CoRE Working Group C. Bormann
Internet-Draft Universitaet Bremen TZI
Intended status: Standards Track Z. Shelby, Ed.
Expires: July 16, 2012 Sensinode
January 13, 2012

Blockwise transfers in CoAP
draft-ietf-core-block-05

Abstract

CoAP is a RESTful transfer protocol for constrained nodes and networks. Basic CoAP messages work well for the small payloads we expect from temperature sensors, light switches, and similar building-automation devices. Occasionally, however, applications will need to transfer larger payloads -- for instance, for firmware updates. With HTTP, TCP does the grunt work of slicing large payloads up into multiple packets and ensuring that they all arrive and are handled in the right order.

CoAP is based on datagram transports such as UDP or DTLS, which limits the maximum size of resource representations that can be transferred without too much fragmentation. Although UDP supports larger payloads through IP fragmentation, it is limited to 64 KiB and, more importantly, doesn't really work well for constrained applications and networks.

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.

In summary, the Block options provide a minimal way to transfer larger representations in a block-wise fashion.

Table of Contents

1. Introduction

The CoRE WG is tasked with standardizing an Application Protocol for Constrained Networks/Nodes, 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:

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 (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,

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

CoAP is based on datagram transports such as UDP, which limit the maximum size of resource representations that can be transferred without creating unreasonable levels of IP fragmentation. 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) to enable the transport of larger representations is possible up to the maximum size of the underlying datagram protocol (such as UDP), but the fragmentation/reassembly process loads the lower layers with conversation state that is better managed in the application layer.

This 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 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.)

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:

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

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

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

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

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

- If needed, the Block options can also be used as is to provide random access to power-of-two sized blocks within a resource representation.

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 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 datagrams in constrained networks:

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

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

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

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 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.

In most cases, all blocks being transferred for a body will be of the same 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; still this size will be given as the block size even for the final block.

2.1. The Block Options

Type	C/E	Name	Format	Length	Default
19	Critical	Block1	uint	1-3 B	0 (see below)
17	Critical	Block2	uint	1-3 B	0 (see below)

Table 1: Block Option Numbers

Both Block1 and Block2 options can be present both in 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 [I-D.ietf-core-coap], 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, 2.05 -- see section "Payload" of [I-D.ietf-core-coap]).

(As a memory aid: Block_1_ pertains to the payload of the _1st_ part of the request-response exchange, i.e. the request, and Block_2_ pertains to the payload of the _2nd_ part of the request-response exchange, i.e. the response.)

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 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.

Three items of information may need to be transferred in a Block 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 option is a 1-, 2- or 3-byte integer which encodes these three fields, see Figure 1.

```
 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 as a three-bit unsigned integer (0 for 2**4 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 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)". (Note that, 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 block.)

The default value of both the Block1 and the Block2 Option is zero, indicating that the current block is the first and only block of the transfer (block number 0, M bit not set); however, there is no explicit size implied by this default value.

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. The block number is a variable-size (4, 12, or 20 bit) unsigned integer (uint, see Appendix A of [I-D.ietf-core-coap]) indicating the block number being requested or provided. Block number 0 indicates the first block of a 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 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.

The Block options are used in one of three roles:

o In descriptive usage, i.e. a Block2 Option in a response (e.g., a 2.05 response for GET), or a Block1 Option in a request (e.g., 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 are required to complete the transfer of that body.

  * The block size given by SZX MUST match the size of the payload in bytes, if the M bit is set. (The block size given is irrelevant 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 here. (The effect is that, if the server uses the smaller of its preferred block size and the one 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 block numbers other than 0).

- 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 and plans to implement the request atomically (e.g., acts only upon reception of the last block of payload). 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.

   * 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.

2.2. Using the Block Options

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

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 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 also uses the Block2 Option, 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, all requests in a single block-wise transfer MUST ultimately use 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 Block option in the response.) The server SHOULD use 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 the representations of which 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, 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's reassembler, 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 client MAY facilitate identifying the sequence by using the Token Option with a non-default value. 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.

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 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 whether the request or the acknowledgement was lost, this strategy is useful mostly for idempotent requests.)

In a blockwise 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. If not all previous blocks are available at the server at this time, the transfer fails and error code 4.08 (Request Entity Incomplete) MUST be returned. 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.

If multiple concurrently proceeding block-wise request payload transfer (e.g., PUT or POST) operations are possible, the requester SHOULD use the Token Option to clearly separate the different sequences. In this case, when reassembling the representation from the blocks being exchanged to enable atomic processing, the reassembler MUST compare any Token Options present (and, as usual, taking an absent Token Option to default to the empty Token). If atomic processing is not desired, there is no need to process the Token Option (but it is still returned in the response as usual).

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 separating the kind of Block option (1 or 2), block number (NUM), more bit (M), and block size exponent (2**(SZX+4)) by slashes. E.g., a Block2 Option value of 33 would be shown as 2/2/0/32), or a Block1 Option value of 59 would be shown as 1/3/1/128.

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 128 bytes of payload each, and third ACK contains 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 blockwise GET

In the second example (Figure 3), the client anticipates the blockwise transfer (e.g., because of a size indication in the link-format description) and sends a 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: Blockwise GET with early negotiation

In the third example (Figure 4), the client is surprised by the need for a blockwise 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: Blockwise 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 (Figure 5, Figure 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: Blockwise 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: Blockwise GET with late negotiation and lost ACK

The following examples demonstrate a PUT exchange; a POST exchange looks the same, with different requirements on atomicity/idempotence. To ensure that the blocks relate to the same version of the resource representation carried in the request, the client in Figure 7 sets the Token to "v17" in all requests. Note that, as with the GET, the responses to the requests that have a more bit in the request Block2 Option are provisional; only the final response tells the client that the PUT succeeded.

```
CLIENT                                                     SERVER
  |                                                          |
  | CON [MID=1234], PUT, /options, v17, 1/0/1/128    ------> |
  |                                                          |
  | <------   ACK [MID=1234], 2.04 Changed, 1/0/1/128        |
  |                                                          |
  | CON [MID=1235], PUT, /options, v17, 1/1/1/128    ------> |
  |                                                          |
  | <------   ACK [MID=1235], 2.04 Changed, 1/1/1/128        |
  |                                                          |
  | CON [MID=1236], PUT, /options, v17, 1/2/0/128    ------> |
  |                                                          |
  | <------   ACK [MID=1236], 2.04 Changed, 1/2/0/128        |
```

Figure 7: Simple atomic blockwise 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, v17, 1/0/1/128    ------> |
  |                                                          |
  | <------   ACK [MID=1234], 2.04 Changed, 1/0/0/128        |
  |                                                          |
  | CON [MID=1235], PUT, /options, v17, 1/1/1/128    ------> |
  |                                                          |
  | <------   ACK [MID=1235], 2.04 Changed, 1/1/0/128        |
  |                                                          |
  | CON [MID=1236], PUT, /options, v17, 1/2/0/128    ------> |
  |                                                          |
  | <------   ACK [MID=1236], 2.04 Changed, 1/2/0/128        |
```

Figure 8: Simple stateless blockwise PUT

Finally, a server receiving a blockwise 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, v17, 1/0/1/128    ------> |
     |                                                          |
     | <------   ACK [MID=1234], 2.04 Changed, 1/0/1/32         |
     |                                                          |
     | CON [MID=1235], PUT, /options, v17, 1/4/1/32     ------> |
     |                                                          |
     | <------   ACK [MID=1235], 2.04 Changed, 1/4/1/32         |
     |                                                          |
     | CON [MID=1236], PUT, /options, v17, 1/5/1/32     ------> |
     |                                                          |
     | <------   ACK [MID=1235], 2.04 Changed, 1/5/1/32         |
     |                                                          |
     | CON [MID=1237], PUT, /options, v17, 1/6/0/32     ------> |
     |                                                          |
     | <------   ACK [MID=1236], 2.04 Changed, 1/6/0/32         |
```

Figure 9: Simple atomic blockwise PUT with negotiation

4. HTTP Mapping Considerations

In this subsection, we give some brief examples for the influence 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. E.g., 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, there is more variation in how HTTP servers might implement ranges. Some WebDAV servers do, but in general the CoAP-to-HTTP intermediary will have to try sending the payload of all the blocks of a block-wise transfer 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 overlapping 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 a 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.)

5. IANA Considerations

This draft adds the following option numbers to the CoAP Option Numbers registry of [I-D.ietf-core-coap]:

Number	Name	Reference
17	Block2	[RFCXXXX]
19	Block1	[RFCXXXX]

Table 2: CoAP Option Numbers

This draft adds the following response code to the CoAP Response Codes registry of [I-D.ietf-core-coap]:

Code	Description	Reference
136	4.08 Request Entity Incomplete	[RFCXXXX]

Table 3: CoAP Response Codes

6. 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; it is RECOMMENDED that an intrusion detection system (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 a specific security association, 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.

6.1. Mitigating Resource Exhaustion Attacks

Certain blockwise requests may induce the server to create state, e.g. to create a snapshot for the blockwise 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), e.g. because there is

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 blockwise transfers in an expedient way to minimize the likelihood of running into such a timeout.

6.2. Mitigating Amplification Attacks

[I-D.ietf-core-coap] discusses the susceptibility of CoAP end-points 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.

7. Acknowledgements

Much of the content of this draft is the result of discussions with the [I-D.ietf-core-coap] authors, and via many CoRE WG discussions. Tokens were suggested by Gilman Tolle and refined by Klaus Hartke.

Charles Palmer provided extensive editorial comments to a previous version of this draft, some of which the authors hope to have covered in this version.

8. References

8.1. Normative References

[I-D.ietf-core-coap]
Shelby, Z., Hartke, K., Bormann, C., and B. Frank, "Constrained Application Protocol (CoAP)", draft-ietf-core-coap-08 (work in progress), October 2011.

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

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

8.2. Informative References

[REST] Fielding, R., "Architectural Styles and the Design of Network-based Software Architectures", 2000.

Appendix A. Historical Note

(This appendix to be deleted by the RFC editor.)

An earlier version of this draft used a single option:

Type	C/E	Name	Format	Length	Default
13	Critical	Block	uint	1-3 B	0 (see below)

Note that this option number has since been reallocated in [I-D.ietf-core-coap]; no backwards compatibility is provided after July 1st, 2011.

Authors' Addresses

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

Phone: +49-421-218-63921
Fax: +49-421-218-7000
Email: cabo@tzi.org

Zach Shelby (editor)
Sensinode
Kidekuja 2
Vuokatti 88600
Finland

Phone: +358407796297
Email: zach@sensinode.com