tus resumable upload protocol

Version: 0.2.1 (SemVer)
Date: 2013-04-15
Authors: Felix Geisendörfer, Kevin van Zonneveld, Tim Koschützki, Naren Venkataraman

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

1. Status

This protocol is under development and is not ready for general adoption yet.

The goal is to have a 1.0 release candidate in a few weeks, at which point only minor revisions will be made to the protocol. After a few months 1.0 will be frozen, and changes will be considered very carefully. Ideally there will be no need for a version 2.0.

2. Contributing

This protocol is authored and owned by the tus community. We welcome patches and feedback via Github. All contributors will be listed as authors.

Please also let us know about any implementations (open source or commercial) if you'd like to be listed on the implementations page.

3. Abstract

The protocol provides a mechanism for resumable file uploads via HTTP 1.1 (RFC 2616).

4. Notation

Characters enclosed by square brackets indicate a placeholder (e.g. `[size]`).

5. Core Protocol

The core protocol describes how to resume an interrupted upload. It assumes that you already have a URL for the upload, usually created via the File Creation extension.

All clients and servers MUST implement the core protocol.

5.1. Example

A HEAD request is used to determine the offset at which the upload should be continued.

The example below shows the continuation of a 100 byte upload that was interrupted after 70 bytes were transfered.

Request:

```
HEAD /files/24e533e02ec3bc40c387f1a0e460e216 HTTP/1.1

Host: tus.example.org
```

Response:

```
HTTP/1.1 200 Ok

Offset: 70
```

Given the offset, the client uses the PATCH method to resume the upload:

Request:

```
PATCH /files/24e533e02ec3bc40c387f1a0e460e216 HTTP/1.1

Host: tus.example.org

Content-Length: 30

Offset: 70



[remaining 30 bytes]
```

Response:

```
HTTP/1.1 200 Ok
```

5.2. Headers

5.2.1. Offset

The `Offset` header is a request and response header that indicates a byte offset within a resource. The value MUST be an integer that is `0` or larger.

5.3. Requests

5.3.1. HEAD

Servers MUST always return an `Offset` header for `HEAD` requests against a tus resource, even if it is `0`, or the upload is already considered completed.

5.3.2. PATCH

Servers MUST accept `PATCH` requests against any tus resource and apply the bytes contained in the message at the given `Offset`. All `PATCH` requests MUST use`Content-Type: application/offset+octet-stream`.

The `Offset` value SHOULD be equal, but MAY also be smaller than the current offset of the resource, and servers MUST handle `PATCH` operations containing the same data at the same absolute offsets idempotently. The behavior of using `Offset` values larger than the current upload offset is undefined, see the Parallel Chunks extension.

Clients SHOULD send all remaining bytes of a resource in a single `PATCH` request, but MAY also use multiple small requests for scenarios where this is desirable (e.g. NGINX buffering requests before they reach their backend).

Servers MUST acknowledge successful `PATCH` operations using a `200 Ok` status, which implicitly means that clients can assume that the new `Offset` = `Offset` +`Content-Length`.

Both clients and servers SHOULD attempt to detect and handle network errors predictably. They may do so by checking for read/write socket errors, as well as setting read/write timeouts. Both clients and servers SHOULD use a 30 second timeout. A timeout SHOULD be handled by closing the underlaying connection.

Servers SHOULD always attempt to process partial message bodies in order to store as much of the received data as possible.

Clients SHOULD use a randomized exponential back off strategy after encountering a network error or receiving a `500 Internal Server Error`. It is up to the client to decide to give up at some point.

6. Protocol Extensions

Clients and servers are encouraged to implement as many of the extensions described below as possible. There is no feature detection for clients, instead they should allow individual extensions to be enabled/disabled in order to match the available server features.

6.1. File Creation

All clients and servers SHOULD implement the file creation API. In cases where that does not make sense, a custom mechanism may be used instead.

6.1.1. Example

An empty POST request is used to create a new upload resource. The `Final-Length`header indicates the final size of the file.

Request:

```
POST /files HTTP/1.1

Host: tus.example.org

Content-Length: 0

Final-Length: 100
```

Response:

```
HTTP/1.1 201 Created

Location: http://tus.example.org/files/24e533e02ec3bc40c387f1a0e460e216
```

The new resource has an implicit offset of `0` allowing the client to use the core protocol for performing the actual upload.

6.1.2. Headers

6.1.2.1. Final-Length

The `Final-Length` header indicates the final size of a new resource in bytes. This way a server will implicitly know when a file has completed uploading. The value MUST be a non-negative integer.

6.1.3. Requests

6.1.3.1. POST

Clients MUST use a `POST` against a well known file creation url to request the creation of a new file resource. The request MUST include a `Final-Length` header.

Servers MUST acknowledge a successful file creation request with a `201 Created`response code and include an absolute url for the created resource in the `Location`header.

Clients then continue to perform the actual upload of the file using the core protocol.

6.2. Checksums

This extension will define how to provide per file or per chunk checksums for uploaded files.

6.3. Parallel Chunks

This extension will define how to upload several chunks of a file in parallel in order to overcome the throughput limitations of individual tcp connections.

6.4. Metadata

This extension will define how to provide meta information when uploading files.

6.5. Streams

This extension will define how to upload finite streams of data that have an unknown length at the beginning of the upload.

7. FAQ

7.1. Why is the protocol using custom headers?

We have carefully investigated the use of existing headers such as `Range` and`Content-Range`, but unfortunately they are defined in a way that makes them unsuitable for resumable file uploads.

We also considered using existing `PATCH` payload formats such asmultipart/byteranges, but unfortunately the XHR2 FormData interface does not support custom headers for multipart parts, and the send() method does not allow streaming arbitrary data without loading all of it into memory.

That being said, custom headers also allowed us to greatly simplify the implementation requirements for clients and servers, so we're quite happy with them.

7.2. Why are you not using the "X-" prefix for your headers?

The "X-" prefix for headers has been deprecated, see RFC 6648.

7.3. How can I deal with bad http proxies?

If you are dealing with http proxies that strip/modify http headers or can't handle`PATCH` requests properly, you should consider using https which will make it impossible for proxies to modify your requests.

If that is not an option for you, please reach out to us, we are open to defining a compatibility protocol extension.

7.4. How are pause/resume handled? When should I delete partial uploads?

Needs to be written ...

8. License