

HTTP
Internet-Draft
Intended status: Standards Track
Expires: 3 September 2026

M. Kleidl, Ed.
Transloadit
G. Zhang, Ed.
Apple Inc.
L. Pardue, Ed.
Cloudflare
2 March 2026

Resumable Uploads for HTTP
draft-ietf-httpbis-resumable-upload-11

Abstract

HTTP data transfers can encounter interruption due to reasons such as canceled requests or dropped connections. If the intended recipient can indicate how much of the data was received prior to interruption, a sender can resume data transfer at that point instead of attempting to transfer all of the data again. HTTP range requests support this concept of resumable downloads from server to client. This document describes a mechanism that supports resumable uploads from client to server using HTTP.

About This Document

This note is to be removed before publishing as an RFC.

Status information for this document may be found at <https://datatracker.ietf.org/doc/draft-ietf-httpbis-resumable-upload/>.

Discussion of this document takes place on the HTTP Working Group mailing list (<mailto:ietf-http-wg@w3.org>), which is archived at <https://lists.w3.org/Archives/Public/ietf-http-wg/>. Working Group information can be found at <https://httpwg.org/>.

Source for this draft and an issue tracker can be found at <https://github.com/httpwg/http-extensions/labels/resumable-upload>.

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 <https://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 3 September 2026.

Copyright Notice

Copyright (c) 2026 IETF Trust and the persons identified as the document authors. All rights reserved.

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

Table of Contents

1. Introduction	4
2. Conventions and Definitions	5
3. Overview	5
3.1. Example 1: Complete upload of representation data with known size	6
3.2. Example 2: Upload as a series of parts	8
4. Upload Resource	10
4.1. State	11
4.1.1. Offset	11
4.1.2. Completeness	11
4.1.3. Length	12
4.1.4. Limits	13
4.2. Upload Creation	15
4.2.1. Client Behavior	15
4.2.2. Server Behavior	17
4.2.3. Examples	18
4.3. Offset Retrieval	19
4.3.1. Client Behavior	20
4.3.2. Server Behavior	20
4.3.3. Examples	21
4.4. Upload Append	22
4.4.1. Client Behavior	22
4.4.2. Server Behavior	23
4.4.3. Examples	25
4.5. Upload Cancellation	26

4.5.1. Client Behavior	26
4.5.2. Server Behavior	26
4.5.3. Example	26
4.6. Concurrency	26
5. Status Code 104 (Upload Resumption Supported)	27
6. Media Type application/partial-upload	28
7. Problem Types	28
7.1. Mismatching Offset	28
7.2. Completed Upload	29
7.3. Inconsistent Length	30
8. Content Codings	30
9. Transfer Codings	30
10. Integrity Digests	31
10.1. Representation Digests	31
10.2. Content Digests	31
11. Responses to Uploads	32
12. Upload Strategies	32
12.1. Optimistic Upload Creation	33
12.1.1. Upgrading To Resumable Uploads	33
12.2. Careful Upload Creation	34
13. Security Considerations	34
14. IANA Considerations	36
14.1. HTTP Fields	36
14.2. HTTP Status Code	36
14.3. Media Type	36
14.4. HTTP Problem Types	38
15. References	38
15.1. Normative References	38
15.2. Informative References	39
Appendix A. Changes	40
Since draft-ietf-httpbis-resumable-upload-10	40
Since draft-ietf-httpbis-resumable-upload-09	40
Since draft-ietf-httpbis-resumable-upload-08	40
Since draft-ietf-httpbis-resumable-upload-07	41
Since draft-ietf-httpbis-resumable-upload-06	41
Since draft-ietf-httpbis-resumable-upload-05	41
Since draft-ietf-httpbis-resumable-upload-04	41
Since draft-ietf-httpbis-resumable-upload-03	41
Since draft-ietf-httpbis-resumable-upload-02	42
Since draft-ietf-httpbis-resumable-upload-01	42
Since draft-ietf-httpbis-resumable-upload-00	42
Since draft-tus-httpbis-resumable-uploads-protocol-02	43
Since draft-tus-httpbis-resumable-uploads-protocol-01	43
Since draft-tus-httpbis-resumable-uploads-protocol-00	43
Appendix B. Draft Version Identification	43
Acknowledgments	44
Authors' Addresses	44

1. Introduction

HTTP data transfers can encounter interruption due to reasons such as canceled requests or dropped connections. If the intended recipient can indicate how much of the data was received prior to interruption, a sender can resume data transfer at that point instead of attempting to transfer all of the data again. HTTP range requests (see Section 14 of [HTTP]) support this concept of resumable data transfers for downloads from server to client. While partial PUT is one method for uploading a partial representation (via Content-Range in the request), there are caveats that affect its deployability; see Section 14.5 of [HTTP].

Canceled upload request can be triggered for various reasons, including but not limited to:

explicit client cancellation: e.g., terminating a user-agent process

implicit client cancellation: e.g., terminating a tab, garbage collecting a process or internal error

explicit server cancellation: e.g., scheduled maintenance

implicit server cancellation: e.g., DoS mitigation or internal error

Connections can be dropped due to a variety of network or transport layer reasons triggered by endpoints or on-path elements.

This specification defines a new mechanism for resumable uploads from client to server that can seamlessly fall back to conventional uploads. When an upload is interrupted, clients can send subsequent requests to query the server state and use this information to send the remaining representation data. Alternatively, they can cancel the upload entirely. Unlike ranged downloads, this protocol does not support transferring an upload as multiple requests in parallel.

Utilizing resumable uploads, applications can recover from unintended interruptions, but also interrupt an upload on purpose to later resume it, for example, when a user wants to pause an upload, the device's network connectivity changes, or bandwidth should be saved for higher priority tasks.

The document introduces the concept of an upload resource to facilitate resumable uploads (Section 4) and defines new header fields to communicate the state of the upload (Section 4.1), the status code 104 (Upload Resumption Supported) to indicate the server's support for resumable uploads (Section 5), and the application/partial-upload media type to label partial representation data when resuming an upload (Section 6).

2. Conventions and Definitions

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 BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

Some examples in this document contain long lines that may be folded, as described in [RFC8792].

The terms Structured Header, Item, Dictionary, String, Integer, and Boolean are imported from [STRUCTURED-FIELDS].

The terms "representation", "representation data", "representation metadata", "content", "client" and "server" are from Section 3 of [HTTP].

The term "URI" is used as defined in Section 4 of [HTTP].

The term "patch document" is taken from [PATCH].

An `_upload resource_` is a temporary resource on the server that facilitates the resumable upload of one representation (Section 4).

3. Overview

Resumable uploads are supported in HTTP through use of a temporary resource, an `_upload resource_` (Section 4), that is separate from the resource being uploaded to and specific to that upload. By interacting with the upload resource, a client can retrieve the current offset of the upload (Section 4.3), append to the upload (Section 4.4), and cancel the upload (Section 4.5).

The remainder of this section uses examples to illustrate different interactions with the upload resource. HTTP message exchanges, and thereby resumable uploads, use representation data (see Section 8.1 of [HTTP]). This means that resumable uploads can be used with many forms of content, such as static files, in-memory buffers, data from streaming sources, or on-demand generated data. Examples are purely

illustrative and non-normative. Implementations of this protocol are expected to follow normative requirements defined in other sections, together with applying security considerations presented in Section 13.

3.1. Example 1: Complete upload of representation data with known size

In this example, the client first attempts to upload representation data with a known size in a single HTTP request to the resource at `/project/123/files`. An interruption occurs and the client then attempts to resume the upload using subsequent HTTP requests to the upload resource at `/uploads/abc`.

1) The client notifies the server that it wants to begin an upload (Section 4.2). The server reserves the required resources to accept the upload from the client and then sends an interim response to the client, which signals the server's support of resumable upload as well as the upload resource's URI via the Location header field (Section 10.2.2 of [HTTP]). The client can start sending the representation data in the request content immediately after the request header. Alternatively, it could also await the acknowledgement in form of the interim response.

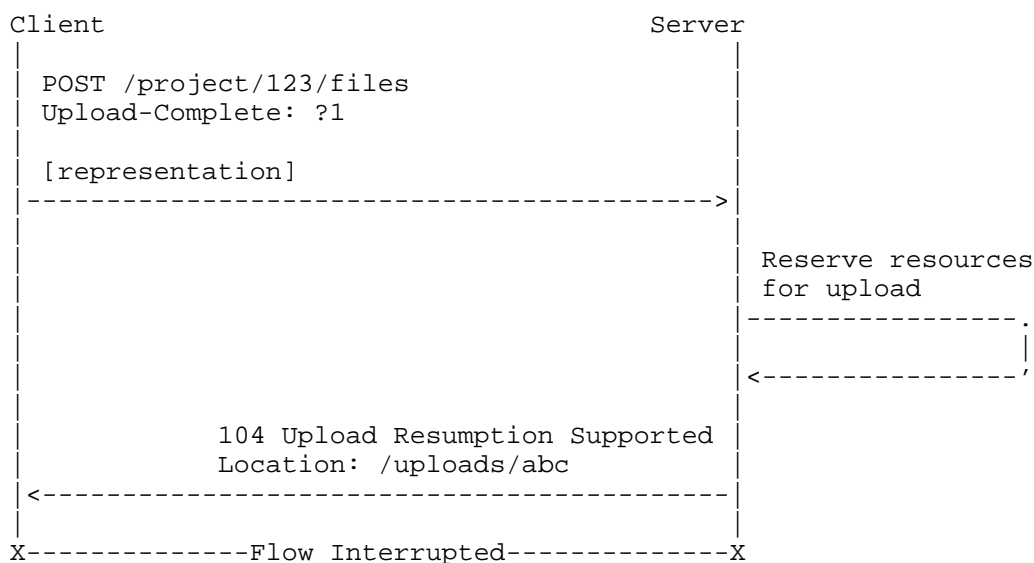


Figure 1: Upload Creation

2) If the connection to the server is interrupted, the client might want to resume the upload. However, before this is possible the client needs to know the amount of representation data that the server received before the interruption. It does so by retrieving the offset (Section 4.3) from the upload resource.

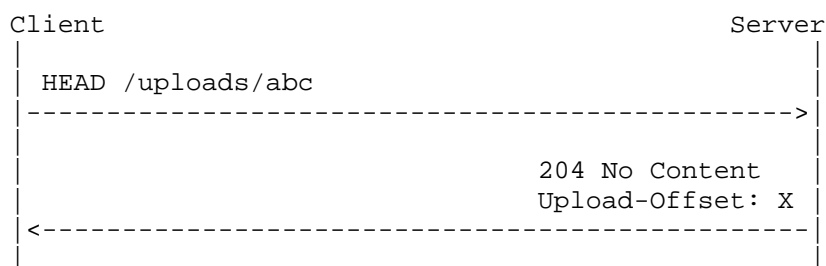


Figure 2: Offset Retrieval

3) The client can resume the upload by sending the remaining representation data to the upload resource (Section 4.4), appending to the already stored representation data in the upload using the application/partial-upload media type. The Upload-Offset value is included to ensure that the client and server agree on the offset that the upload resumes from. Once the remaining representation data is transferred, the server processes the entire representation and responds with whatever the initial request to /project/123/files would have produced if it had not been interrupted, e.g. a 200 (OK) response.

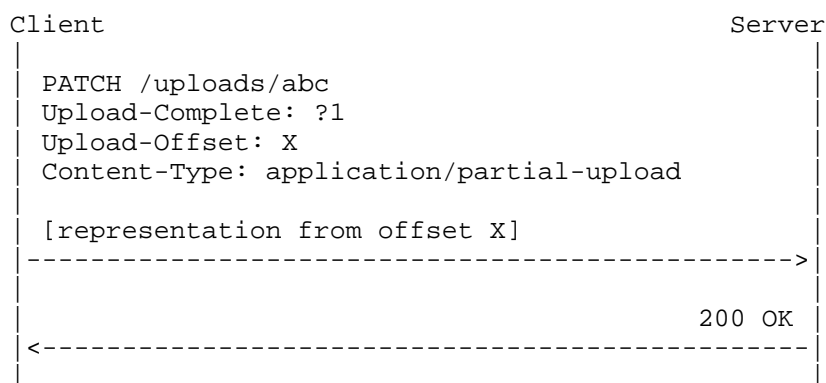


Figure 3: Upload Append

4) If the client is not interested in completing the upload, it can instruct the upload resource to delete the upload and free all related resources (Section 4.5).

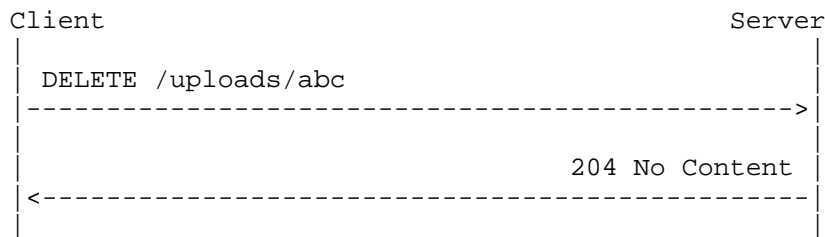


Figure 4: Upload Cancellation

3.2. Example 2: Upload as a series of parts

In some cases, clients might prefer to upload a representation as a series of parts sent serially across multiple HTTP messages. One use case is to overcome server limits on HTTP message content size. Another use case is where the client does not know the final size of the representation data, such as when the data originates from a streaming source.

This example shows how the client, communicating to a server known to support resumable upload, can upload parts of a representation incrementally.

1) If the client is aware that the server supports resumable upload, it can start an upload with the Upload-Complete field value set to false and the first part of the representation.

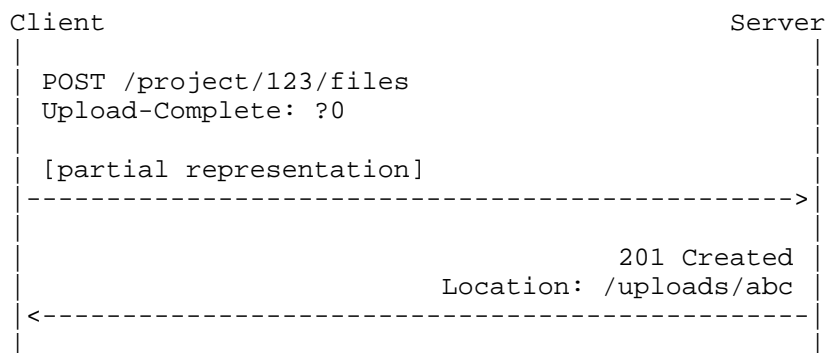


Figure 5: Upload creation with partial representation data

2) Next, intermediate parts are appended (Section 4.4) with the Upload-Complete field value set to false, indicating that they are not the last part of the representation data. The offset value in the Upload-Offset header field is taken from the previous response when creating the upload or appending to it.

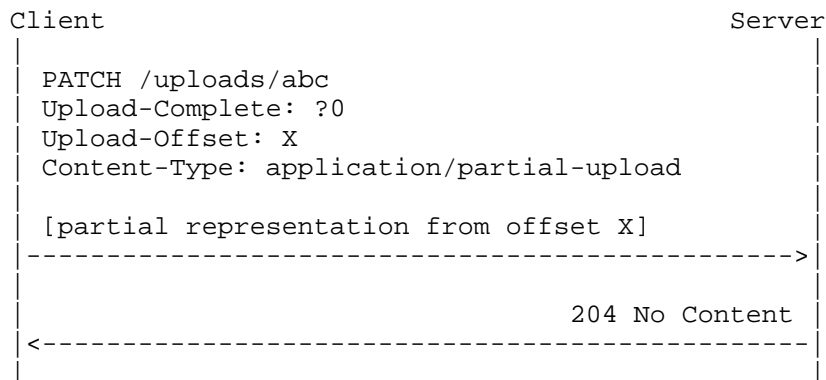


Figure 6: Appending partial representation data to upload

3) If the connection was interrupted, the client might want to resume the upload, similar to the previous example (Section 3.1). The client retrieves the offset (Section 4.3) to learn the amount of representation data received by the server and then continues appending the remaining parts to the upload as in the previous step.

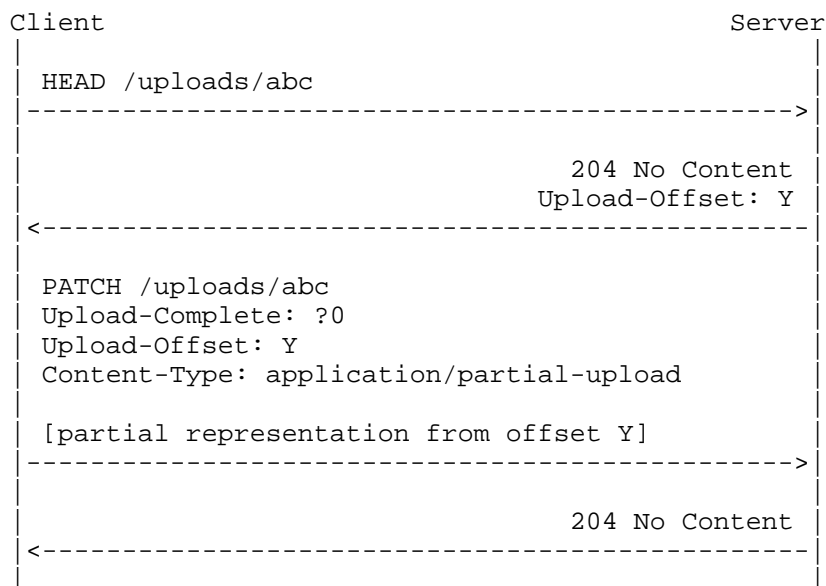


Figure 7: Resuming an interrupted upload

4) The request to append the last part of the representation data has a Upload-Complete field value set to true to indicate the complete transfer. Once the remaining representation data is transferred, the

server processes the entire representation and responds with whatever the initial request to `/project/123/files` would have produced if it had received the entire representation, e.g. a 200 (OK) response.

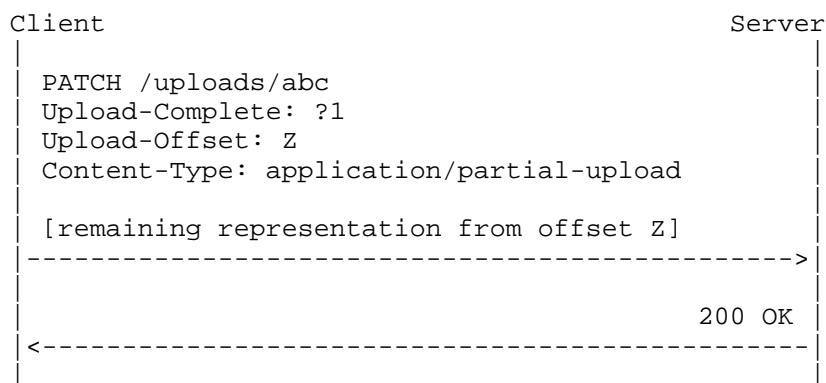


Figure 8: Appending remaining representation data

4. Upload Resource

A resumable upload is enabled through interaction with an upload resource. When a resumable upload begins, the server is asked to create an upload resource through a request to another resource (Section 4.2). This upload resource is responsible for handling the upload of a representation. Using the upload resource, the client can query the upload progress (Section 4.3), append representation data (Section 4.4), or cancel the upload (Section 4.5).

An upload resource is specific to the upload of one representation. For uploading multiple representations, multiple upload resources have to be used.

The server can clean up an upload resource and make it inaccessible immediately after the upload is complete. However, if a client didn't receive the last response acknowledging the upload's completion and the upload resource is not available anymore, the client cannot verify the upload's state with the server. Therefore, the server **SHOULD** keep the upload resource available for a reasonable amount of time after the upload is complete, or provide an alternate, application-specific way for the client to verify the completion of the upload.

An upload resource **SHOULD** be unique. Reuse of a URI for a different upload resource **SHOULD** be avoided in order to reduce the chance of misdirected or corrupted upload resources, as well as the potential security issues described in Section 13.

4.1. State

The state of an upload consists of the following properties that are tracked by the upload resource.

4.1.1. Offset

The offset is the number of bytes from the representation data that have been received, either during the creation of the upload resource (Section 4.2) and by appending to it (Section 4.4). The offset can be retrieved from the upload resource (Section 4.3) and is required when appending representation data (Section 4.4) to synchronize the client and resource regarding the amount of transferred representation data.

Representation data received by the upload resource cannot be removed again and, therefore, the offset **MUST NOT** decrease. If the server loses any part of the state, it **MUST** deactivate the upload resource and reject further interaction with it.

The Upload-Offset request and response header field conveys the offset. Upload-Offset is an Item Structured Header Field ([STRUCTURED-FIELDS]). Its value is a non-negative Integer (Section 3.3.1 of [STRUCTURED-FIELDS]) and indicates the current offset as viewed by the message sender. Other values **MUST** cause the entire header field to be ignored.

The Upload-Offset header field in responses serves as an acknowledgement of the received representation data and as a guarantee that no retransmission of it will be necessary. Clients can use this guarantee to free resources associated to transferred representation data.

4.1.2. Completeness

An upload is incomplete until it is explicitly marked as completed by the client. After this point, no representation data can be appended anymore.

The Upload-Complete request and response header field conveys the completeness state. Upload-Complete is an Item Structured Header Field ([STRUCTURED-FIELDS]). Its value is a Boolean (Section 3.3.6 of [STRUCTURED-FIELDS]) and indicates whether the upload is complete or not. Other values **MUST** cause the entire header field to be ignored.

An upload is marked as completed if a request for creating the upload resource (Section 4.2) or appending to it (Section 4.4) included the Upload-Complete header field with a true value and the request content was fully received.

4.1.3. Length

The length of an upload is the number of bytes of representation data that the client intends to upload.

Even the client might not know the total length of the representation data when starting the transfer, for example, because the representation is taken from a streaming source. However, a client SHOULD communicate the length to the upload resource as soon as it becomes known. There are two different ways for the client to indicate and the upload resource to discover the length from requests for creating the upload resource (Section 4.2) or appending to it (Section 4.4):

1. If the request includes the Upload-Complete field value set to true and a valid Content-Length header field, the request content is the remaining representation data. The length is then the sum of the current offset (Section 4.1.1) and the Content-Length header field value.
2. The request can include the Upload-Length header field defined below.

The Upload-Length request and response header field is an Item Structured Header Field ([STRUCTURED-FIELDS]). Its value is a non-negative Integer (Section 3.3.1 of [STRUCTURED-FIELDS]) and indicates the number of bytes of the entire representation data. Other values MUST cause the entire header field to be ignored.

If indicators (1) and (2) are both present in the same request, their indicated lengths MUST match. If multiple requests include indicators, their indicated values MUST match. A server can use the problem type [PROBLEM] of "https://iana.org/assignments/http-problem-types#inconsistent-upload-length" (Section 7.3) in responses to indicate inconsistent length values.

The Upload-Length field can be used in response to an offset retrieval; see Section 4.3.2.

The upload resource might not know the length until the upload is complete.

Note that the length and offset values do not determine whether an upload is complete. Instead, the client uses the Upload-Complete (Section 4.1.2) header field to indicate that a request completes the upload. The offset could match the length, but the upload can still be incomplete.

4.1.4. Limits

An upload resource MAY enforce one or multiple limits, which are communicated to the client via the Upload-Limit response header field. Upload-Limit is a Dictionary Structured Header Field ([STRUCTURED-FIELDS]). Its value is a Dictionary (Section 3.2 of [STRUCTURED-FIELDS]). Other values MUST cause the entire header field to be ignored.

The following key-value pairs are defined:

max-size: Specifies a maximum size for the representation data, counted in bytes. The server might not create an upload resource if the length (Section 4.1.3) deduced from the upload creation request is larger than the maximum size. The server might also deactivate the upload resource if the offset (Section 4.1.1) exceeds the maximum size. The value is an Integer.

min-size: Specifies a minimum size for the representation data, counted in bytes. The server might not create an upload resource if the length (Section 4.1.3) deduced from the upload creation request is smaller than the minimum size or no length can be deduced at all. The value is an Integer.

max-append-size: Specifies a maximum size counted in bytes for the request content in a single upload append request (Section 4.4). The server might reject requests exceeding this limit. A client that is aware of this limit MUST NOT send larger upload append requests. The value is an Integer.

min-append-size: Specifies a minimum size counted in bytes for the request content in a single upload append request (Section 4.4). The server might reject requests below this limit. A client that is aware of this limit MUST NOT send smaller upload append requests. The value is an Integer. Requests completing the upload by including the Upload-Complete: ?1 header field are exempt from this limit.

max-age: Specifies the remaining lifetime of the upload resource in

seconds counted from the generation of the response. After the resource's lifetime is reached, the server might make the upload resource inaccessible and a client SHOULD NOT attempt to access the upload resource as these requests will likely fail. The value is an Integer.

Clients usually discover limits through the Upload-Limit header field when the upload resource is created (Section 4.2). Throughout the lifetime of the upload resource, these limits SHOULD NOT change in a way that causes failures for clients adhering to the initially discovered limits. In particular, whereas the values for max-size, max-append-size, and max-age can increase without harm, they SHOULD NOT decrease as they can cause ongoing uploads to fail. Similarly, the value for min-append-size SHOULD NOT increase. If the client discovers that it cannot continue the upload while adhering to the limits, it SHOULD stop the current request immediately and cancel the upload (Section 4.5).

Receivers of Upload-Limit parse the Dictionary as described in Section 4.2 of [STRUCTURED-FIELDS]. Where the Dictionary is successfully parsed, this document places two additional requirements on Dictionary members. First, a member with an unknown key MUST be ignored. Second, a member with a known key but a value of unexpected type MUST cause the entire Upload-Limit header field to be ignored, or alternatively the complete HTTP message MUST be treated as malformed.

A server that supports the creation of a resumable upload resource (Section 4.2) for a target URI MUST include the Accept-Patch (Section 3.1 of [PATCH]) header field in response to an OPTIONS request for the target URI. If a server supports the creation of upload resources for any target URI, it SHOULD include the Accept-Patch header field in response to an OPTIONS request for the target * (if this target is supported). The value of the Accept-Patch header field MUST include the full application/partial-upload media type, even if it already contains */* or application/*. The Upload-Limit header field SHOULD be included in the response to an OPTIONS request if limits apply.

A client can use an OPTIONS request to discover support for resumable uploads and potential limits before creating an upload resource. While the values */* and application/* in the Accept-Patch response header field indicate the acceptance of the application/partial-upload in PATCH requests, it does not necessarily indicate support for all aspects of resumable uploads as defined in this document. To reduce the likelihood of failing requests, the limits announced in an OPTIONS response SHOULD NOT be less restrictive than the limits applied to an upload once the upload resource has been created,

unless the request to create an upload resource included additional information that warrants different limits. For example, a server might announce a general maximum size limit of 1GB, but reduce it to 100MB when the media type indicates an image.

Servers, including intermediaries, can (and often do) apply restrictions on the size of individual request message content. There is no standard mechanism to communicate such existing size restriction. A server that implements one can respond with a 413 Content Too Large status code; see Section 15.5.14 of [HTTP]. Appending to an upload resource, as a series of appends, can be used to upload data up to the max-size limit without encountering per-message limits. The min-append-size and max-append-size limits apply to the upload resource. Servers might apply restrictions that are smaller than the append limits, which would also result in a failed request. Clients could deal with such situations by retrying an upload append using a smaller size, as long as the new size resides between min-append-size and max-append-size. Cases where an append uses min-append-size yet fails with a 413 Content Too Large response might indicate a deployment mismatch that cannot be recovered from.

4.2. Upload Creation

4.2.1. Client Behavior

A client can start a resumable upload from any request that can carry content by including the Upload-Complete header field (Section 4.1.2). As a consequence, all request methods that allow content are possible, such as POST, PUT, and PATCH.

The Upload-Complete header field is set to true if the request content includes the entire representation data that the client intends to upload. This is also a requirement for transparently upgrading to resumable uploads from traditional uploads (Section 12.1.1).

If the client knows the representation data's length, it SHOULD indicate the length in the request to help the server allocate necessary resources for the upload and provide early feedback if the representation violates a limit (Section 4.1.4). This indication can be done through the Upload-Length header field or the combination of the Content-Length and Upload-Complete: ?1 header fields (see Section 4.1.3).

The client SHOULD respect any limits (Section 4.1.4) announced in the Upload-Limit header field in interim or final responses. In particular, if the allowed maximum size is less than the amount of representation data the client intends to upload, the client SHOULD stop the current request immediately and cancel the upload (Section 4.5).

The request content can be empty. If the Upload-Complete header field is then set to true, the client intends to upload an empty representation. An Upload-Complete header field set to false is also valid. This can be used to retrieve the upload resource's URI before transferring any representation data. Since interim responses are optional, this technique provides another mechanism to learn the URI, at the cost of an additional round-trip before data upload can commence.

Representation metadata included in the initial request (see Section 8.3 of [HTTP]) can affect how servers act on the uploaded representation data. The Content-Type header field (Section 8.3 of [HTTP]) indicates the media type of the representation. The Content-Encoding header field (Section 8.4 of [HTTP]) names the content codings applied to the representation. The Content-Disposition header field ([CONTENT-DISPOSITION]) can be used to transmit a filename. For this purpose, the inline disposition type is RECOMMENDED.

If the client received a final response with a

- * 2xx (Successful) status code and the entire representation data was transferred in the request content, the upload is complete and the response comes from the targeted resource processing the representation.
- * 2xx (Successful) status code and the entire representation data was not transferred in the request content, the Location response header field points the client to the created upload resource. The client can continue appending representation data to it (Section 4.4).
- * 4xx (Client Error) status code, the client SHOULD NOT attempt to retry or resume the upload, unless the semantics of the response allow or recommend the client to retry the request.
- * 5xx (Server Error) status code or no final response at all due to connectivity issues, the client MAY automatically attempt upload resumption by retrieving the current offset (Section 4.3) if it received the URI of the upload resource in a 104 (Upload Resumption Supported) interim response.

4.2.2. Server Behavior

Upon receiving a request with the Upload-Complete header field, the server can choose to offer resumption support by creating an upload resource. If so, it SHOULD announce the upload resource by sending an interim response with the 104 (Upload Resumption Supported) status code unless the server is not capable of sending interim responses. The interim response allows the client to resume the upload even if the message exchange gets later interrupted. When sent, it MUST include Location header field pointing to the upload resource and MUST include the Upload-Limit header field with the corresponding limits (Section 4.1.4) if existing.

The resource targeted by this initial request is responsible for processing the representation data transferred in the resumable upload according to the method and header fields in the initial request, while the upload resource enables resuming the transfer.

If the Upload-Complete request header field is set to true, the client intends to transfer the entire representation data in one request. If the request content was fully received, no resumable upload is needed and the resource proceeds to process the request and generate a response.

If the Upload-Complete header field is set to false, the client intends to transfer the representation over multiple requests. If the request content was fully received, the server MUST include the Location response header field pointing to the upload resource and MUST include the Upload-Limit header field with the corresponding limits if existing. Servers are RECOMMENDED to use the 201 (Created) status code.

The server MUST record the length according to Section 4.1.3 if the Upload-Length or Upload-Complete: ?1 header fields are included in the request.

While the request content is being received, the server MAY send multiple interim responses with a 104 (Upload Resumption Supported) status code and the Upload-Offset header field set to the current offset to inform the client about the upload progress.

Where a response requires a Location header field to be included, all interim and final response messages for the same request MUST contain an identical Location value. However, final responses including the Upload-Complete: ?1 header field are exempt from this requirement because they are the result of processing the transferred representation and the Location value does not necessarily represent the upload location. Where the Location value is expected to be

identical across multiple messages, clients SHOULD verify this. If verification fails, clients SHOULD abort the current request and cancel the upload (Section 4.5).

The server might not receive the entire request content when the upload is interrupted, for example because of dropped connection or canceled request. In this case, the server SHOULD append as much of the request content as possible to the upload resource, allowing the client to resume the upload from where it was interrupted. In addition, the upload resource MUST NOT be considered complete then.

4.2.3. Examples

A) The following example shows an upload creation, where the entire 123456789 bytes are transferred in the initial request. The server sends multiple interim responses and one final response from processing the uploaded representation.

```
POST /project/123/files HTTP/1.1
```

```
Host: example.com
```

```
Content-Disposition: inline; filename="file name.jpg"; filename*=UTF-8''file%20name.jp
```

g

```
Content-Length: 123456789
```

```
Upload-Length: 123456789
```

```
Upload-Complete: ?1
```

```
[content (123456789 bytes)]
```

```
HTTP/1.1 104 Upload Resumption Supported
```

```
Location: https://example.com/upload/b530ce8ff
```

```
Upload-Limit: max-size=1234567890
```

```
HTTP/1.1 104 Upload Resumption Supported
```

```
Upload-Offset: 23456789
```

```
HTTP/1.1 200 OK
```

```
Location: https://example.com/upload/b530ce8ff
```

```
Upload-Limit: max-size=1234567890
```

```
Content-Type: application/json
```

```
{"attachmentId": "b530ce8ff"}
```

B) The following example shows an upload creation, where only the first 23456789 bytes of a 123456789 bytes upload are transferred. The server acknowledges the received representation data and that the upload is not complete yet. The client can continue appending data.

```
POST /upload HTTP/1.1
Host: example.com
Upload-Complete: ?0
Content-Length: 23456789
Upload-Length: 123456789
```

```
[partial content (23456789 bytes)]
```

```
HTTP/1.1 104 Upload Resumption Supported
Location: https://example.com/upload/3fd4994ad
```

```
HTTP/1.1 201 Created
Location: https://example.com/upload/3fd4994ad
Upload-Limit: max-size=1234567890
```

C) The following example shows an upload creation, where the server responds with a 5xx status code. Thanks to the interim response containing the upload resource URI, the client can resume the upload.

```
POST /upload HTTP/1.1
Host: example.com
Upload-Complete: ?1
Content-Length: 123456789
Upload-Length: 123456789
```

```
[content (123456789 bytes)]
```

```
HTTP/1.1 104 Upload Resumption Supported
Location: https://example.com/upload/0587fa44b
```

```
HTTP/1.1 500 Internal Server Error
```

D) The following example shows an upload creation being rejected by the server because uploads to the target resource are not allowed. The client cannot continue the upload.

```
POST /upload-not-allowed HTTP/1.1
Host: example.com
Upload-Complete: ?1
Content-Length: 123456789
Upload-Length: 123456789
```

```
[content (123456789 bytes)]
```

```
HTTP/1.1 405 Method Not Allowed
```

4.3. Offset Retrieval

4.3.1. Client Behavior

If the client wants to resume the upload after an interruption, it has to know the amount of representation data received by the upload resource so far. It can fetch the offset by sending a HEAD request to the upload resource. Upon a successful response, the client can continue the upload by appending representation data (Section 4.4) starting at the offset indicated by the Upload-Offset response header field.

The offset can be less than or equal to the number of bytes of representation data that the client has already sent. The client MAY reject an offset which is greater than the number of bytes it has already sent during this upload by not continuing the upload and canceling the upload resource (Section 4.5). The client is expected to handle backtracking of a reasonable length. If the offset is invalid for this upload, or if the client cannot backtrack to the offset and reproduce the same representation data it has already sent, the upload MUST be considered a failure. The client then MUST NOT continue the upload and SHOULD cancel the upload (Section 4.5).

The client MUST NOT perform offset retrieval while creation (Section 4.2) or appending (Section 4.4) is in progress as this can cause the previous request to be terminated by the server as described in Section 4.6.

If the client received a response with a

- * 2xx (Successful) status code, the client can continue appending representation data to it (Section 4.4) if the upload is not complete yet.
- * 307 (Temporary Redirect) or 308 (Permanent Redirect) status code, the client MAY retry retrieving the offset from the new URI.
- * 4xx (Client Error) status code, the client SHOULD NOT attempt to retry or resume the upload, unless the semantics of the response allow or recommend the client to retry the request.
- * 5xx (Server Error) status code or no final response at all due to connectivity issues, the client MAY retry retrieving the offset.

4.3.2. Server Behavior

A successful response to a HEAD request against an upload resource

- * MUST include the offset in the Upload-Offset header field (Section 4.1.1),

- * MUST include the completeness state in the Upload-Complete header field (Section 4.1.2),
- * MUST include the length in the Upload-Length header field, unless the client has not supplied one, by providing the corresponding headers as described in (Section 4.1.3),
- * MUST indicate the limits in the Upload-Limit header field (Section 4.1.4), and
- * SHOULD include the Cache-Control header field with the value no-store to prevent HTTP caching ([CACHING]).

The resource SHOULD NOT generate a response with the 301 (Moved Permanently) and 302 (Found) status codes because clients might follow the redirect without preserving the HEAD method.

4.3.3. Examples

A) The following example shows an offset retrieval request. The server indicates the current offset and that the upload is not complete yet. The client can continue to append representation data.

```
HEAD /upload/c35e2cd29 HTTP/1.1
Host: example.com
```

```
HTTP/1.1 204 No Content
Upload-Complete: ?0
Upload-Offset: 25000000
Upload-Length: 100000000
Upload-Limit: max-age=3600
Cache-Control: no-store
```

B) The following example shows an offset retrieval request for a completed upload. The client does not need to continue the upload.

```
HEAD /upload/a9edb781b HTTP/1.1
Host: example.com
```

```
HTTP/1.1 204 No Content
Upload-Complete: ?1
Upload-Offset: 100000000
Upload-Length: 100000000
Cache-Control: no-store
```

4.4. Upload Append

4.4.1. Client Behavior

A client can continue the upload and append representation data by sending a PATCH request with the application/partial-upload media type (Section 6) to the upload resource. The request content is the representation data to append.

The client **MUST** indicate the offset of the request content inside the representation data by including the Upload-Offset request header field. To ensure that the upload resource will accept the request, the offset **SHOULD** be taken from an immediate previous response for retrieving the offset (Section 4.3) or appending representation data (Section 4.4).

The request **MUST** include the Upload-Complete header field. Its value is true in two cases:

- * the request has content that is the end of the representation data. Once the content is fully received by the server, the upload is complete.
- * the request has no content. Once the request is processed by the server, the upload is complete. This usage requires the full representation data to have been received via prior requests.

If the client received a final response with the Upload-Complete: ?1 header field, the upload is complete and the corresponding response comes from the resource processing the representation according to the initial request (see Section 4.2). Note that the status code does not necessarily indicate success. 4xx (Client Error) or 5xx (Server Error) status codes indicate in this case that the representation was fully transmitted, but an error occurred while processing it. Resuming the upload would not resolve this error.

If the client received a final response with the Upload-Complete: ?0 header field or the header field missing, the next step depends on the status code:

- * A 2xx (Successful) status code indicates that representation data was appended but the upload is not complete. The client can continue appending representation data.
- * For a 307 (Temporary Redirect) or 308 (Permanent Redirect) status code, the client **MAY** retry appending to the new URI.

- * For a 4xx (Client Error) status code, the client SHOULD NOT attempt to retry or resume the upload, unless the semantics of the response allow or recommend the client to retry the request.
- * For a 5xx (Server Error) status code, the client MAY automatically attempt upload resumption by retrieving the current offset (Section 4.3).

If no final response was received at all due to connectivity issues, the client MAY automatically attempt upload resumption by retrieving the current offset (Section 4.3).

4.4.2. Server Behavior

An upload resource applies a PATCH request with the application/partial-upload media type (Section 6) by appending the patch document in the request content to the upload resource.

The server might not receive the entire patch document when the upload is interrupted, for example because of a dropped connection or canceled request. In this case, the server SHOULD append as much of the patch document as possible to the upload resource, starting at its beginning and without discontinuities. Appending a continuous section starting at the patch document's beginning constitutes a successful PATCH as defined in Section 2 of [PATCH]. Saving the received data allows the client to resume the upload from where it was interrupted. In addition, the upload resource MUST NOT be considered complete then.

If the Upload-Offset request header field value does not match the current offset (Section 4.1.1), the upload resource MUST reject the request with a 409 (Conflict) status code. The response MUST include the correct offset in the Upload-Offset header field. The response can use the problem type [PROBLEM] of "https://iana.org/assignments/http-problem-types#mismatching-upload-offset" (Section 7.1).

If the upload is already complete (Section 4.1.2), the server MUST NOT modify the upload resource and MUST reject the request. The choice of response depends on the nature of the upload request and server state, including but not limited to:

- * If the client attempted to append a non-zero length document, the server MUST treat this as an inconsistent length failure. The server can use the problem type [PROBLEM] of "https://iana.org/assignments/http-problem-types#inconsistent-upload-length" (Section 7.3) in the response.
- * If the client attempted to append a zero-length document:

- If the server retains knowledge of the upload resource but it is not available, the 410 (Gone) status code can be used; the response could also contain a Location header field to indicate the location of the completed upload. The server can use the problem type [PROBLEM] of "https://iana.org/assignments/http-problem-types#completed-upload" (Section 7.2) in the response.
- If the server has not retained knowledge of the upload resource, a 404 (Not Found) status code can be used.

If the Upload-Complete request header field is set to true, the client intends to transfer the remaining representation data in one request. If the request content was fully received, the upload is marked as complete and the upload resource SHOULD generate the response that matches what the resource, that was targeted by the initial upload creation (Section 4.2), would have generated if it had received the entire representation in the initial request. However, the response MUST include the Upload-Complete header field with a true value, allowing clients to identify whether a response, in particular error responses, is related to the resumable upload itself or the processing of the upload representation.

If the Upload-Complete request header field is set to false, the client intends to transfer the remaining representation over multiple requests. If the request content was fully received, the upload resource acknowledges the appended data by sending a 2xx (Successful) response with the Upload-Complete header field set to false.

The server MUST record the length according to Section 4.1.3 if the Upload-Length or Upload-Complete header fields are included in the request. If the length is known, the server MUST prevent the offset from exceeding the upload length by rejecting the request once the offset exceeds the length, marking the upload resource invalid and rejecting any further interaction with it. It is not sufficient to rely on the Content-Length header field for enforcement because the header field might not be present.

While the request content is being received, the server SHOULD send interim responses with a 104 (Upload Resumption Supported) status code and the Upload-Offset header field set to the current offset to inform the client about the upload progress. These interim responses MUST NOT include the Location header field.

4.4.3. Examples

A) The following example shows an upload append request. The client transfers the next 23456789 bytes at an offset of 23456789 and does not indicate that the upload is then completed. The server generates one interim response and finally acknowledges the new offset:

```
PATCH /upload/37a504d87 HTTP/1.1
Host: example.com
Upload-Complete: ?0
Upload-Offset: 23456789
Content-Length: 23456789
Content-Type: application/partial-upload
```

```
[content (23456789 bytes)]
```

```
HTTP/1.1 104 Upload Resumption Supported
Upload-Offset: 35185184
```

```
HTTP/1.1 204 No Content
Upload-Complete: ?0
```

B) The next example shows an upload append, where the client transfers the remaining 4567890 bytes and completes the upload with a length of 1234567890 bytes. The server processes the uploaded representation and generates the responding response, in this example containing extracted meta data:

```
PATCH /upload/d38d6ffe8 HTTP/1.1
Host: example.com
Upload-Complete: ?1
Upload-Offset: 1230000000
Content-Length: 4567890
Content-Type: application/partial-upload
```

```
[content (4567890 bytes)]
```

```
HTTP/1.1 200 OK
Upload-Complete: ?1
Content-Type: application/json
```

```
{
  "metadata": {
    [...]
  }
}
```

4.5. Upload Cancellation

4.5.1. Client Behavior

If the client wants to terminate the transfer without the ability to resume, it can send a DELETE request to the upload resource. Doing so is an indication that the client is no longer interested in continuing the upload, and that the server can release any resources associated with it.

4.5.2. Server Behavior

Upon receiving a DELETE request, the server SHOULD deactivate the upload resource.

The server SHOULD terminate any in-flight requests to the upload resource before sending the response by abruptly terminating their HTTP connection(s) or stream(s) as described in Section 4.6.

The resource SHOULD NOT generate a response with the 301 (Moved Permanently) and 302 (Found) status codes because clients might follow the redirect without preserving the DELETE method.

4.5.3. Example

The following example shows an upload cancellation:

```
DELETE /upload/5688a431c HTTP/1.1
Host: example.com
```

```
HTTP/1.1 204 No Content
```

4.6. Concurrency

Resumable uploads, as defined in this document, do not permit uploading representation data in parallel to the same upload resource. The client MUST NOT perform multiple representation data transfers for the same upload resource in parallel.

Even if the client is well-behaved and doesn't send concurrent requests, network interruptions can occur in such a way that the client considers a request as failed while the server is unaware of the problem and considers the request still ongoing. The client might then try to resume the upload with the best intentions, resulting in concurrent requests from the server's perspective. Therefore, the server MUST take measures to prevent race conditions, data loss and corruption from concurrent requests to append representation data (Section 4.4) and/or cancellation (Section 4.5)

to the same upload resource. In addition, the server MUST NOT send outdated information in responses when retrieving the offset (Section 4.3). This means that the offset sent by the server MUST be accepted in a subsequent request to append representation data if no other request to append representation data or cancel was received in the meantime. In other words, clients have to be able to use received offsets.

The RECOMMENDED approach is as follows: If an upload resource receives a new request to retrieve the offset (Section 4.3), append representation data (Section 4.4), or cancel the upload (Section 4.5) while a previous request for creating the upload (Section 4.2) or appending representation data (Section 4.4) is still ongoing, the resource SHOULD prevent race conditions, data loss, and corruption by terminating the previous request before processing the new request. Due to network delay and reordering, the resource might still be receiving representation data from an ongoing transfer for the same upload resource, which in the client's perspective has failed. Since the client is not allowed to perform multiple transfers in parallel, the upload resource can assume that the previous attempt has already failed. Therefore, the server MAY abruptly terminate the previous HTTP connection or stream.

Since implementing this approach is not always technically possible or feasible, other measures can be considered as well. A simpler approach is that the server only processes a new request to retrieve the offset (Section 4.3), append representation data (Section 4.4), or cancellation (Section 4.5) once all previous requests have been processed. This effectively implements exclusive access to the upload resource through an access lock. However, since network interruptions can occur in ways that cause the request to hang from the server's perspective, it might take the server significant time to realize the interruption and time out the request. During this period, the client will be unable to access the resource and resume the upload, causing friction for the end users. Therefore, the recommended approach is to terminate previous requests to enable quick resumption of uploads.

5. Status Code 104 (Upload Resumption Supported)

The 104 (Upload Resumption Supported) status code is can be used for two purposes:

- * When responding to requests to create uploads, an interim response with the 104 (Upload Resumption Supported) status code can be sent to indicate the server's support for resumable uploads, as well as the URI and limits of the corresponding upload resource in the Location and Upload-Limit header fields, respectively (see Section 4.2). This notifies the client early about the ability to resume the upload in case of network interruptions.
- * While processing the content of a request to append representation data or create an upload, the server can regularly send interim responses with the 104 (Upload Resumption Supported) status code to indicate the current upload progress in the Upload-Offset header field (see Section 4.2 and Section 4.4). This allows the client to show more accurate progress information about the amount of data received by the server. In addition, clients can use this information to release representation data that was buffered, knowing that it doesn't have to be re-transmitted.

When creating or appending resumable uploads, the client can generate a 100-continue expectation because it wants an indication that the server is willing to accept the upload. The client MAY treat an interim response with the 104 (Upload Resumption Supported) status code as fulfilling the 100-continue expectation and start sending the request content. However, the server MUST NOT omit the 100 (Continue) response because it has sent an interim response with the 104 (Upload Resumption Supported) status code.

6. Media Type application/partial-upload

The application/partial-upload media type describes a contiguous block from the representation data that should be uploaded to a resource. There is no minimum block size and the block might be empty. The block can be a subset of the representation data, where the start and/or end of the block don't line up with the start and/or end of the representation data respectively.

7. Problem Types

7.1. Mismatching Offset

This section defines the "https://iana.org/assignments/http-problem-types#mismatching-upload-offset" problem type [PROBLEM]. A server can use this problem type when responding to an upload append request (Section 4.4) to indicate that the Upload-Offset header field in the request does not match the upload resource's offset.

Two problem type extension members are defined: the `expected-offset` and `provided-offset` members. A response using this problem type SHOULD populate both members, with the value of `expected-offset` taken from the upload resource and the value of `provided-offset` taken from the upload append request.

The following example shows an example response, where the resource's offset was 12500000, but the client attempted to append at offset 25000000:

NOTE: '\ ' line wrapping per RFC 8792

HTTP/1.1 409 Conflict
Content-Type: application/problem+json

```
{
  "type": "https://iana.org/assignments/http-problem-types#\
    mismatching-upload-offset",
  "title": "offset from request does not match offset of resource",
  "expected-offset": 12500000,
  "provided-offset": 25000000
}
```

7.2. Completed Upload

This section defines the `"https://iana.org/assignments/http-problem-types#completed-upload"` problem type [PROBLEM]. A server can use this problem type when responding to an upload append request (Section 4.4) to indicate that the upload has already been completed and cannot be modified.

The following example shows an example response:

NOTE: '\ ' line wrapping per RFC 8792

HTTP/1.1 400 Bad Request
Content-Type: application/problem+json

```
{
  "type": "https://iana.org/assignments/http-problem-types#\
    completed-upload",
  "title": "upload is already completed"
}
```

7.3. Inconsistent Length

This section defines the "https://iana.org/assignments/http-problem-types#inconsistent-upload-length" problem type [PROBLEM]. A server can use this problem type when responding to an upload creation (Section 4.2) or upload append request (Section 4.4) to indicate that the request includes inconsistent upload length values, as described in Section 4.1.3.

The following example shows an example response:

NOTE: '\ ' line wrapping per RFC 8792

HTTP/1.1 400 Bad Request

Content-Type: application/problem+json

```
{
  "type": "https://iana.org/assignments/http-problem-types#\
    inconsistent-upload-length",
  "title": "inconsistent length values for upload"
}
```

8. Content Codings

Since the codings listed in Content-Encoding are a characteristic of the representation (see Section 8.4 of [HTTP]), both the client and the server always compute the values for Upload-Offset and optionally Upload-Length on the content coded data (that is, the representation data). Moreover, the content codings are retained throughout the entire upload, meaning that the server is not required to decode the representation data to support resumable uploads. See Appendix A of [DIGEST-FIELDS] for more information.

9. Transfer Codings

Unlike Content-Encoding (see Section 8.4.1 of [HTTP]), Transfer-Encoding (see Section 6.1 of [HTTP/1.1]) is a property of the message, not of the representation. Moreover, transfer codings can be applied in transit (e.g., by proxies). This means that a client does not have to consider the transfer codings to compute the upload offset, while a server is responsible for transfer decoding the message before computing the upload offset. The same applies to the value of Upload-Length. Please note that the Content-Length header field cannot be used in conjunction with the Transfer-Encoding header field.

10. Integrity Digests

The integrity of an entire upload or individual upload requests can be verifying using digests from [DIGEST-FIELDS].

10.1. Representation Digests

Representation digests help verify the integrity of the entire representation data that has been uploaded so far, which might stretch across multiple requests.

If the client knows the integrity digest of the entire representation data before creating an upload resource, it can include the Repr-Digest header field when creating an upload (Section 4.2). Once the upload is completed, the server can compute the integrity digest of the received representation data and compare it to the provided digest. If the digests don't match, the server SHOULD consider the upload failed, not process the representation further, and signal the failure to the client. This way, the integrity of the entire representation data can be protected.

Alternatively, when creating an upload (Section 4.2), the client can ask the server to compute and return the integrity digests using a Want-Repr-Digest field conveying the preferred algorithms. The response SHOULD include at least one of the requested digests, but might not include it. The server SHOULD compute the representation digests using the preferred algorithms once the upload is complete and include the corresponding Repr-Digest header field in the response. Alternatively, the server can compute the digest continuously during the upload and include the Repr-Digest header field in responses to upload creation (Section 4.2) and upload appending requests (Section 4.4) even when the upload is not completed yet. This allows the client to simultaneously compute the digest of the transmitted representation data, compare its digest to the server's digest, and spot data integrity issues. If an upload is spread across multiple requests, data integrity issues can be found even before the upload is fully completed.

10.2. Content Digests

Content digests help verify the integrity of the content in an individual request.

If the client knows the integrity digest of the content from an upload creation (Section 4.2) or upload appending (Section 4.4) request, it can include the Content-Digest header field in the request. Once the content has been received, the server can compute the integrity digest of the received content and compare it to the

provided digest. If the digests don't match the server SHOULD consider the transfer failed, not append the content to the upload resource, and signal the failure to the client. This way, the integrity of an individual request can be protected.

11. Responses to Uploads

HTTP uploads often not only transfer a representation to the server but also send back information to the client. For resumable uploads, this works similarly to conventional HTTP uploads. The server can include information in the response to the request that transferred the remaining representation data and included the Upload-Complete: ?1 header field. Clients can regard this response as the final response for the entire upload, as detailed in Section 4.2 and Section 4.4.

However, due to network interruptions, the upload resource might receive the remaining representation data, complete the upload, and send a response to the client, which then does not receive the response. The client can learn that the upload is complete by retrieving its state (Section 4.3), but resumable uploads as defined in this document do not offer functionality to recover the missed response.

To address this issue, the server can send the desired information in header fields, which are included in both the final response and responses when the client fetches the upload state via a HEAD request (Section 4.3). This way, the client can retrieve the information from the header even if it failed to receive the final response. If the piece of information is too large for header fields, the server could make it available as a separate resource for retrieval and point the client to its URI in an appropriate header field.

12. Upload Strategies

The definition of the upload creation request (Section 4.2) provides the client with flexibility to choose whether the representation data is fully or partially transferred in the first request, or if no representation data is included at all. Which behavior is best largely depends on the client's capabilities, its intention to avoid data re-transmission, and its knowledge about the server's support for resumable uploads.

The following subsections describe two typical upload strategies that are suited for common environments. Note that these modes are never explicitly communicated to the server and clients are not required to stick to one strategy, but can mix and adapt them to their needs.

12.1. Optimistic Upload Creation

An "optimistic upload creation" can be used independent of the client's knowledge about the server's support for resumable uploads. However, the client must be capable of handling and processing interim responses. An upload creation request then includes the full representation data because the client anticipates that it will be transferred without interruptions or resumed if an interruption occurs.

The benefit of this method is that if the upload creation request succeeded, the representation data was transferred in a single request without additional round trips.

A possible drawback is that the client might be unable to resume an upload. If an upload is interrupted before the client receives a 104 (Upload Resumption Supported) interim response with the upload resource's URI, the client cannot resume that upload due to the missing URI. The interim response might not be received if the interruption happens too early in the message exchange, the server does not support resumable uploads at all, the server does not support sending the 104 (Upload Resumption Supported) interim response, or an intermediary dropped the interim response. Without a 104 response, the client needs to either treat the upload as failed or retry the entire upload creation request if this is allowed by the application.

A client might wait for a limited duration to receive a 104 (Upload Resumption Supported) interim response before starting to transmit the request content. This way, the client can learn about the resource's support for resumable uploads and/or the upload resource's URI. This is conceptually similar to how a client might wait for a 100 (Continue) interim response (see Section 10.1.1 of [HTTP]) before committing to work.

12.1.1. Upgrading To Resumable Uploads

Optimistic upload creation allows clients and servers to automatically upgrade non-resumable uploads to resumable ones. In a non-resumable upload, the representation is transferred in a single request, usually POST or PUT, without any ability to resume from interruptions. The client can invite the server to upgrade such a request to a resumable upload by adding the Upload-Complete: ?1 header field to the original request. The Upload-Length header field SHOULD be added if the representation data's length is known upfront. The request is not changed otherwise.

A server that supports resumable uploads at the target URI can create an upload resource and send its URI in a 104 (Upload Resumption Supported) interim response for the client to resume the upload after interruptions. A server that does not support resumable uploads or does not want to upgrade to a resumable upload for this request ignores the Upload-Complete: ?1 header. The transfer then falls back to a non-resumable upload without additional cost.

This upgrade can also be performed transparently by a library or program that acts as a HTTP client by sending requests on behalf of a user. When the user instructs the client to send a non-resumable request, the client can perform the upgrade transparently and handle potential interruptions and resumptions under the hood without involving the user. The last response received by the client is considered the response for the entire upload and should be provided to the user.

12.2. Careful Upload Creation

For a "careful upload creation" the client knows that the server supports resumable uploads and sends an empty upload creation request without including any representation data. Upon successful response reception, the client can use the included upload resource URI to transmit the representation data (Section 4.4) and resume the upload at any stage if an interruption occurs. The client should inspect the response for the Upload-Limit header field, which would indicate limits applying to the remaining upload procedure.

The retransmission of representation data or the ultimate upload failure that can happen with an "optimistic upload creation" is therefore avoided at the expense of an additional request that does not carry representation data.

This approach is best suited if the client cannot receive interim responses, e.g. due to a limitation in the provided HTTP interface, or if large representations are transferred where the cost of the additional request is minuscule compared to the effort of transferring the representation itself.

13. Security Considerations

The upload resource URI is the identifier used for modifying the upload. Without further protection of this URI, an attacker may obtain information about an upload, append data to it, or cancel it. To prevent this, the server SHOULD ensure that only authorized clients can access the upload resource. To reduce the risk of unauthorized access, it is RECOMMENDED to generate upload resource URIs in such a way that makes it hard to be guessed by unauthorized

clients. In addition, servers may embed information about the storage or processing location of the uploaded representation in the upload resource URI to make routing requests more efficient. If so, they **MUST** ensure that no internal information is leaked in the URI that is not intended to be exposed.

Uploaded representation data and its metadata are untrusted input. Server operators have to be careful of where uploaded data is written and subsequently accessed, especially if the operations cause the representation to be processed or executed by the server. In addition, metadata **MUST** be validated and/or sanitized if the server takes its values into consideration for processing or storing the representation.

Some servers or intermediaries provide scanning of content uploaded by clients. Any scanning mechanism that relies on receiving a complete representation in a single request message can be defeated by resumable uploads because content can be split across multiple messages. Servers or intermediaries wishing to perform content scanning **SHOULD** consider how resumable uploads can circumvent scanning and take appropriate measures. Possible strategies include waiting for the upload to complete before scanning the entire representation, or disabling resumable uploads.

There can be a significant delay between the creation of an upload resource and its completion. Policy decisions or authorization checks performed on the initial request might become outdated or invalid by the time the upload completes. To mitigate vulnerabilities arising from time-of-check to time-of-use (TOCTOU) conditions, the server **SHOULD** validate that the user is still allowed to perform the requested action before finalizing the upload. This includes, for example, validating access privileges and quota policies associated with the upload resource.

Resumable uploads are vulnerable to Slowloris-style attacks [SLOWLORIS]. A malicious client may create upload resources and keep them alive by regularly sending PATCH requests with no or small content to the upload resources. This could be abused to exhaust server resources by creating and holding open uploads indefinitely with minimal work. Servers **SHOULD** provide mitigations for Slowloris attacks, such as increasing the maximum number of clients the server will allow, limiting the number of uploads a single client is allowed to make, imposing restrictions on the minimum transfer speed an upload is allowed to have, and restricting the length of time an upload resource can exist.

Uploads performed as a series of appends can be used to upload data up to the max-size limit, which could be a larger size than a server or intermediary might normally permit in conventional single upload request message content. Servers or intermediaries need to consider that relying solely on message content limits to constrain resources allocated to uploads might not be an effective strategy when using resumable uploads.

14. IANA Considerations

14.1. HTTP Fields

IANA is asked to register the following entries in the "Hypertext Transfer Protocol (HTTP) Field Name Registry":

Field Name	Status	Structured Type	Reference
Upload-Offset	permanent	Item	Section 4.1.1 of this document
Upload-Complete	permanent	Item	Section 4.1.2 of this document
Upload-Length	permanent	Item	Section 4.1.3 of this document
Upload-Limit	permanent	Dictionary	Section 4.1.4 of this document

Table 1

14.2. HTTP Status Code

IANA is asked to register the following entry in the "HTTP Status Codes" registry:

Value: 104 (suggested value)

Description: Upload Resumption Supported

Specification: Section 5 of this document

14.3. Media Type

IANA is asked to register the following entry in the "Media Types" registry:

Type name: application

Subtype name: partial-upload

Required parameters: N/A

Optional parameters: N/A

Encoding considerations: binary

Security considerations: see Section 13 of this document

Interoperability considerations: N/A

Published specification: Section 6 of this document

Applications that use this media type: Applications that transfer files over unreliable networks or want pause- and resumable uploads.

Fragment identifier considerations: N/A

Additional information:

* Deprecated alias names for this type: N/A

* Magic number(s): N/A

* File extension(s): N/A

* Macintosh file type code(s): N/A

* Windows Clipboard Name: N/A

Person and email address to contact for further information: See the Authors' Addresses section of this document.

Intended usage: COMMON

Restrictions on usage: N/A

Author: See the Authors' Addresses section of this document.

Change controller: IETF

14.4. HTTP Problem Types

IANA is asked to register the following entry in the "HTTP Problem Types" registry:

Type URI: <https://iana.org/assignments/http-problem-types#mismatching-upload-offset>

Title: Mismatching Upload Offset

Recommended HTTP status code: 409

Reference: Section 7.1 of this document

IANA is asked to register the following entry in the "HTTP Problem Types" registry:

Type URI: <https://iana.org/assignments/http-problem-types#completed-upload>

Title: Upload Is Completed

Recommended HTTP status code: 400

Reference: Section 7.2 of this document

IANA is asked to register the following entry in the "HTTP Problem Types" registry:

Type URI: <https://iana.org/assignments/http-problem-types#inconsistent-upload-length>

Title: Inconsistent Upload Length Values

Recommended HTTP status code: 400

Reference: Section 7.3 of this document

15. References

15.1. Normative References

[CACHING] Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Caching", STD 98, RFC 9111, DOI 10.17487/RFC9111, June 2022, <<https://www.rfc-editor.org/rfc/rfc9111>>.

[CONTENT-DISPOSITION]

Reschke, J., "Use of the Content-Disposition Header Field in the Hypertext Transfer Protocol (HTTP)", RFC 6266, DOI 10.17487/RFC6266, June 2011, <<https://www.rfc-editor.org/rfc/rfc6266>>.

[DIGEST-FIELDS]

Polli, R. and L. Pardue, "Digest Fields", RFC 9530, DOI 10.17487/RFC9530, February 2024, <<https://www.rfc-editor.org/rfc/rfc9530>>.

[HTTP]

Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP Semantics", STD 97, RFC 9110, DOI 10.17487/RFC9110, June 2022, <<https://www.rfc-editor.org/rfc/rfc9110>>.

[HTTP/1.1]

Fielding, R., Ed., Nottingham, M., Ed., and J. Reschke, Ed., "HTTP/1.1", STD 99, RFC 9112, DOI 10.17487/RFC9112, June 2022, <<https://www.rfc-editor.org/rfc/rfc9112>>.

[PATCH]

Dusseault, L. and J. Snell, "PATCH Method for HTTP", RFC 5789, DOI 10.17487/RFC5789, March 2010, <<https://www.rfc-editor.org/rfc/rfc5789>>.

[PROBLEM]

Nottingham, M., Wilde, E., and S. Dalal, "Problem Details for HTTP APIs", RFC 9457, DOI 10.17487/RFC9457, July 2023, <<https://www.rfc-editor.org/rfc/rfc9457>>.

[RFC2119]

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

[RFC8174]

Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/rfc/rfc8174>>.

[STRUCTURED-FIELDS]

Nottingham, M. and P. Kamp, "Structured Field Values for HTTP", RFC 9651, DOI 10.17487/RFC9651, September 2024, <<https://www.rfc-editor.org/rfc/rfc9651>>.

15.2. Informative References

[RFC8792]

Watsen, K., Auerswald, E., Farrel, A., and Q. Wu, "Handling Long Lines in Content of Internet-Drafts and RFCs", RFC 8792, DOI 10.17487/RFC8792, June 2020, <<https://www.rfc-editor.org/rfc/rfc8792>>.

[SLOWLORIS]

"RSnake" Hansen, R., "Welcome to Slowloris - the low bandwidth, yet greedy and poisonous HTTP client!", June 2009, <<https://web.archive.org/web/20150315054838/http://ha.ckers.org/slowloris/>>.

Appendix A. Changes

This section is to be removed before publishing as an RFC.

Since draft-ietf-httpbis-resumable-upload-10

- * Add recommended disposition type for file name indication.

Since draft-ietf-httpbis-resumable-upload-09

- * Requires Accept-Patch in OPTIONS.
- * Add security consideration regarding time-of-check to time-of-use.
- * Lift requirement on Upload-Complete for all final responses.
- * Relax requirements on limit changes.
- * Describe the interaction between 100 and 104 responses.
- * Numerous editorial improvements.

Since draft-ietf-httpbis-resumable-upload-08

- * Clarify definitions of new header fields.
- * Make handling of OPTIONS * optional.
- * Require server to announce limits using Upload-Limit.
- * Require clients to adhere to known limits.
- * Rephrase requirements for concurrency handling, focusing on the outcome.
- * Remove requirement for 204 status code for DELETE responses.
- * Increase the draft interop version.
- * Add section about 104 status code.
- * Rephrase recommendation for sending information back to client.

Since draft-ietf-httpbis-resumable-upload-07

- * Clarify server handling when upload length is exceeded.
- * Extend security considerations about upload resource URIs, representation metadata, and untrusted inputs.
- * Allow clients to retry for appropriate 4xx responses.

Since draft-ietf-httpbis-resumable-upload-06

- * Minor editorial improvements to introduction and examples.
- * Define structured types for new header fields.

Since draft-ietf-httpbis-resumable-upload-05

- * Increase the draft interop version.
- * Numerous editorial changes.
- * Rename expires limit to max-age.
- * Require Upload-Complete, but not Upload-Offset or Upload-Limit, for append responses.
- * Add problem type for inconsistent length values.
- * Reduce use of "file" in favor of "representation".

Since draft-ietf-httpbis-resumable-upload-04

- * Clarify implications of Upload-Limit header.
- * Allow client to fetch upload limits upfront via OPTIONS.
- * Add guidance on upload creation strategy.
- * Add Upload-Length header to indicate length during creation.
- * Describe possible usage of Want-Repr-Digest.

Since draft-ietf-httpbis-resumable-upload-03

- * Add note about Content-Location for referring to subsequent resources.
- * Require application/partial-upload for appending to uploads.

- * Explain handling of content and transfer codings.
- * Add problem types for mismatching offsets and completed uploads.
- * Clarify that completed uploads must not be appended to.
- * Describe interaction with Digest Fields from RFC9530.
- * Require that upload offset does not decrease over time.
- * Add Upload-Limit header field.
- * Increase the draft interop version.

Since draft-ietf-httpbis-resumable-upload-02

- * Add upload progress notifications via informational responses.
- * Add security consideration regarding request filtering.
- * Explain the use of empty requests for creation uploads and appending.
- * Extend security consideration to include resource exhaustion attacks.
- * Allow 200 status codes for offset retrieval.
- * Increase the draft interop version.

Since draft-ietf-httpbis-resumable-upload-01

- * Replace Upload-Incomplete header with Upload-Complete.
- * Replace terminology about procedures with HTTP resources.
- * Increase the draft interop version.

Since draft-ietf-httpbis-resumable-upload-00

- * Remove Upload-Token and instead use Server-generated upload URL for upload identification.
- * Require the Upload-Incomplete header field in Upload Creation Procedure.
- * Increase the draft interop version.

Since draft-tus-httpbis-resumable-uploads-protocol-02

None

Since draft-tus-httpbis-resumable-uploads-protocol-01

- * Clarifying backtracking and preventing skipping ahead during the Offset Receiving Procedure.
- * Clients auto-retry 404 is no longer allowed.

Since draft-tus-httpbis-resumable-uploads-protocol-00

- * Split the Upload Transfer Procedure into the Upload Creation Procedure and the Upload Appending Procedure.

Appendix B. Draft Version Identification

This section is to be removed before publishing as an RFC.

To assist the development of implementations and interoperability testing while this document is still a draft, an interop version is defined. Implementations of this draft use the interop version to identify the iteration of the draft that they implement. The interop version is bumped for breaking changes.

The current interop version is 8.

Client implementations of draft versions of the protocol MUST send a header field Upload-Draft-Interop-Version with the interop version as its value to its requests. The Upload-Draft-Interop-Version field value is an Integer.

Server implementations of draft versions of the protocol MUST NOT send a 104 (Upload Resumption Supported) informational response when the interop version indicated by the Upload-Draft-Interop-Version header field in the request is missing or mismatching.

Server implementations of draft versions of the protocol MUST also send a header field Upload-Draft-Interop-Version with the interop version as its value to the 104 (Upload Resumption Supported) informational response.

Client implementations of draft versions of the protocol MUST ignore a 104 (Upload Resumption Supported) informational response with missing or mismatching interop version indicated by the Upload-Draft-Interop-Version header field.

The reason both the client and the server are sending and checking the draft version is to ensure that implementations of the final RFC will not accidentally interop with draft implementations, as they will not check the existence of the Upload-Draft-Interop-Version header field.

Acknowledgments

This document is based on an Internet-Draft specification written by Jiten Mehta, Stefan Matsson, and the authors of this document.

The tus v1 protocol (<https://tus.io/>) is a specification for a resumable file upload protocol over HTTP. It inspired the early design of this protocol. Members of the tus community helped significantly in the process of bringing this work to the IETF.

The authors would like to thank Mark Nottingham for substantive contributions to the text, along with the following in alphabetical order for their thorough reviews of the document:

- * Mike Bishop
- * Roy T. Fielding
- * Julian Reschke
- * Glenn Strauss
- * Willy Tarreau

Authors' Addresses

Marius Kleidl (editor)
Transloadit
Email: marius@transloadit.com

Guoye Zhang (editor)
Apple Inc.
Email: guoye_zhang@apple.com

Lucas Pardue (editor)
Cloudflare
Email: lucas@lucaspardue.com