CoRE Working Group
Internet-Draft
Intended status: Standards Track
Expires: September 28, 2017

C. Amsuess
Energy Harvesting Solutions

March 27, 2017

Request-Tag option
draft-amsuess-core-request-tag-00

Abstract

This memo describes an optional extension to the Constrained Application Protocol (CoAP, [RFC7252] and [RFC7959]) that allows matching of request blocks. This primarily serves to transfer the security properties that Object Security of CoAP (OSCOAP, [I-D.ietf-core-object-security]) provides for single requests to blockwise transfers. The security of blockwise transfer in OSCOAP is reflected on in a dedicated section.

Status of This Memo

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

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

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

This Internet-Draft will expire on September 28, 2017.

Table of Contents

1. Introduction

The OSCOAP protocol provides a security layer for CoAP that, given a security context shared with a peer, provides

- encryption of payload and some options,
- integrity protection of the encrypted data and some more message options,
- protection against replays once a request has reached the server, and
- protected matching between request and response messages.

It does not (and should not) provide sequential delivery. In particular, it does not protect against requests being delayed; the corresponding attack and mitigation is described in [I-D.mattsson-core-coap-actuators].

The goal of this memo is to provide protection to the bodies of a blockwise fragmented request/response pair that is equivalent to the protection that would be provided if the complete request and

response bodies fit into single messae each. (Packing long payloads into single OSCOAP messages is actually possible using the outer blockwise mechanism, but does not go well with the constraints of devices CoAP is designed for). [Author's note: The results of this might move back into OSCOAP - for now, the matter is explored here.]

The proposed method of matching blocks to each other is the introduction of a Request-Tag option, which is similar to the ETag sent along with responses, but ephemeral and set by the client. It is phrased in a way that it can not only be used in OSCOAP, but also by other security mechanisms (eg. CoAP over DTLS), or for other purposes (see Appendix A).

In order to minimize the impact on message sizes, the Request-Tag option is designed to be only used when required[, and its interaction with OSCOAP should mandate actively setting it only in rare cases. If this is still insufficient, compressing it into the AAD can still be considered].

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

The terms "payload" and "body" are used as in [RFC7959]. The complete interchange of a request and a response body is called a REST "operation", while a request and response message (as matched by their tokens) is called an "exchange".

2. The Request-Tag option

A new option is defined for all request methods:

No.	C	U	N	R	Name	Format	Length	Default
TBD	x	x	-		Request-Tag	opaque	0-8	(none)

C=Critical, U=Unsafe, N=NoCacheKey, R=Repeatable

Figure 1: Option summary

It is critical (because a client that wants to secure its request body can't have a server ignore it), unsafe (because it needs to understood by any proxy that does blockwise (dis)assembly), and not repeatable. ([Does "unsafe" make nocachekey irrelevant? I think so.])

A client MAY set the Request-Tag option to indicate that the receiving server MUST NOT act on any block in the same blockwise operation that has a different Request-Tag set. A server MUST NOT use blocks with and blocks without Request-Tag option either.

[Note on future development: If it turns out we need to compress the option into the AAD, this might hook in here and specify that when OSCOAP and blockwise is in use, the client MUST set a Request-Tag if and only if it sets a Block1 option in descriptive usage, and is value MUST be the partial IV of that message. That value MUST then be included somewhere in the AAD of every block message _after_ the first, where this compression proposal so far fails because the verifying server would have to know at AAD-building time whether or not this is an inner blockwise request.]

If the Request-Tag option is set, the client MAY perform simultaneous operations that utilize Block1 fragmentation from the same endpoint towards the same resource, lifting the limitation of [RFC7959] section 2.5. The server is still under no obligation to keep state of more than one transaction. When an operation is in progress and a second one can not be served at the same time, the server MUST either respond to the second request with a 5.03 response code (in which it SHOULD indicate the time it is willing to wait for additional blocks in the first open operation in the Max-Age option), or cancel the first operation by responding 4.08 in subsequent exchanges in the first operations. Clients that see the latter behavior SHOULD [or MUST?] fall back to serializing requests as it would without the Request-Tag option.

[Author's note: The above paragraph sounds problematic to me. For further exploration of those error cases, I'd need to know how simultaneous operations (even on different resources) from different endpoints are handled in constrained clients; I only did stateless operations in constrained devices so far.]

The option is not used in responses.

If a request that uses Request-Tag is rejected with 4.02 Bad Option, the client MAY retry the operation without it, but it then needs to serialize all operations that affect the same resource. Security requirements can forbid dropping the Request-Tag option.

2.1. For inclusion in OSCOAP

[Author's note: If this stays a document of its own, OSCOAP should make a normative reference to it and state something like:

> Whenever the Block1 option is used as inner option, the Request-Tag option must be considered. A Request-Tag value (where the absence of a Request-Tag option is counted as a value too, and distinct from the empty option) can only be reused when all request messages sent in a different exchange with the same option value have either been answered (and successfully unprotected), or their sender sequence numbers differ from the next request by at least the window size (in which case they can not be accepted by the server after the new request has started).
>
> If the client follows the suggestion of only storing its own sequence numbers to persistent memory every K requests, it must increment the stored sequence number counter before using the last window-size sequence numbers available, because the remaining sequence numbes might only be used with certain constraints (it might be necessary to set a Request-Tag on them).

With this text, clients could even work around ever needing to send the option by bumping their sequence number - looks like bad behavior in the first place, but then again, it is just a variant of the "forbid out-of-order sequence numbers in blockwise" alternative option.

AFAICT this would be the first actual use of the window size; so far client and server can well interact with different replay window sizes. Probably it's OK to be the first user of the parameter.

For the options list:

The Request-Discriminator option is added to the "E=*" category in the options list, and is listed together with Block1/2 in all other places they are mentioned.

For somewhere else (?):

A server responding an inner Block2 option SHOULD use an ETag on it, even if the result is not cachable (eg. the response to a POST request), and take reasonable measures against identical ETags on distinct states, otherwise OSCOAP does not provide integrity protection of the response body.

]

3. Security properties of blockwise transfer

Blockwise transfer, specified in [RFC7959], fragments REST operations into exchanges of individual blocks. It provides, at the discretion of the server, direct access to parts of a resource representation

(where the client can fetch or send any block in any sequence, also called "random access") or sequential access (where the operation is started by exchanging the first block, and terminates in the exchange of the last block).

The individual blocks are correlated only by the client _endpoint_ (or security context if applicable), the requested _URI_, and _time_ (and thereby server state, where the operation is available at most until another request with the same endpoint/URI combination arrives).

The specification does include security considerations, which do advise against allowing random write access, but does not contain a mechanism that allows protecting the integrity of the operation's body. Consequently, the attacks described below are possible even when blockwise transfer is used over DTLS to the author's knowledge.

3.1. Blockwise transfer cases

There are several shapes a blockwise exchange can take, named here for further reference. Requests or responses bodies are called "small" or "large" here if they do or do not, respectively, fit in a single message. Empty bodies are small. Naming consists of case discrimination letters for No blockwise, Sequential transfer and Random access in the Block1 and Block2 phases, respectively.

[Author's note: I'd appreciate real examples to replace the more contrived ones; the worst are marked with (?).]

- _NN_: Request and response bodies are both small. No fragmentation happens.

  Examples: GETs to sensors, PUTs to actors.

  Integrity protection: Request/response matching is sufficient.

- _NS_: A small request causes a large response, which gets fragmented and sequentially fetched by the client.

  Examples: GETting an unfiltered link-format list, PUTting a compressed image to a picture frame that decides to return its (decompressed) state in full in the response(?).

  Integrity protection: The full request is copied in each subsequent request.

  Changes in the response need to be covered by the server setting a unique ETag.

Client and server could still disagree over whether the requests constitute a single or distinct REST operations; that's a general issue that should be pointed out. Note, however, that the _SS_ case _does_ provide that distinction! ["making non-blockwise as safe as blockwise" is not part of the mission statement (only the other way round), so probably we should accept this here and not try to provide that assurance - it would make every request a Request-Tag candidate, and rule out anything but NSTART=1 because the client couldn't know whether Block2 will be used.]

- _NR_: A small request is used to access a large one at random offsets.

  Examples: Inspecting a device's exposed memory.

  Integrity protection: Likewith _NS_, just that the distinction between single and distinct REST operations is presumably [check!] not meaningful anyway.

- _SN_: A large request is sent in sequential blocks with a small (typically empty) response.

  The server can, after any block, indicate that it has processed the blocks so far, and send a status for the processed ones.

  Examples: FETCHing a complex query, POSTing one's resource list to a resource directory.

  Integrity protection: The same Request-Tag gets set to all request blocks. The server treats blocks with a different tag (eg. replays from an earlier transmission) as different operations and possibly rejects them as incomplete entities.

- _RN_: A large request is sent in a random-access pattern, resulting in a small response(s) (typically, one response each, as the server would in that scenario send successful responses after each block or small groups of blocks.

  Examples: Storing data in a memory region of a device. (?)

  Integrity protection: The client can set a Request-Tag if it wants to group operations, but there is presumably [check!] no correlation to protect anyway.

- _SR_, _RR_: Large requests (sequentially or randomly requested) that have their large responses fetched in random access patterns - these cases are explicitly forbidden in blockwise transfer ([RFC7959] section 2.7).

- _RS_: [That's a tough one. A) I can't come up with examples, and B) the same section 2.7 says that Block2 processing starts when the _last_ block is done, implying that the request is sequential but not outright prescribing it. Furthermore, can there be inbetween successful replies?]

- _SS_: A large request is sent sequentially, and the large response is fetched in sequential blocks after the request has been transmitted in full.

  Integrity protection: The client sets a Request-Tag as in the _SN_ case. The last exchange (itself protected by OSCOAP's request/response matching) carries the Request-Tag option, and as with _NS_, the server sets an ETag.

  This is a case for which the Request-Tag use might need extending to the Block2 phase; while the protection is sufficient by passing the link on from Request-Tag to ETag, the server's state might be overridden by a simultaneous request (which the Request-Tag option promises to deal with), and the client may fail to retrieve the data because another request clears the state. This is problematic more for the proxy use case than for protected blockwise transfers. It is not fatal for the proxy case, though: It would need to serialize only the last exchange of the Block1 phase and the complete Block2 phase, but in that it does not depend on the client's data any more, can finish the Block2 phase quickly and spool the data for the client to fetch before finishing the next operation.

[Note that the _NS_ picture frame example is by far the worst and farest-fetched. I'd like to have an example of a non-safe request resulting in fragmented responses, but that behavior is usually discouraged (PUT responses typically being empty, POST responses bearing a Location), but not outright forbidden, and catered for in blockwise where it comes to combined use of Block1 and Block2.]

3.2. Attack scenarios

This section outlines some attacks that should be mitigated by the Request-Tag option. They are written with a malicious proxy between client and server in mind; whether that is a forward, reverse, transparent proxy, or any other entity on the data path that can intercept and inject packages into the communication is irrelevant to the attacks.

The illustrations draw terminology (especially the "@" and "X" symbols) from [I-D.mattsson-core-coap-actuators].

The scenarios typically require the attacker to have a good idea of the content of the packages that are transferred. Note that the attacker can see the codes of the messages.

3.2.1. "Promote Valjean" (on blockwise case SN)

In this scenario, blocks from two operations on a POST-accepting resource are combined to make the server execute an action that was not intended by the authorized client. This works only if the client attempts a second operation after first operation failed (due what the attacker made appear like a network outage) within the replay window. The client does not receive a confirmation on the second operation either, but by the time, the server has already executed the unauthorized action.

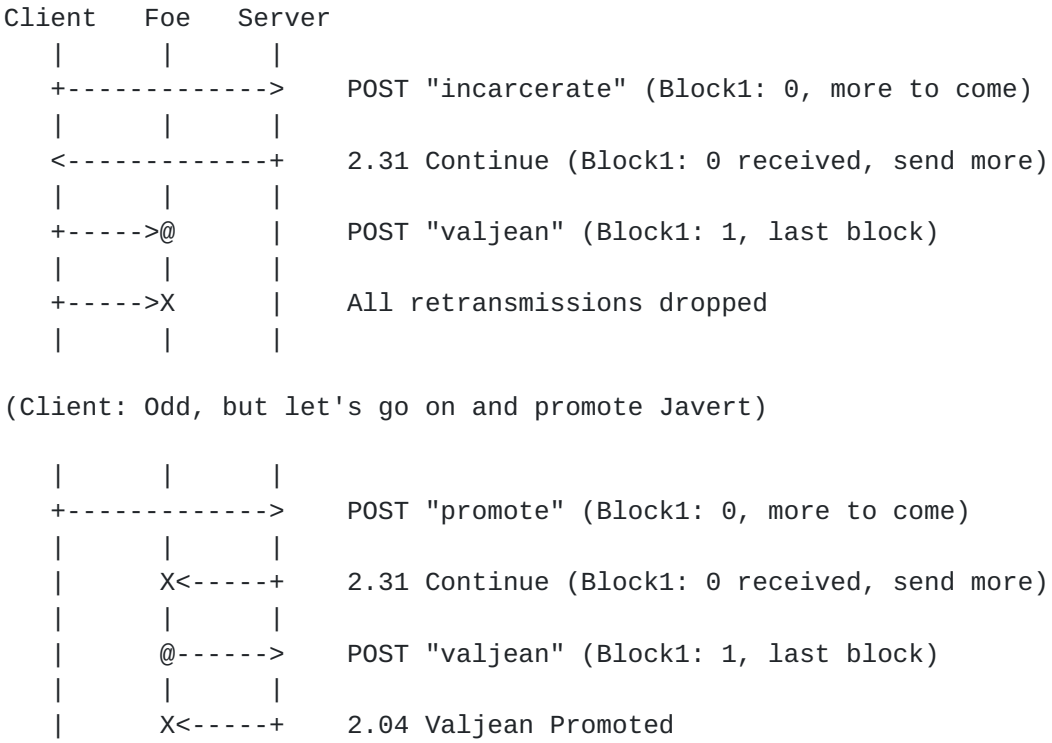


Figure 2: Attack example

With Request-Tag in place, the client would have assigned a different Request-Tag to the "promote" line, and the server would have either reacted to the "valjean" POST by incarcerating valjean (if it could keep both operation states at the same time), or responded 5.03 to the "promote" request until a timeout, or responded 4.08 to the injected "valjean" request.

The client would only have been free to use the same Request-Tag on the "promote" POST as on the "incarcerate" POST if, in the meantime, it had exchanged enough messages that the latest message of the first use ("valjean") is dropped from the server's window, and thus the sever would not accept its replay.

3.2.2. "Free the hitman" (blockwise case SN or SS)

In this example, mismatched Block1 packages against a resource that passes judgement are mixed up to create a response matched to the wrong operation.

Again, a first operation is aborted by the proxy ("Homeless stole apples. What shall we do with him?" - "Set him free."), and a part of that operation is later used in a different operation to prime the server for responding leniently to another operation that would originally have been "Hitman killed someone. What shall we do with him?" - "Hang him.".

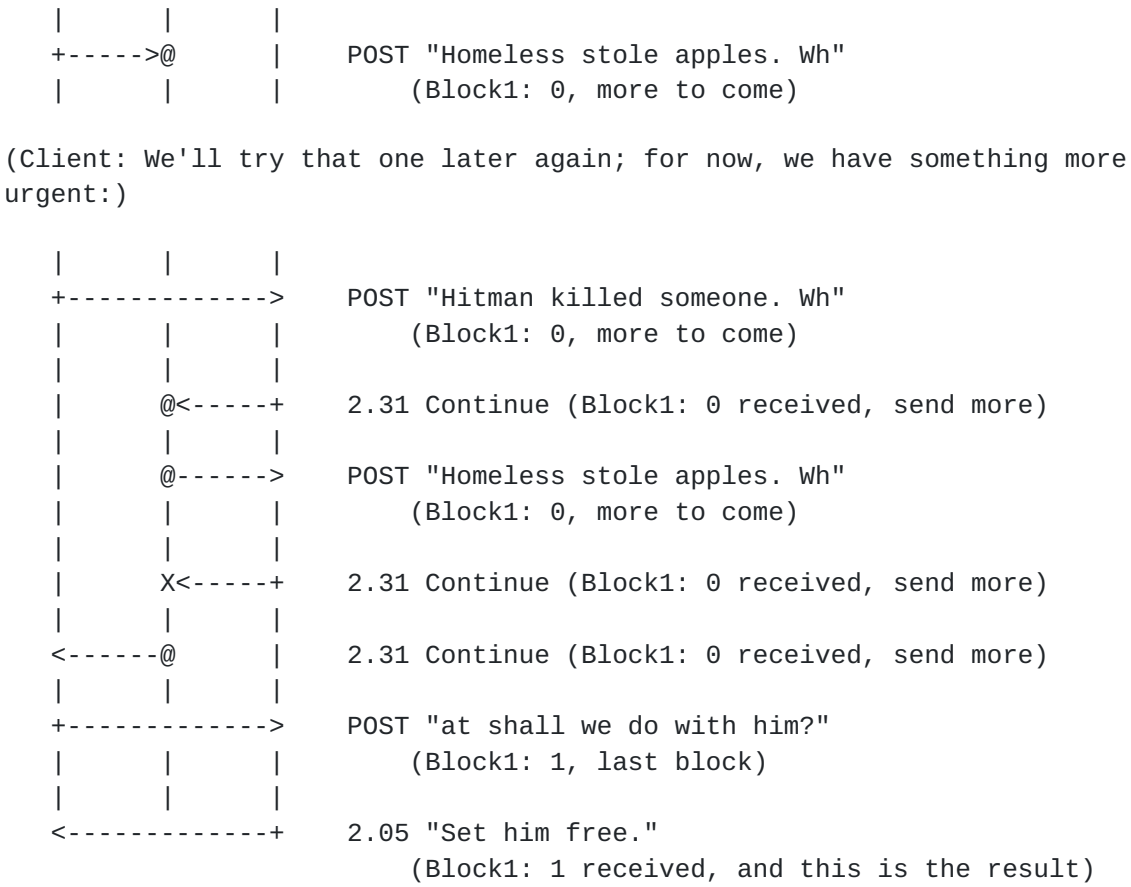


Figure 3: Attack example

The example works equivalently with longer responses, placing it in the _SS_ category instead of the _SN_.

[More examples would help, especially for the other blockwise cases. Is it relevant to distinguish non-piggybacked responses?]

4. Rationale

This part is informative and serves to illustrate why this option is necessary, and how it is different from similar concepts.

Why not...

- forbid out-of-order sequence numbers in blockwise?

  This could be a viable path. To see whether this works, the Section 3.1 chapter would hopefully help. (It should not rule out legitimate cases of random acces, after all).

  This would exclude other uses of the option like that in Appendix A.

- put an option in OSCOAP?

  This would work, and might in the end happen with compression of the Request-Tag option into the AAD.

  As before, this would exclude other uses cases.

- open up an endpoint per operation?

  This was explored in an earlier draft version as Request-Discriminator, which would have been a lightweight way to "multiplex" different endpoints (at least for the purpose of blockwise making references to them) into one secured connection.

  It is still the author's assumption that this would laregly be equivalent to the Request-Tag both in the OSCOAP application and in the use case explored in Appendix A, but the Request-Tag path is being explored currently because it is easier to understand, explain and reason about, while the Request-Discriminator way might result in less normative text with more comments, and possibly have similar effects in implementation codebases.

  A Request-Discriminator option could, among other things, be used by a proxies that act as OSCOAP terminators (eg. network interfaces in composite devices that use unencrypted CoAP on internal serial lines) to disambiguate request from different

security contexts towards crypto-unware but blockwise-capable components.

5. Security Considerations

When used in combination with OSCOAP or other security layers to prevent block mixing between REST operations, it is crucial to only reuse request tags as specified, and not to use any affected sequence numbers (which means the latest sequence number plus the window size) should information about used request tags get lost.

While the Request-Tag is not echoed back by the server unlike the Token, the client should still refrain from setting it to internal values (like memory address of state data) to avoid exposing internal data to a server that it could use in unrelated attacks.

6. IANA Considerations

[Missing: have a number assigned and the option published]

7. References

7.1. 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>.

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

7.2. Informative References

[I-D.ietf-core-object-security] Selander, G., Mattsson, J., Palombini, F., and L. Seitz, "Object Security of CoAP (OSCOAP)", draft-ietf-core-object-security-01 (work in progress), December 2016.

[I-D.mattsson-core-coap-actuators]
Mattsson, J., Fornehed, J., Selander, G., and F. Palombini, "Controlling Actuators with CoAP", draft-mattsson-core-coap-actuators-02 (work in progress), November 2016.

Appendix A. Use of Request-Tag by proxies

In pre-OSCOAP practice, proxies rarely face situations where simultaneous Block1 operations from different affect a single resource and can not be executed in parallel due to the constraints of only one Block1 operation being possible per endpoint pair and resource. (If that happens, the proxy can either serialize the requests, or 5.03 the second requester until the first request has completed).

With OSCOAP, all clients access the resource "/" as far as a proxy is concerned, which would lead to more frequent situations in which it would need to serialize requests. Clients that employ OSCOAP's outer-blockwise mechanism find themselves in a similar situation.

Those proxies and clients can utilize the Request-Tag option work off those requests in parallel by assigning them different Request-Tag values. To a proxy, this will only mean an increase in state of up to eight bytes per operation (if it could handle unencrypted simultaneous requests, it would tell them apart by their URIs; here, it tells them apart by their request tags). The state a server needs to keep per operation increases by the same eight bytes compared to serving the same simultaneous requests directly to different endopoints.

Appendix B. Examples

B.1. OSCOAP inner-blockwise

All messages exchanged in the following diagrams transferred as OSCOAP protected messages. The field data shown indicates code, payload and options of the unprotected (ie. inner) messages. Payloads are symbolic and do not necessarily line up in any block size when taken literally. Sequence numbers used are indicated at the sender side, and the window size used is 32.

Figure 4 shows how under usual circumstances, the Request-Tag option does not need to be set:

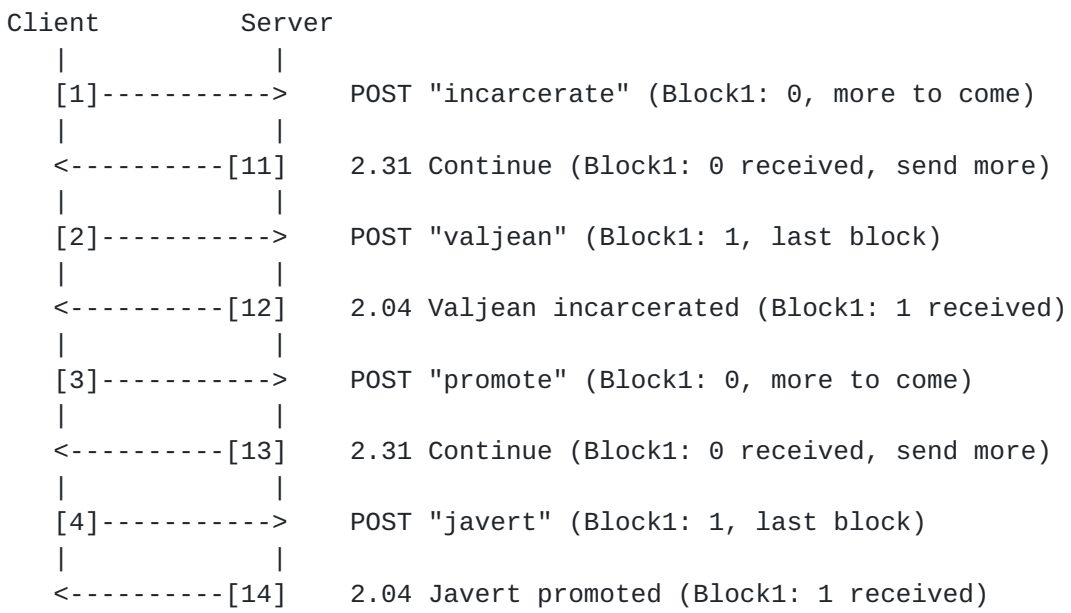


Figure 4: Back to back block transfer

If there is any doubt about whether all sent sequence numbers of a Request-Tag value are either acknowledged or off the window, the client uses a different value as in Figure 5. The client here uses the shortest possible value, the empty string:

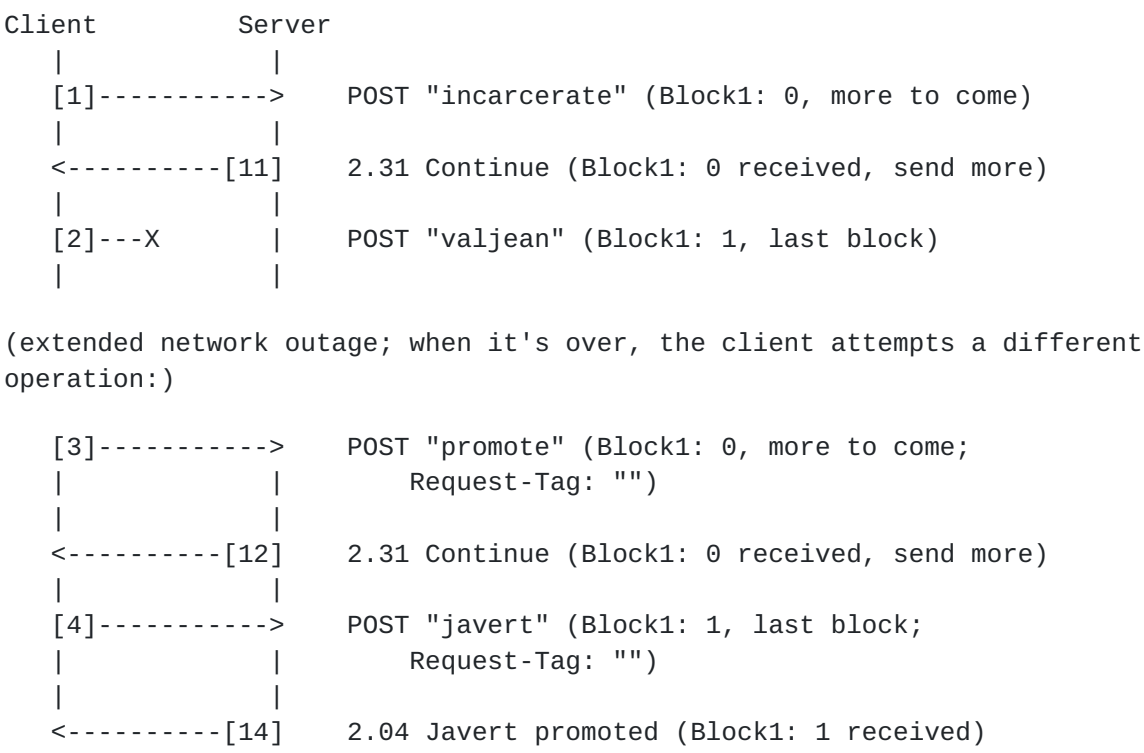
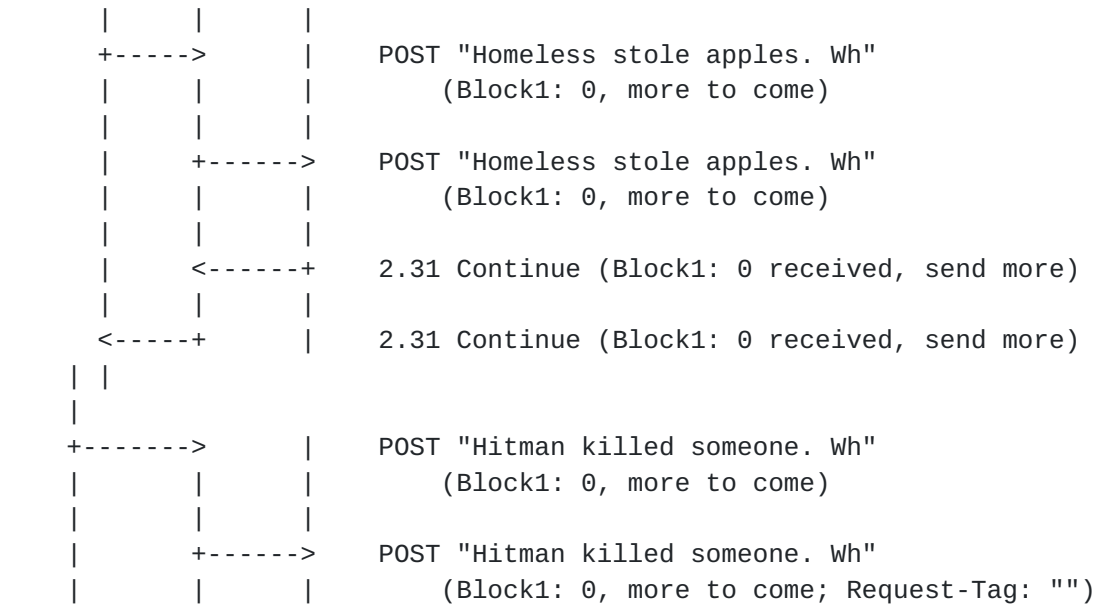


Figure 5: Behavior after extended package loss

B.2. Use by proxies

A proxy can use the Request-Tag option to work off operations from different clients (indicated by the two origin lines under "Clients") towards a single resource:



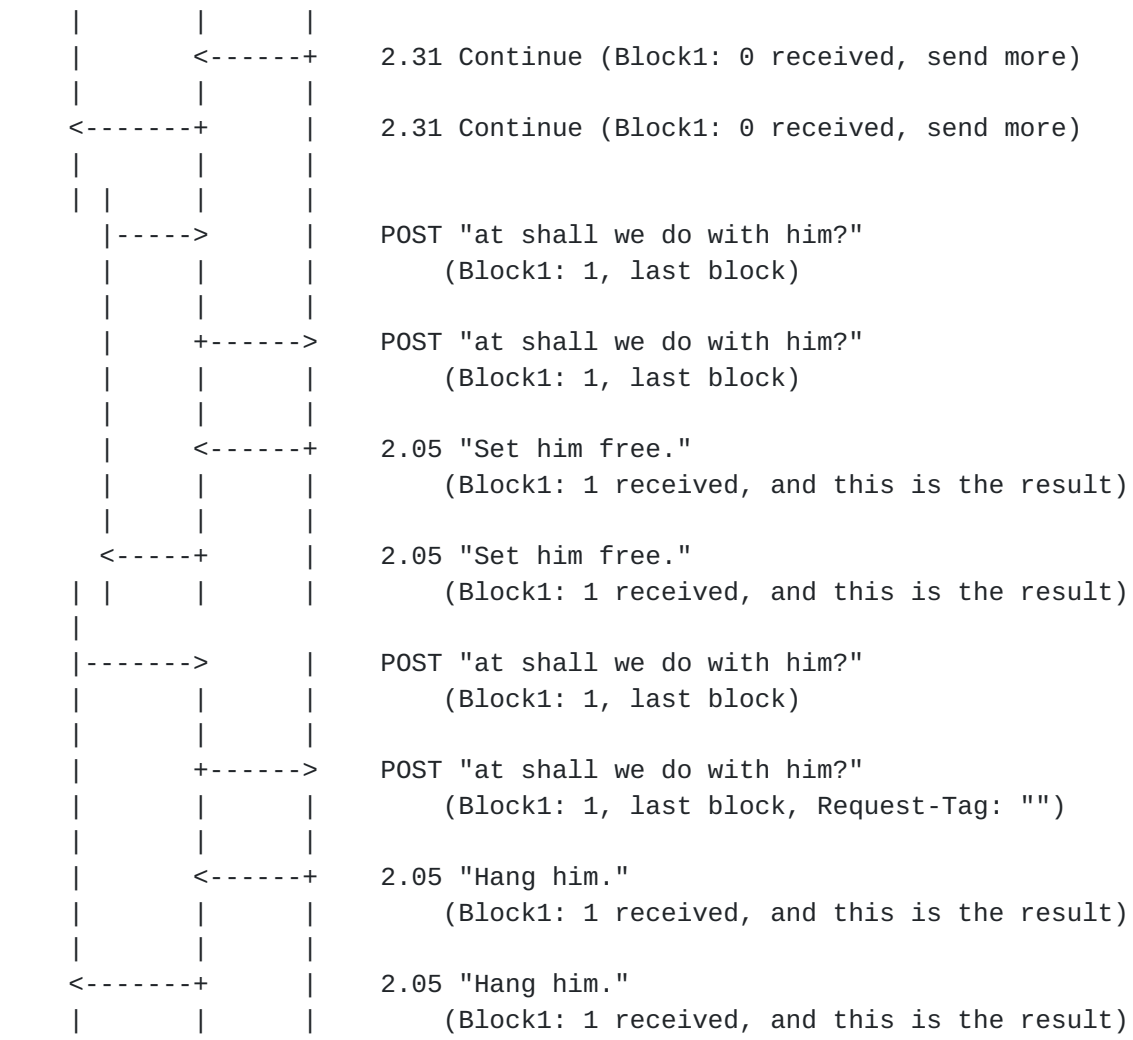


Figure 6: Proxy example

Author's Address

Christian Amsuess
Energy Harvesting Solutions

Email: c.amsuess@energyharvesting.at