, /soap, 1:0/1/128 ------>               |
  |                                                              |
  | <------     ACK [MID=1234], 2.31 Continue, 1:0/1/128         |
  |                                                              |
  | CON [MID=1235], POST, /soap, 1:1/1/128 ------>               |
  |                                                              |
  | <------     ACK [MID=1235], 2.31 Continue, 1:1/1/128         |
  |                                                              |
  | CON [MID=1236], POST, /soap, 1:2/0/128, 2:0/0/64 ------>     |
  |                                                              |
  | <------     ACK [MID=1236], 2.04 Changed, 1:2/0/128, 2:0/1/64|
  |                                                              |
  | CON [MID=1237], POST, /soap, 2:1/0/64       ------>          |
  | (no payload for requests with Block2 with NUM != 0)          |
  |                                                              |
  | <------     ACK [MID=1237], 2.04 Changed, 2:1/1/64           |
  |                                                              |
  | CON [MID=1238], POST, /soap, 2:2/0/64       ------>          |
  |                                                              |
  | <------     ACK [MID=1238], 2.04 Changed, 2:2/1/64           |
  |                                                              |
  | CON [MID=1239], POST, /soap, 2:3/0/64       ------>          |
  |                                                              |
  | <------     ACK [MID=1239], 2.04 Changed, 2:3/0/64           |
  |                                                              |
```

         Figure 11: Atomic Block-Wise POST with Block-Wise Response,
                              Early Negotiation

3.4. Combining Observe and Block2

In the following example, the server first sends a direct response
(Observe sequence number 62350) to the initial GET request (the
resulting block-wise transfer is as in Figure 4 and has therefore
been left out). The second transfer is started by a 2.05
notification that contains just the first block (Observe sequence
number 62354); the client then goes on to obtain the rest of the
blocks.

```
     CLIENT  SERVER
       |     |
       +----->|      Header: GET 0x41011636
       | GET  |       Token: 0xfb
       |     |    Uri-Path: status-icon
       |     |     Observe: (empty)
       |     |
       |<----+       Header: 2.05 0x61451636
       | 2.05 |       Token: 0xfb
       |     |      Block2: 0/1/128
       |     |     Observe: 62350
       |     |        ETag: 6f00f38e
       |     |     Payload: [128 bytes]
       |     |
       |     |    (Usual GET transfer left out)
      ...    |    (Notification of first block)
       |     |
       |<----+       Header: 2.05 0x4145af9c
       | 2.05 |       Token: 0xfb
       |     |      Block2: 0/1/128
       |     |     Observe: 62354
       |     |        ETag: 6f00f392
       |     |     Payload: [128 bytes]
       +- - ->|      Header: 0x6000af9c
       |     |
       |     |    (Retrieval of remaining blocks)
       |     |
       +----->|      Header: GET 0x41011637
       | GET  |       Token: 0xfc
       |     |    Uri-Path: status-icon
       |     |      Block2: 1/0/128
       |     |
       |<----+       Header: 2.05 0x61451637
       | 2.05 |       Token: 0xfc
       |     |      Block2: 1/1/128
       |     |        ETag: 6f00f392
       |     |     Payload: [128 bytes]
       |     |
       +----->|      Header: GET 0x41011638
       | GET  |       Token: 0xfc
       |     |    Uri-Path: status-icon
       |     |      Block2: 2/0/128
       |     |
```

```
    |<-----+        Header: 2.05 0x61451638
    | 2.05 |         Token: 0xfc
    |      |        Block2: 2/0/128
    |      |          ETag: 6f00f392
    |      |       Payload: [53 bytes]
```

          Figure 12: Observe Sequence with Block-Wise Response

   (Note that the choice of token 0xfc in this example is arbitrary;
   tokens are just shown in this example to illustrate that the requests
   for additional blocks cannot make use of the token of the Observation
   relationship. As a general comment on tokens, there is no other
   mention of tokens in this document, as block-wise transfers handle
   tokens like any other CoAP exchange. As usual, the client is free to
   choose tokens for each exchange as it likes.)

   In the following example, the client also uses early negotiation to
   limit the block size to 64 bytes.

```
      CLIENT   SERVER
        |        |
        +----->|        Header: GET 0x41011636
        | GET  |         Token: 0xfb
        |      |      Uri-Path: status-icon
        |      |       Observe: (empty)
        |      |        Block2: 0/0/64
        |      |
        |<-----+        Header: 2.05 0x61451636
        | 2.05 |         Token: 0xfb
        |      |        Block2: 0/1/64
        |      |       Observe: 62350
        |      |          ETag: 6f00f38e
        |      |       Max-Age: 60
        |      |       Payload: [64 bytes]
        |      |
        |      |     (Usual GET transfer left out)
        |      |
        ...    |     (Notification of first block)
        |      |
        |<-----+        Header: 2.05 0x4145af9c
        | 2.05 |         Token: 0xfb
        |      |        Block2: 0/1/64
        |      |       Observe: 62354
        |      |          ETag: 6f00f392
        |      |       Payload: [64 bytes]
        |      |
        |      |
```
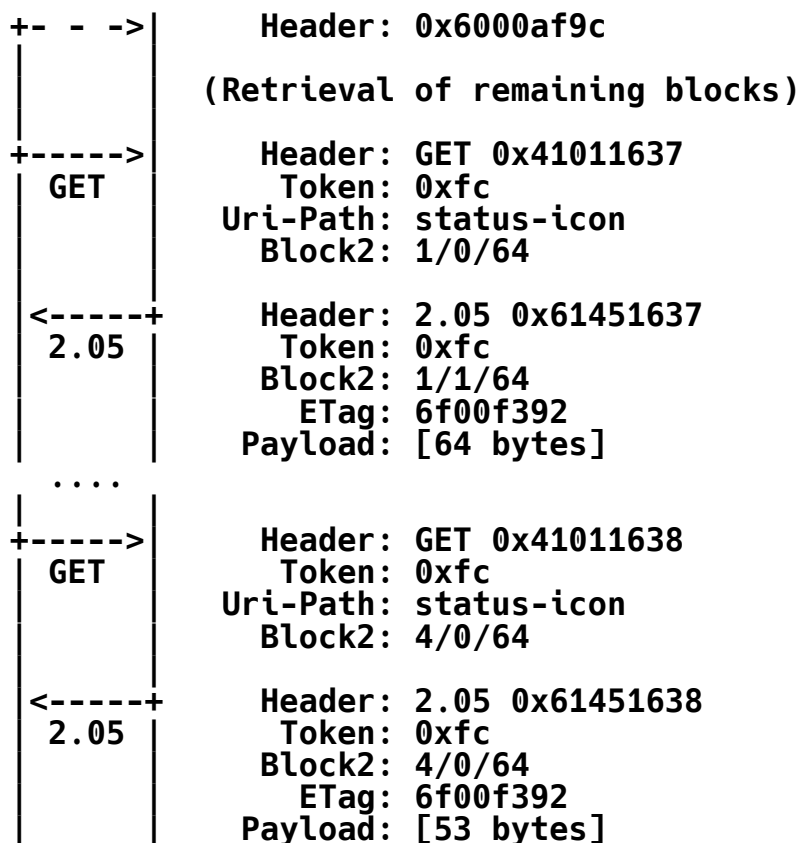
```
    +- - ->|        Header: 0x6000af9c
    |      |
    |      |     (Retrieval of remaining blocks)
    |      |
    +----->|        Header: GET 0x41011637
    | GET  |         Token: 0xfc
    |      |      Uri-Path: status-icon
    |      |        Block2: 1/0/64
    |      |
    |<-----+        Header: 2.05 0x61451637
    | 2.05 |         Token: 0xfc
    |      |        Block2: 1/1/64
    |      |          ETag: 6f00f392
    |      |       Payload: [64 bytes]
    |  ....
    |      |
    +----->|        Header: GET 0x41011638
    | GET  |         Token: 0xfc
    |      |      Uri-Path: status-icon
    |      |        Block2: 4/0/64
    |      |
    |<-----+        Header: 2.05 0x61451638
    | 2.05 |         Token: 0xfc
    |      |        Block2: 4/0/64
    |      |          ETag: 6f00f392
    |      |       Payload: [53 bytes]
```

          Figure 13: Observe Sequence with Early Negotiation

4. The Size2 and Size1 Options

   In many cases when transferring a large resource representation block
   by block, it is advantageous to know the total size early in the
   process. Some indication may be available from the maximum size
   estimate attribute "sz" provided in a resource description [RFC6690].
   However, the size may vary dynamically, so a more up-to-date
   indication may be useful.

   This specification defines two CoAP options, Size1 for indicating the
   size of the representation transferred in requests, and Size2 for
   indicating the size of the representation transferred in responses.
   (Size1 has already been defined in Section 5.10.9 of [RFC7252] to
   provide "size information about the resource representation in a
   request"; however, that section only details the narrow case of
   indicating in 4.13 responses the maximum size of request payload that
   the server is able and willing to handle. The present specification
   provides details about its use as a request option as well.)

The Size2 Option may be used for two purposes:

o In a request, to ask the server to provide a size estimate along
   with the usual response ("size request"). For this usage, the
   value MUST be set to 0.

o In a response carrying a Block2 Option, to indicate the current
   estimate the server has of the total size of the resource
   representation, measured in bytes ("size indication").

Similarly, the Size1 Option may be used for two purposes:

o In a request carrying a Block1 Option, to indicate the current
   estimate the client has of the total size of the resource
   representation, measured in bytes ("size indication").

o In a 4.13 response, to indicate the maximum size that would have
   been acceptable [RFC7252], measured in bytes.

Apart from conveying/asking for size information, the Size options
have no other effect on the processing of the request or response.
If the client wants to minimize the size of the payload in the
resulting response, it should add a Block2 Option to the request with
a small block size (e.g., setting SZX=0).

The Size options are "elective", i.e., a client MUST be prepared for
the server to ignore the size estimate request. Either Size option
MUST NOT occur more than once in a single message.

No.	C	U	N	R	Name	Format	Length	Default
60			x		Size1	uint	0-4	(none)
28			x		Size2	uint	0-4	(none)

Table 2: Size Option Numbers

Implementation Notes:

o As a quality of implementation consideration, block-wise transfers
   for which the total size considerably exceeds the size of one
   block are expected to include size indications, whenever those can
   be provided without undue effort (preferably with the first block
   exchanged). If the size estimate does not change, the indication
   does not need to be repeated for every block.

o The end of a block-wise transfer is governed by the M bits in the
   Block options, _not_ by exhausting the size estimates exchanged.

o As usual for an option of type uint, the value 0 is best expressed
   as an empty option (0 bytes). There is no default value for
   either Size option.

o The Size options are neither critical nor unsafe, and are marked
   as No-Cache-Key.

5. HTTP-Mapping Considerations

   In this subsection, we give some brief examples of the influence that
   the Block options might have on intermediaries that map between CoAP
   and HTTP.

   For mapping CoAP requests to HTTP, the intermediary may want to map
   the sequence of block-wise transfers into a single HTTP transfer.
   For example, for a GET request, the intermediary could perform the
   HTTP request once the first block has been requested and could then
   fulfill all further block requests out of its cache. A constrained
   implementation may not be able to cache the entire object and may use
   a combination of TCP flow control and (in particular if timeouts
   occur) HTTP range requests to obtain the information necessary for
   the next block transfer at the right time.

   For PUT or POST requests, historically there was more variation in
   how HTTP servers might implement ranges; recently, [RFC7233] has
   defined that Range header fields received with a request method other
   than GET are not to be interpreted. So, in general, the CoAP-to-HTTP
   intermediary will have to try sending the payload of all the blocks
   of a block-wise transfer for these other methods within one HTTP
   request. If enough buffering is available, this request can be
   started when the last CoAP block is received. A constrained
   implementation may want to relieve its buffering by already starting
   to send the HTTP request at the time the first CoAP block is
   received; any HTTP 408 status code that indicates that the HTTP
   server became impatient with the resulting transfer can then be
   mapped into a CoAP 4.08 response code (similarly, 413 maps to 4.13).

   For mapping HTTP to CoAP, the intermediary may want to map a single
   HTTP transfer into a sequence of block-wise transfers. If the HTTP
   client is too slow delivering a request body on a PUT or POST, the
   CoAP server might time out and return a 4.08 response code, which in
   turn maps well to an HTTP 408 status code (again, 4.13 maps to 413).
   HTTP range requests received on the HTTP side may be served out of a
   cache and/or mapped to GET requests that request a sequence of blocks
   that cover the range.

(Note that, while the semantics of CoAP 4.08 and HTTP 408 differ,
this difference is largely due to the different way the two protocols
are mapped to transport. HTTP has an underlying TCP connection,
which supplies connection state, so an HTTP 408 status code can
immediately be used to indicate that a timeout occurred during
transmitting a request through that active TCP connection. The CoAP
4.08 response code indicates one or more missing blocks, which may be
due to timeouts or resource constraints; as there is no connection
state, there is no way to deliver such a response immediately;
instead, it is delivered on the next block transfer. Still, HTTP 408
is probably the best mapping back to HTTP, as the timeout is the most
likely cause for a CoAP 4.08. Note that there is no way to
distinguish a timeout from a missing block for a server without
creating additional state, the need for which we want to avoid.)

6. IANA Considerations

This document adds the following option numbers to the "CoAP Option
Numbers" registry defined by [RFC7252]:

Number	Name	Reference
23	Block2	RFC 7959
27	Block1	RFC 7959
28	Size2	RFC 7959

Table 3: CoAP Option Numbers

This document adds the following response codes to the "CoAP Response
Codes" registry defined by [RFC7252]:

Code	Description	Reference
2.31	Continue	RFC 7959
4.08	Request Entity Incomplete	RFC 7959

Table 4: CoAP Response Codes

7. Security Considerations

Providing access to blocks within a resource may lead to surprising
vulnerabilities. Where requests are not implemented atomically, an
attacker may be able to exploit a race condition or confuse a server
by inducing it to use a partially updated resource representation.
Partial transfers may also make certain problematic data invisible to
Intrusion Detection Systems (IDSs); it is RECOMMENDED that an IDS
that analyzes resource representations transferred by CoAP implement
the Block options to gain access to entire resource representations.
Still, approaches such as transferring even-numbered blocks on one
path and odd-numbered blocks on another path, or even transferring
blocks multiple times with different content and obtaining a
different interpretation of temporal order at the IDS than at the
server, may prevent an IDS from seeing the whole picture. These
kinds of attacks are well understood from IP fragmentation and TCP
segmentation; CoAP does not add fundamentally new considerations.

Where access to a resource is only granted to clients making use of
specific security associations, all blocks of that resource MUST be
subject to the same security checks; it MUST NOT be possible for
unprotected exchanges to influence blocks of an otherwise protected
resource. As a related consideration, where object security is
employed, PUT/POST should be implemented in the atomic fashion,
unless the object security operation is performed on each access and
the creation of unusable resources can be tolerated. Future end-to-
end security mechanisms that may be added to CoAP itself may have
related security considerations, this includes considerations about
caching of blocks in clients and in proxies (see Sections 2.10 and 5
for different strategies in performing this caching); these security
considerations will need to be described in the specifications of
those mechanisms.

A stateless server might be susceptible to an attack where the
adversary sends a Block1 (e.g., PUT) block with a high block number:
A naive implementation might exhaust its resources by creating a huge
resource representation.

Misleading size indications may be used by an attacker to induce
buffer overflows in poor implementations, for which the usual
considerations apply.

7.1. Mitigating Resource Exhaustion Attacks

Certain block-wise requests may induce the server to create state,
e.g., to create a snapshot for the block-wise GET of a fast-changing
resource to enable consistent access to the same version of a
resource for all blocks, or to create temporary resource

representations that are collected until pressed into service by a
final PUT or POST with the more bit unset. All mechanisms that
induce a server to create state that cannot simply be cleaned up
create opportunities for denial-of-service attacks. Servers SHOULD
avoid being subject to resource exhaustion based on state created by
untrusted sources. But even if this is done, the mitigation may
cause a denial-of-service to a legitimate request when it is drowned
out by other state-creating requests. Wherever possible, servers
should therefore minimize the opportunities to create state for
untrusted sources, e.g., by using stateless approaches.

Performing segmentation at the application layer is almost always
better in this respect than at the transport layer or lower (IP
fragmentation, adaptation-layer fragmentation), for instance, because
there are application-layer semantics that can be used for mitigation
or because lower layers provide security associations that can
prevent attacks. However, it is less common to apply timeouts and
keepalive mechanisms at the application layer than at lower layers.
Servers MAY want to clean up accumulated state by timing it out (cf.
response code 4.08), and clients SHOULD be prepared to run block-wise
transfers in an expedient way to minimize the likelihood of running
into such a timeout.

7.2. Mitigating Amplification Attacks

[RFC7252] discusses the susceptibility of CoAP endpoints for use in
amplification attacks.

A CoAP server can reduce the amount of amplification it provides to
an attacker by offering large resource representations only in
relatively small blocks. With this, 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.

8. References

8.1. Normative References

   [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate
              Requirement Levels", BCP 14, RFC 2119,
              DOI 10.17487/RFC2119, March 1997,
              <http://www.rfc-editor.org/info/rfc2119>.

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

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

8.2. Informative References

   [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>.

   [RFC4919] Kushalnagar, N., Montenegro, G., and C. Schumacher, "IPv6
              over Low-Power Wireless Personal Area Networks (6LoWPANs):
              Overview, Assumptions, Problem Statement, and Goals",
              RFC 4919, DOI 10.17487/RFC4919, August 2007,
              <http://www.rfc-editor.org/info/rfc4919>.

   [RFC4944] Montenegro, G., Kushalnagar, N., Hui, J., and D. Culler,
              "Transmission of IPv6 Packets over IEEE 802.15.4
              Networks", RFC 4944, DOI 10.17487/RFC4944, September 2007,
              <http://www.rfc-editor.org/info/rfc4944>.

   [RFC6690] Shelby, Z., "Constrained RESTful Environments (CoRE) Link
              Format", RFC 6690, DOI 10.17487/RFC6690, August 2012,
              <http://www.rfc-editor.org/info/rfc6690>.

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

   [RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer
              Protocol (HTTP/1.1): Message Syntax and Routing",
              RFC 7230, DOI 10.17487/RFC7230, June 2014,
              <http://www.rfc-editor.org/info/rfc7230>.

   [RFC7233] Fielding, R., Ed., Lafon, Y., Ed., and J. Reschke, Ed.,
              "Hypertext Transfer Protocol (HTTP/1.1): Range Requests",
              RFC 7233, DOI 10.17487/RFC7233, June 2014,
              <http://www.rfc-editor.org/info/rfc7233>.

Acknowledgements

   Much of the content of this document is the result of discussions
   with the [RFC7252] authors, and via many CoRE WG discussions.

   Charles Palmer provided extensive editorial comments to a previous
   draft version of this document, some of which have been covered in
   this document. Esko Dijk reviewed a more recent version, leading to
   a number of further editorial improvements, a solution to the 4.13
   ambiguity problem, and the section about combining Block and
   multicast (Section 2.8). Markus Becker proposed getting rid of an
   ill-conceived default value for the Block2 and Block1 Options. Peter
   Bigot insisted on a more systematic coverage of the options and
   response code. Qin Wu provided a review for the IETF Operations
   directorate, and Goeran Selander commented on the security
   considerations.

   Kepeng Li, Linyi Tian, and Barry Leiba wrote up an early version of
   the Size option, which is described in this document. Klaus Hartke
   wrote some of the text describing the interaction of Block2 with
   Observe. Matthias Kovatsch provided a number of significant
   simplifications of the protocol.

   The IESG reviewers provided very useful comments. Spencer Dawkins
   even suggested new text. He and Mirja Kuehlewind insisted on more
   explicit information about the layering of block-wise transfers on
   top of the base protocol. Ben Campbell helped untangle some MUST/
   SHOULD soup. Comments by Alexey Melnikov, as well as the Gen-ART
   review by Jouni Korhonen, resulted in further improvements to the
   text.

Authors' Addresses

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

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

Zach Shelby (editor)
ARM
150 Rose Orchard
San Jose, CA 95134
United States of America

Phone: +1-408-203-9434
Email: zach.shelby@arm.com