Workgroup: HTTP
Internet-Draft:
draft-tus-httpbis-resumable-uploads-protocol-01
Published: 16 June 2022
Intended Status: Standards Track
Expires: 18 December 2022

Authors: M. Kleidl — Transloadit Ltd
J. Mehta — Apple Inc.
G. Zhang — Apple Inc.
L. Pardue — Cloudflare
S. Matsson — JellyHive

tus - Resumable Uploads Protocol

Abstract

HTTP clients often encounter interrupted data transfers as a result of canceled requests or dropped connections. Prior to interruption, part of a representation may have been exchanged. To complete the data transfer of the entire representation, it is often desirable to issue subsequent requests that transfer only the remainder of the representation. 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.

Discussion Venues

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

Discussion of this document takes place on the tus-v2 GitHub repository at https://github.com/tus/tus-v2.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

HTTP clients often encounter interrupted data transfers as a result of canceled requests or dropped connections. Prior to interruption, part of a representation (see Section 3.2 of [HTTP]) might have been exchanged. To complete the data transfer of the entire representation, it is often desirable to issue subsequent requests that transfer only the remainder of the representation. HTTP range requests (see Section 14 of [HTTP]) support this concept of resumable downloads from server to client.

HTTP methods such as POST or PUT can be used by clients to request processing of representation data enclosed in the request message. The transfer of representation data from client to server is often referred to as an upload. Uploads are just as likely as downloads to suffer from the effects of data transfer interruption. Humans can play a role in upload interruptions through manual actions such as pausing an upload. Regardless of the cause of an interruption, servers may have received part of the representation before its occurrence and it is desirable if clients can complete the data transfer by sending only the remainder of the representation. The process of sending additional parts of a representation using subsequent HTTP requests from client to server is herein referred to as a resumable upload.

Connection interruptions are common and the absence of a standard mechanism for resumable uploads has lead to a proliferation of custom solutions. Some of those use HTTP, while others rely on other transfer mechanisms entirely. An HTTP-based standard solution is desirable for such a common class of problem.

This document defines the Resumable Uploads Protocol, an optional mechanism for resumable uploads using HTTP that is backwards-compatible with conventional HTTP uploads. When an upload is interrupted, clients can send subsequent requests to query the server state and use this information to the send remaining data. Alternatively, they can cancel the upload entirely.

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.

The terms byte sequence, Item, string, sf-binary, sf-boolean, sf-integer, sf-string, and sf-token are imported from [STRUCTURED-FIELDS].

The terms client and server are imported from [HTTP].

Upload: A sequence of one or more procedures, uniquely identified by a token chosen by a client.

Procedure: An HTTP message exchange for that can be used for resumable uploads.

3. Uploading Overview

The Resumable Uploads Protocol consists of several procedures that rely on HTTP message exchanges. The following procedures are defined:

*Upload Creation Procedure (Section 4)

*Offset Retrieving Procedure (Section 5)

*Upload Appending Procedure (Section 6)

*Upload Cancellation Procedure (Section 7)

A single upload is a sequence of one or more procedures. Each upload is uniquely identified by a token chosen by a client. The token is carried in the Upload-Token header field; see Section 9.1.

The remainder of this section uses examples of a file upload to illustrate permutations of procedure sequence. Note, however, that HTTP message exchanges use representation data (see Section 8.1 of [HTTP]), which means that procedures can apply to many forms of content.

3.1. Example 1: Complete upload of file with known size

In this example, the client first attempts to upload a file with a known size in a single HTTP request. An interruption occurs and the client then attempts to resume the upload using subsequent HTTP requests.

1) The Upload Creation Procedure (Section 4) can be used to notify the server that the client wants to begin an upload. The server should then reserve the required resources to accept the upload from the client. The client also begins transferring the entire file in the request body. The request includes the Upload-Token header, which is used for identifying future requests related to this upload. An informational response can be sent to the client to signal the support of resumable upload on the server.

```
Client                                  Server
|                                            |
| POST with Upload-Token                     |
|------------------------------------------->|
|                                            |
|                                            | Reserve resources
|                                            | for Upload-Token
|                                            |------------------
|                                            |                 |
|                                            |<-----------------
|                                            |
|            104 Upload Resumption Supported |
|<-------------------------------------------|
|                                            |
| Flow Interrupted                           |
|------------------------------------------->|
|                                            |
```

Figure 1: Upload Creation Procedure

2) If the connection to the server gets interrupted during the Upload Creation Procedure, the client may want to resume the upload. Before this is possible, the client must know the amount of data that the server was able to receive before the connection got interrupted. To achieve this, the client uses the Offset Retrieving Procedure (Section 5) to obtain the upload's offset.

```
Client                                      Server
|                                                |
| HEAD with Upload-Token                         |
|----------------------------------------------->|
|                                                |
|              204 No Content with Upload-Offset |
|<-----------------------------------------------|
|                                                |
```

Figure 2: Offset Retrieving Procedure

3) After the Offset Retrieving Procedure (Section 5) completes, the client can resume the upload by sending the remaining file content to the server using the Upload Appending Procedure (Section 6), appending to the already stored data in the upload. The Upload-Offset value is included to ensure that the client and server agree on the offset that the upload resumes from.

```
Client                                          Server
|                                                    |
| PATCH with Upload-Token and Upload-Offset          |
|--------------------------------------------------->|
|                                                    |
|                      201 Created on completion     |
|<---------------------------------------------------|
|                                                    |
```

Figure 3: Upload Appending Procedure

4) If the client is not interested in completing the upload anymore, it can instruct the server to delete the upload and free all related resources using the Upload Cancellation Procedure (Section 7).

```
Client                                          Server
|                                                    |
| DELETE with Upload-Token                           |
|--------------------------------------------------->|
|                                                    |
|                   204 No Content on completion     |
|<---------------------------------------------------|
|                                                    |
```

Figure 4: Upload Cancellation Procedure

3.2. Example 2: Upload as a series of parts

In some cases clients might prefer to upload a file as a series of parts sent 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, such as when file data originates from a streaming source.

This example shows how the client, with prior knowledge about the server's resumable upload support, can upload parts of a file over a sequence of procedures.

1) If the client is aware that the server supports resumable upload, it can use the Upload Creation Procedure with the Upload-Incomplete header to start an upload.

```
Client                                      Server
|                                                |
| POST with Upload-Token, Upload-Offset,         |
| and Upload-Incomplete                          |
|----------------------------------------------->|
|                                                |
|             201 Created with Upload-Incomplete |
|              on completion                     |
|<-----------------------------------------------|
|                                                |
```

Figure 5: Upload Creation Procedure Incomplete

2) After creation, the following parts are sent using the Upload Appending Procedure (Section 6), and the last part of the upload does not have the Upload-Incomplete header.

```
Client                                      Server
|                                                |
| PATCH with Upload-Token and Upload-Offset      |
|----------------------------------------------->|
|                                                |
|                      201 Created on completion |
|<-----------------------------------------------|
|                                                |
```

Figure 6: Upload Appending Procedure Last Chunk

4. Upload Creation Procedure

The Upload Creation Procedure is intended for starting a new upload. A limited form of this procedure **MAY** be used by the client without the knowledge of server support of the Resumable Uploads Protocol.

This procedure is designed to be compatible with a regular upload. Therefore all methods are allowed with the exception of GET, HEAD, DELETE, and OPTIONS. All response status codes are allowed. The client is **RECOMMENDED** to use the POST method if not otherwise intended. The server **MAY** only support a limited number of methods.

The request **MUST** include the Upload-Token header field (Section 9.1) which uniquely identifies an upload. The client **MUST NOT** reuse the token for a different upload. The request **MUST NOT** include the Upload-Offset header.

If the end of the request body is not the end of the upload, the Upload-Incomplete header field (Section 9.3) **MUST** be set to true.

If the server already has an active upload with the same token in the Upload-Token header field, it **MUST** respond with 409 (Conflict) status code.

The server **MUST** send the Upload-Offset header in the response if it considers the upload active, either when the response is a success (e.g. 201 (Created)), or when the response is a failure (e.g. 409 (Conflict)). The value **MUST** be equal to the end offset of the entire upload, or the begin offset of the next chunk if the upload is still incomplete. The client **SHOULD** consider the upload failed if the response status code indicates a success but the offset in the Upload-Offset header field in the response does not equal to the begin offset plus the number of bytes uploaded in the request.

If the request completes successfully and the entire upload is complete, the server **MUST** acknowledge it by responding with a successful status code between 200 and 299 (inclusive). Server is **RECOMMENDED** to use 201 (Created) response if not otherwise specified. The response **MUST NOT** include the Upload-Incomplete header with the value of true.

If the request completes successfully but the entire upload is not yet complete indicated by the Upload-Incomplete header, the server **MUST** acknowledge it by responding with the 201 (Created) status code, the Upload-Incomplete header set to true.

```
:method: POST
:scheme: https
:authority: example.com
:path: /upload
upload-token: :SGVs…SGU=:
upload-draft-interop-version: 2
content-length: 100
[content (100 bytes)]

:status: 104
upload-draft-interop-version: 2

:status: 201
upload-offset: 100
```

```
:method: POST
:scheme: https
:authority: example.com
:path: /upload
upload-token: :SGVs…SGU=:
upload-draft-interop-version: 2
upload-offset: 0
upload-incomplete: ?1
content-length: 25
[partial content (25 bytes)]

:status: 201
upload-incomplete: ?1
upload-offset: 25
```

The client **MAY** automatically attempt upload resumption when the connection is terminated unexpectedly, or if a server error status code between 500 and 599 (inclusive) is received. The client **SHOULD NOT** automatically retry if a client error status code between 400 and 499 (inclusive) is received.

File metadata can affect how servers might act on the uploaded file. Clients can send Representation Metadata (see Section 8.3 of [HTTP]) in the Upload Creation Procedure request that starts an upload. Servers **MAY** interpret this metadata or **MAY** ignore it. The Content-Type header can be used to indicate the MIME type of the file. The Content-Disposition header can be used to transmit a filename. If included, the parameters **SHOULD** be either filename, filename* or boundary.

4.1. Feature Detection

If the client has no knowledge of whether the server supports resumable upload, the Upload Creation Procedure **MAY** be used with some additional constraints. In particular, the Upload-Incomplete header field (Section 9.3) **MUST NOT** be sent in the request if the server support is unclear. This allows the upload to function as if it is a regular upload.

If the server detects the Upload Creation Procedure and it supports resumable upload, an informational response with 104 (Upload Resumption Supported) status **MAY** be sent to the client while the request body is being uploaded.

The client **MUST NOT** attempt to resume an upload if it did not receive the 104 (Upload Resumption Supported) informational response, and it does not have other signals of whether the server supporting resumable upload.

4.2. Draft Version Identification

> **RFC Editor's Note:** Please remove this section and Upload-Draft-Interop-Version from all examples prior to publication of a final version of this document.

The current interop version is 2.

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. Its ABNF is

```
Upload-Draft-Interop-Version = sf-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.

5. Offset Retrieving Procedure

If an upload is interrupted, the client **MAY** attempt to fetch the offset of the incomplete upload by sending a HEAD request to the server with the same Upload-Token header field (Section 9.1). The client **MUST NOT** initiate this procedure without the knowledge of server support.

The request **MUST** use the HEAD method and include the Upload-Token header. The request **MUST NOT** include the Upload-Offset header or the Upload-Incomplete header. The server **MUST** reject the request with the Upload-Offset header or the Upload-Incomplete header by sending a 400 (Bad Request) response.

If the server considers the upload associated with this token active, it **MUST** send back a 204 (No Content) response. The response **MUST** include the Upload-Offset header set to the current resumption offset for the client. The response **MUST** include the Upload-Incomplete header which is set to true if and only if the upload is incomplete. An upload is considered complete if and only if the server completely and successfully received a corresponding Upload Creation Procedure (Section 4) or Upload Appending Procedure (Section 6) request with the Upload-Incomplete header being omitted or set to false.

The client **MUST NOT** perform the Offset Retrieving Procedure (Section 5) while the Upload Creation Procedure (Section 4) or the Upload Appending Procedure (Section 6) is in progress.

The offset **MUST** be accepted by a subsequent Upload Appending Procedure (Section 6). Due to network delay and reordering, the server might still be receiving data from an ongoing transfer for the same token, which in the client perspective has failed. The server **MAY** terminate any transfers for the same token before sending the response by abruptly terminating the HTTP connection or stream. Alternatively, the server **MAY** keep the ongoing transfer alive but ignore further bytes received past the offset.

The client **MUST NOT** start more than one Upload Appending Procedures (Section 6) based on the resumption offset from a single Offset Retrieving Procedure (Section 5).

The response **SHOULD** include Cache-Control: no-store header to prevent HTTP caching.

If the server does not consider the upload associated with this token active, it **MUST** respond with 404 (Not Found) status code.

```
:method: HEAD
:scheme: https
:authority: example.com
:path: /upload
upload-token: :SGVs…SGU=:
upload-draft-interop-version: 2

:status: 204
upload-offset: 100
cache-control: no-store
```

The client **MAY** automatically start uploading from the beginning using Upload Creation Procedure (Section 4) if 404 (Not Found) status code is received. The client **SHOULD NOT** automatically retry if a status code other than 204 and 404 is received.

6. Upload Appending Procedure

The Upload Appending Procedure is used for resuming an existing upload.

The request **MUST** use the PATCH method and include the Upload-Token header. The Upload-Offset header field (Section 9.2) **MUST** be set to the resumption offset.

If the end of the request body is not the end of the upload, the Upload-Incomplete header field (Section 9.3) **MUST** be set to true.

The server **SHOULD** respect representation metadata received in the Upload Creation Procedure (Section 4) and ignore any representation metadata received in the Upload Appending Procedure (Section 6).

If the server does not consider the upload associated with the token in the Upload-Token header field active, it **MUST** respond with 404 (Not Found) status code.

The client **MUST NOT** perform multiple upload transfers for the same token using Upload Creation Procedures (Section 4) or Upload Appending Procedures (Section 6) in parallel to avoid race conditions and data loss or corruption. The server is **RECOMMENDED** to take measures to avoid parallel upload transfers: The server **MAY** terminate any ongoing Upload Creation Procedure (Section 4) or Upload Appending Procedure (Section 6) for the same token. Since the client is not allowed to perform multiple transfers in parallel, the server can assume that the previous attempt has already failed. Therefore, the server **MAY** abruptly terminate the previous HTTP connection or stream.

If the offset in the Upload-Offset header field does not match the offset provided by the immediate previous Offset Retrieving Procedure (Section 5), or the end offset of the immediate previous incomplete transfer, the server **MUST** respond with 409 (Conflict) status code.

The server **MUST** send the Upload-Offset header in the response if it considers the upload active, either when the response is a success (e.g. 201 (Created)), or when the response is a failure (e.g. 409 (Conflict)). The value **MUST** be equal to the end offset of the entire upload, or the begin offset of the next chunk if the upload is still incomplete. The client **SHOULD** consider the upload failed if the response status code indicates a success but the offset in the Upload-Offset header field in the response does not equal to the begin offset plus the number of bytes uploaded in the request.

If the request completes successfully and the entire upload is complete, the server **MUST** acknowledge it by responding with a successful status code between 200 and 299 (inclusive). Server is **RECOMMENDED** to use 201 (Created) response if not otherwise specified. The response **MUST NOT** include the Upload-Incomplete header with the value of true.

If the request completes successfully but the entire upload is not yet complete indicated by the Upload-Incomplete header, the server **MUST** acknowledge it by responding with the 201 (Created) status code, the Upload-Incomplete header set to true.

```
:method: PATCH
:scheme: https
:authority: example.com
:path: /upload
upload-token: :SGVs…SGU=:
upload-offset: 100
upload-draft-interop-version: 2
content-length: 100
[content (100 bytes)]

:status: 201
upload-offset: 200
```

The client **MAY** automatically attempt upload resumption when the connection is terminated unexpectedly, or if a server error status code between 500 and 599 (inclusive) is received. The client **SHOULD NOT** automatically retry if a client error status code between 400 and 499 (inclusive) is received.

7. Upload Cancellation Procedure

If the client wants to terminate the transfer without the ability to resume, it **MAY** send a DELETE request to the server along with the Upload-Token which is an indication that the client is no longer interested in uploading this body and the server can release resources associated with this token. The client **MUST NOT** initiate this procedure without the knowledge of server support.

The request **MUST** use the DELETE method and include the Upload-Token header. The request **MUST NOT** include the Upload-Offset header or the Upload-Incomplete header. The server **MUST** reject the request with the Upload-Offset header or the Upload-Incomplete header by sending a 400 (Bad Request) response.

If the server has successfully deactivated this token, it **MUST** send back a 204 (No Content) response.

The server **MAY** terminate any ongoing Upload Creation Procedure (Section 4) or Upload Appending Procedure (Section 6) for the same token before sending the response by abruptly terminating the HTTP connection or stream.

If the server does not consider the upload associated with this token active, it **MUST** respond with 404 (Not Found) status code.

If the server does not support cancellation, it **MUST** respond with 405 (Method Not Allowed) status code.

```
:method: DELETE
:scheme: https
:authority: example.com
:path: /upload
upload-token: :SGVs…SGU=:
upload-draft-interop-version: 2
```

```
:status: 204
```

8. Request Identification

The Upload Creation Procedure (Section 4) supports arbitrary methods including PATCH, therefore it is not possible to identify the procedure of a request purely by its method. The following algorithm is **RECOMMENDED** to identify the procedure from a request for a generic implementation:

1. The Upload-Token header is not present: Not a resumable upload.

2. The Upload-Offset header is present: Upload Appending Procedure (Section 6).

3. The method is HEAD: Offset Retrieving Procedure (Section 5).

4. The method is DELETE: Upload Cancellation Procedure (Section 7).

5. Otherwise: Upload Creation Procedure (Section 4).

9. Header Fields

9.1. Upload-Token

The Upload-Token request header field is an Item Structured Header (see Section 3.3 of [STRUCTURED-FIELDS]) carrying the token used for identification of a specific upload. Its value **MUST** be a byte sequence. Its ABNF is

```
Upload-Token = sf-binary
```

If not otherwise specified by the server, the client is **RECOMMENDED** to use 256-bit (32 bytes) cryptographically-secure random binary data as the value of the Upload-Token, in order to ensure that it is globally unique and non-guessable.

A conforming implementation **MUST** be able to handle a Upload-Token field value of at least 128 octets.

9.2. Upload-Offset

The Upload-Offset request and response header field is an Item Structured Header indicating the resumption offset of corresponding upload, counted in bytes. Its value **MUST** be an integer. Its ABNF is

```
Upload-Offset = sf-integer
```

9.3. Upload-Incomplete

The Upload-Incomplete request and response header field is an Item Structured Header indicating whether the corresponding upload is considered complete. Its value **MUST** be a boolean. Its ABNF is

```
Upload-Incomplete = sf-boolean
```

10. Redirection

The 301 (Moved Permanently) status code and the 302 (Found) status code **MUST NOT** be used in Offset Retrieving Procedure (Section 5) and Upload Cancellation Procedure (Section 7) responses. A 308 (Permanent Redirect) response **MAY** be persisted for all subsequent procedures. If client receives a 307 (Temporary Redirect) response in the Offset Retrieving Procedure (Section 5), it **MAY** apply the redirection directly in the immediate subsequent Upload Appending Procedure (Section 6).

11. Security Considerations

The tokens inside the Upload-Token header field can be selected by the client which has no knowledge of tokens picked by other client, so uniqueness cannot be guaranteed. If the token is guessable, an attacker can append malicious data to ongoing uploads. To mitigate these issues, 256-bit cryptographically-secure random binary data is recommended for the token.

It is **OPTIONAL** for the server to partition upload tokens based on client identity established through other channels, such as Cookie or TLS client authentication. The client **MAY** relax the token strength if it is aware of server-side partitioning.

12. IANA Considerations

This specification registers the following entry in the Permanent Message Header Field Names registry established by [RFC3864]:

Header field name: Upload-Token, Upload-Offset, Upload-Incomplete

Applicable protocol: http

Status: standard

Author/change controller: IETF

Specification: This document

Related information: n/a

This specification registers the following entry in the "HTTP Status Codes" registry:

Code: 104

Description: Upload Resumption Supported

Specification: This document

13. Normative References

[HTTP] Fielding, R. T., Nottingham, M., and J. Reschke, "HTTP Semantics", Work in Progress, Internet-Draft, draft-ietf-httpbis-semantics-19, 12 September 2021, <https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-semantics-19>.

[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>.

[RFC3864] Klyne, G., Nottingham, M., and J. Mogul, "Registration Procedures for Message Header Fields", BCP 90, RFC 3864, DOI 10.17487/RFC3864, September 2004, <https://www.rfc-editor.org/rfc/rfc3864>.

[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-H. Kamp, "Structured Field Values for HTTP", RFC 8941, DOI 10.17487/RFC8941, February 2021, <https://www.rfc-editor.org/rfc/rfc8941>.

Appendix A. Changes

A.1. draft-tus-httpbis-resumable-uploads-protocol-00

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

Acknowledgments

TODO acknowledge.

Appendix

Informational Response

The server is allowed to respond to Upload Creation Procedure (Section 4) requests with a 104 (Upload Resumption Supported) intermediate response as soon as the server has validated the request. This way, the client knows that the server supports resumable uploads before the complete response for the Upload Creation Procedure is received. The benefit is the clients can defer starting the actual data transfer until the server indicates full support of the incoming Upload Creation Procedure (i.e. resumable are supported, the provided upload token is active etc).

On the contrary, support for intermediate responses (the 1XX range) in existing software is limited or not at all present. Such software includes proxies, firewalls, browsers, and HTTP libraries for clients and server. Therefore, the 104 (Upload Resumption Supported) status code is optional and not mandatory for the successful completion of an upload. Otherwise, it might be impossible in some cases to implement resumable upload servers using existing software packages. Furthermore, as parts of the current internet infrastructure currently have limited support for intermediate responses, a successful delivery of a 104 (Upload Resumption Supported) from the server to the client should be assumed.

We hope that support for intermediate responses increases in the near future, to allow a wider usage of 104 (Upload Resumption Supported).

Feature Detection

This specification includes a section about feature detection (it was called service discovery in earlier discussions, but this name is probably ill-suited). The idea is to allow resumable uploads to be transparently implemented by HTTP clients. This means that application developers just keep using the same API of their HTTP library as they have done in the past with traditional, non-resumable uploads. Once the HTTP library gets updated (e.g. because mobile OS or browsers start implementing resumable uploads), the HTTP library can transparently decide to use resumable uploads without explicit configuration by the application developer. Of course, in order to use resumable uploads, the HTTP library needs to know whether the server supports resumable uploads. If no support is detected, the HTTP library should use the traditional, non-resumable upload technique. We call this process feature detection.

Ideally, the technique used for feature detection meets following **criteria** (there might not be one approach which fits all requirements, so we have to prioritize them):

1. Avoid additional roundtrips by the client, if possible (i.e. an additional HTTP request by the client should be avoided).

2. Be backwards compatible to HTTP/1.1 and existing network infrastructure: This means to avoid using new features in HTTP/2, or features which might require changes to existing network infrastructure (e.g. nginx or HTTP libraries)

3. Conserve the user's privacy (i.e. the feature detection should not leak information to other third-parties about which URLs have been connected to)

Following **approaches** have already been considered in the past. All except the last approaches have not been deemed acceptable and are therefore not included in the specification. This follow list is a reference for the advantages and disadvantages of some approaches:

Include a support statement in the SETTINGS frame. The SETTINGS frame is a HTTP/2 feature and is sent by the server to the client to exchange information about the current connection. The idea was to include an additional statement in this frame, so the client can detect support for resumable uploads without an additional roundtrip. The problem is that this is not compatible with HTTP/1.1. Furthermore, the SETTINGS frame is intended for information about the current connection (not bound to a request/response) and might not be persisted when transmitted through a proxy.

Include a support statement in the DNS record. The client can detect support when resolving a domain name. Of course, DNS is not semantically the correct layer. Also, DNS might not be involved if the record is cached or retrieved from a hosts files.

Send a HTTP request to ask for support. This is the easiest approach where the client sends an OPTIONS request and uses the response to determine if the server indicates support for resumable uploads. An alternative is that the client sends the request to a well-known URL to obtain this response, e.g. /.well-known/resumable-uploads. Of course, while being fully backwards-compatible, it requires an additional roundtrip.

Include a support statement in previous responses. In many cases, the file upload is not the first time that the client connects to the server. Often additional requests are sent beforehand for authentication, data retrieval etc. The responses for those requests can also include a header which indicates support for resumable uploads. There are two options: - Use the standardized Alt-Svc response header. However, it has been indicated to us that this header might be reworked in the future and could also be semantically different from our intended usage. - Use a new response header Resumable-Uploads: https://example.org/files/* to indicate under which endpoints support for resumable uploads is available.

Send a 104 intermediate response to indicate support. The clients normally starts a traditional upload and includes a header indicate that it supports resumable uploads (e.g. Upload-Offset: 0). If the server also supports resumable uploads, it will immediately respond with a 104 intermediate response to indicate its support, before further processing the request. This way the client is informed during the upload whether it can resume from possible connection errors or not. While an additional roundtrip is avoided, the problem with that solution is that many HTTP server libraries do not support sending custom 1XX responses and that some proxies may not be able to handle new 1XX status codes correctly.

Send a 103 Early Hint response to indicate support. This approach is the similar to the above one, with one exception: Instead of a new 104 (Upload Resumption Supported) status code, the existing 103 (Early Hint) status code is used in the intermediate response. The 103 code would then be accompanied by a header indicating support for resumable uploads (e.g. Resumable-Uploads: 1). It is unclear whether the Early Hints code is appropriate for that, as it is currently only used to indicate resources for prefetching them.

Upload Metadata

The Upload Creation Procedure (Section 4) allows the Content-Type and Content-Disposition header to be included. They are intended to be a standardized way of communicating the file name and file type, if available. However, this is not without controversy. Some argue that since these headers are already defined in other specifications, it is not necessary to include them here again. Furthermore, the Content-Disposition header field's format is not clearly enough defined. For example, it is left open which disposition value should be used in the header. There needs to be more discussion whether this approach is suited or not.

However, from experience with the tus project, users are often asking for a way to communicate the file name and file type. Therefore, we believe it is help to explicitly include an approach for doing so.

FAQ

*__Are multipart requests supported?__ Yes, requests whose body is encoded using the multipart/form-data are implicitly supported. The entire encoded body can be considered as a single file, which is then uploaded using the resumable protocol. The server, of course, must store the delimiter ("boundary") separating each part and must be able to parse the multipart format once the upload is completed.

Authors' Addresses

Marius Kleidl
Transloadit Ltd

Email: marius@transloadit.com

Jiten Mehta
Apple Inc.

Email: jmehta@apple.com

Guoye Zhang
Apple Inc.

Email: guoye_zhang@apple.com

Lucas Pardue
Cloudflare

Email: lucaspardue.24.7@gmail.com

Stefan Matsson
JellyHive

Email: s.matsson@gmail.com