Internet-Draft
Expires: Aug 20, 2026
Intended status: Proposed Standard

M. Toomim
Invisible College

Mar 02, 2026

HTTP Resource Versioning
draft-toomim-httpbis-versions-04

Abstract

HTTP resources change over time. Each change to a resource creates a new "version" of its state. HTTP systems often need a way to identify, read, write, navigate, and/or merge these versions, in order to implement cache consistency, create history archives, settle race conditions, request incremental updates to resources, interpret incremental updates to versions, or implement distributed collaborative editing algorithms.

This document analyzes existing methods of versioning in HTTP, highlights limitations, and specifies a more general versioning approach that can enable new use-cases for HTTP. An upgrade path for legacy intermediaries is provided.

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), its areas, and its working groups. 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."

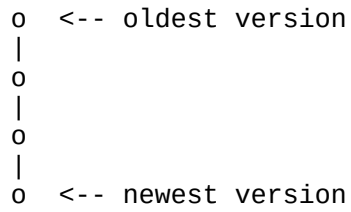
The list of current Internet-Drafts can be accessed at https://www.ietf.org/1id-abstracts.html

The list of Internet-Draft Shadow Directories can be accessed at https://www.ietf.org/shadow.html

Table of Contents

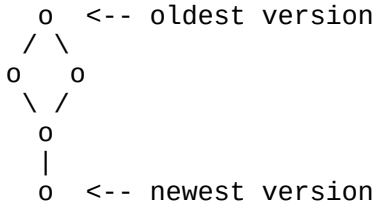
1. Introduction

HTTP resources change over time. Any single computer on the network can observe a resource changing, over time, in a linear sequence of versions:



We call this a "linear" history.

However, when multiple networked computers change a resource "simultaneously" -- before their edits propagate to one another -- the global perspective of version history actually forks into a DAG, or "partial order":



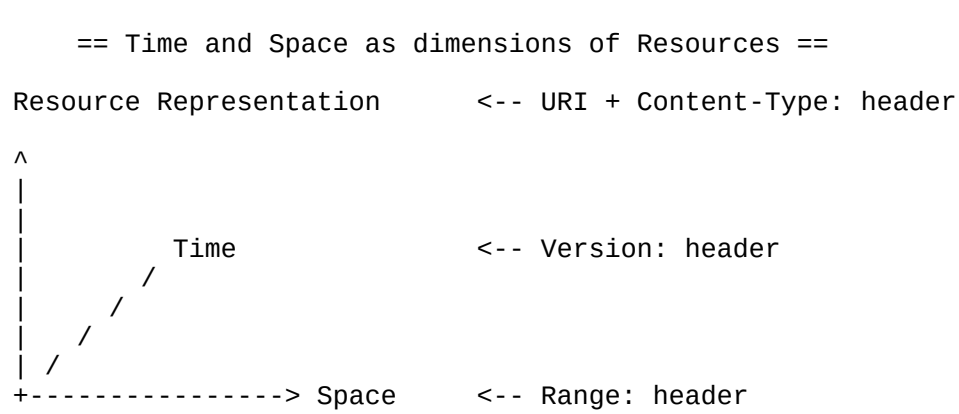
HTTP systems often need to specify, identify, and navigate these histories of versions in order to (1) implement cache consistency, (2) create history archives, (3) settle race conditions, (4) request incremental updates to resources, (5) interpret incremental updates to versions, or (6) merge parallel mutations in distributed collaborative editing algorithms.

Each of these systems needs a way to represent *time*, but different systems have different requirements for how time is represented. This document enumerates these needs, and proposes a general versioning system that satisfies them all.

(Note that this document does NOT speak to the versioning of HTTP APIs -- only HTTP resources, which are used within APIs.)

1.1. Model of time

This specification adds a dimension of time into the existing model of HTTP resources. Whereas HTTP Resources already have a dimension of *space*, which can be queried via the Range header, this specification augments Resources with an analogous dimension of *time*, which can be queried via the Version header:



Any Resource can thus be queried for its Time and Space, via a Version and Range:

```
GET /foo
Version: "v1.2.3"
Range: bytes 500-1000
```

In this model, the term "Version" is synonymous with "Timestamp" -- a Version represents a point in distributed time. A single Version can be specified across multiple resources, to identify their states at the same Version in time. A Version can also be identified without reference to any single resource, and can be thought of as a "Version of the universe" in which Resources exist.

It follows that a "Version of a Resource" specifies the state of the Resource at the time of that Version. The "Version of a Representation" further implies a particular state of the Representation body, or "snapshot."

Note that this model differs in terminology from some other mechanisms, such as WebDAV Versioning, which uses the term Version for what we call a "Version of a Representation."

1.2. Existing Versioning approaches in HTTP

Existing approaches to versioning in HTTP address disparate use-cases, but have limitations and trade-offs. The Last-Modified and ETag headers were invented for cache consistency, but do not provide an ordering of version history through time, nor do they handle forks and merges in distributed time. On the other hand, a number of forking/merging versioning systems have been proposed (WebDAV, Link Relations) that create new resources to represent versions of existing resources, but these rely on central servers to assign version identifiers, and require additional network round-trips to requery these version resources. No HTTP versioning system today allows for articulating custom distributed timestamp formats such as version vectors.

We now enumerate the existing approaches. The next section (1.3) provides an overview of their limitations for a general versioning system.

1.2.1. Versioning with ‘Last-Modified‘

The Last-Modified header specifies a clock date that caches and clients can use to know when a change has occurred:

```
Last-Modified: Sat, 6 Jul 2024 07:28:00 GMT
```

This header is useful for caching and conditional requests (using the If-Modified-Since header). However, it has several limitations:

1. It is limited to the precision of the wallclock. If a resource changes within the same second, the Last-Modified date won’t change, and caches can become inconsistent.

2. It stores linear time; not distributed (partial order) time. It cannot represent the ambiguity of forked time.

3. It is susceptible to clock skew in distributed systems, potentially leading to inconsistencies across different servers.

1.2.2. Versioning with ‘ETag‘

The ETag header allows more precision. It specifies a version with a string that uniquely identifies a cacheable representation:

```
ETag: "2u34fa7yorz0"
```

ETags can be strong or weak, with weak ETags prefixed by W/:

```
ETag: W/"2u34fa7yorz0"
```

ETags are used in conditional requests with If-None-Match and If-Match headers and can be used for optimistic concurrency control. However:

1. While helping with cache validation, ETags are not accurate markers of time. There is no way to order versions by ETag, or know which version came before another.

2. ETags are unique to content, not timestamps. It’s possible for the same ETag to recur over time if the resource changes back and forth between a common state.

3. Strong ETags are sensitive to Content-Encoding. If a single version of a resource is transmitted with different Content-Encodings (e.g., gzip), it will be sent with different strong ETags. Thus, one can have multiple ETags for the same version in history, as well as a single ETag for multiple versions of history.

1.2.3. Versioning encoded within URIs

In practice, application programmers tend to encode versions within URIs:

> https://unpkg.com/braid-text@0.0.18/index.js

This approach is common in API versioning (e.g., /api/v1/resource). However, it has several drawbacks:

1. It loses the semantics of a "resource changing over time." Instead, it creates multiple version resources for every single logical resource.

2. It necessitates additional standards for version history on top of URIs (e.g., Memento, WebDAV, Link Relations for Versioning [RFC5829]).

3. Given a URI, we still need a standard way to extract the version itself, get the previous and next version(s), and understand the format of the version(s) (e.g., major.minor.patch).

4. This approach can lead to URI proliferation, potentially impacting caching strategies and SEO.

5. It may complicate content negotiation and RESTful design principles.

The choice to embed versions into URIs can be useful, but carries with it additional tradeoffs. A versioning system does not need to depend on allocating a URI for each version; but could be compatible with doing so.

1.3. Requirements for general versioning

A general mechanism for versioning HTTP resources could enable a number of new use-cases:

- RSS clients could request incremental updates when polling, instead of re-downloading redundant unchanged feed items after each change to any item

- Servers could accept incoming patches based on old or parallel versions of history, and even rebase those patches for other clients, at other points in history

- Collaborative editing could be built directly into HTTP resources, providing the abilities of Google Docs at any URL

- Git repositories could be hosted directly over HTTP; rather than embedding versioning information within opaque blobs that use HTTP just as a transport

- Caches and archives could hold and serve multiple versions of a resource, enabling audits and distributed backups

- Distributed databases could standardize network APIs to HTTP, while retaining distributed consistency guarantees

To support these use-cases with a single, general Versioning model it must support a number of abilities. This section defines those abilities, and enumerates which existing versioning approaches support, and do not support, them.

1.3.1. Distributed Time

Distributed systems require support for distributed time, including:

(a) A comparable partial order:

Any two Versions (timestamps) A and B must be comparable, such that either:

- A > B: A came after B
- A < B: A came before B
- A <> B: Neither A nor B came before the other

(b) Immune to clock skew or limited precision:

The ordering must not depend on the skew or precision of a CPU’s wallclock.

(c) Merges must be specifiable without central authority:

Any peer must be able to identify the version resulting from the merger of two parallel versions A and B, without relying on a central authority to assign a new version identifier AB.

1.3.2. Integrated into HTTP requests and responses

For some use-cases and intermediaries, it is not reasonable or possible to require additional requests and responses (with additional network round-trips) to be made, and/or new methods or resources to be defined.

These use-cases require versioning information to instead be embedded within existing HTTP requests and responses, at all places that they are helpful. We enumerate these requirements as follows:

(a) Includes versioning in every existing request and response where state is queried or mutated.

(b) Does not require additional round-trips.

(c) Does not require polluting the application’s URL namespace.

1.3.3. Generalizable Timestamps

Some systems have requirements for Version timestamps themselves:

(a) Extensible timestamp formats:

Advanced distributed systems often devise special formats for partially-ordered timestamps that allow inferences for improved performance, such as lamport clocks, vector clocks, version vectors, hash histories, hybrid logical clocks, and append-only-log indices. Implementations can rely on information embedded in these timestamps to compress history metadata, optimize partial-order computations, or infer the value of state.

(b) Independent of resource:

Some applications need to assign a single version to multiple resources, so that one can refer to multiple resources at the same point in time. For instance, a git repository commits multiple files at the same version.

A general versioning system must provide version identifiers that are independent of any particular resource, so that multiple resources can be versioned and referred to at the same points in time.

1.3.4. Feature table of existing approaches

We compose a table of existing approaches, and the features they do not yet provide, here:

	1a	1b	1c	2a	2b	2c	3a	3b
Last-Modified	-	-	-	X	X	X	-	X
ETags	-	-	-	X	X	X	-	-
Memento	-	~	-	-	-	~	-	-
WebDAV Versioning	X	X	-	-	-	-	-	-
Link Relations	X	X	-	X	-	~	-	-

Because no existing versioning approach satisfies all needs, programmers today must implement multiple approaches to versioning in their applications -- each with subtly different logic -- and cannot implement common infrastructure for distributed versioning, archiving, and collaborative editing that works across HTTP systems.

1.4. Overview of Proposed Solution

This document specifies a general versioning model satisfying all 8 requirements above. It features:

1. Version and Parents Headers: New headers to specify the current version of a resource and its parent versions, enabling representation of both linear and non-linear version histories.

2. Version as Sets of Strings: Versions are represented as sets of unique string identifiers, allowing for custom versioning schemes and distributed timestamps.

3. Extensible Version-Type Header: Allows specification of different timestamp formats in custom versioning schemes (e.g., git-style hashes, bytestreams and append-only logs, vector clocks) to allow additional computational inferences for various use cases.

4. Versioned Requests and Responses: Extends standard HTTP methods (GET, PUT, PATCH) with versioning semantics, allowing version-aware interactions with resources.

This system provides a flexible foundation that can be adapted to various versioning needs, from simple content distribution to complex collaborative editing scenarios, while maintaining compatibility with existing HTTP infrastructure.

We start by specifying how to add versioning to HTTP requests and responses.

2. HTTP Resource Versioning

This section defines the core concepts and mechanisms for HTTP Resource Versioning.

2.1. Version and Parents Headers

This specification introduces two new HTTP headers: Version and Parents. These headers communicate version information in requests and responses.

The Version header specifies the current version of a resource:

```
Version: "dkn7ov2vwg"
```

The Parents header specifies the immediate predecessor version(s):

```
Parents: "ajtva12kid", "cmdpvkpll2"
```

These headers may be used in PUT, PATCH, POST, and DELETE requests, and GET and HEAD responses, to convey the version before and after the Update conveyed by the HTTP message. (See [Updates].)

These headers can also be used in GET and HEAD requests to ask a server for a specific version or range of version history.

If an Update (a PUT, PATCH, POST, or DELETE request, or a GET or HEAD response) does not specify a Version header, the recipient MAY generate and assign it a new version ID. If an Update does not specify a Parents header, the recipient MAY presume that the most recent versions it has (the frontier of time) are the parents of the new version.

To describe a merger, a Version header MAY contain multiple IDs:

```
Version: "dkn7ov2vwg", "v2vwgdkn7o"
```

2.1.1. Formatting of Version IDs in Versioning Headers

The Version and Parents headers are formatted as a list of strings in the Structured Headers format [RFC8941]. Each string is called an Event ID.

```
Version: <version>
Parents: <version>

<version>: <event-id>, <event-id>, ...
<event-id>: <string>
```

The ordering of IDs within a Version or Parents header carries no meaning. Event IDs SHOULD be sorted lexicographically whenever received or sent, with exactly one space after "," separators, to canonicalize the set's serialization as a unique string, e.g. as a unique cache key.

The formatting of Event IDs is constrained according to the resource's Version-Type, as defined in Section 4.

2.1.2. Definition of Events and Versions

An "Event" is a unique ID assigned to a single event in history at a peer. Any peer can define an event, by generating a unique ID. Events are generally defined at state mutations, but events MAY also be defined at a "no-op mutation", when a unique identifier to mark time is desired.

Conceptually, for any event E, all prior events that have been observed by that peer before defining E can be referenced as the set ancestors(E). The frontier of this set, or frontier(ancestors(E)), is defined as the events that are not in the ancestors(e') for any other event e' in ancestors(E). We define the parents(E) to be this frontier(ancestors(E)).

A Version in time is likewise defined by the set of observed events at that time, and we define any Version V as the frontier(observed_events) at that time.

Thus, the version immediately after any event E is simply the set {E}, and the version after the merger of two parallel events E1 and E2 is the set {E1, E2}.

As a result, the version of a resource immediately after a mutation will contain just a single string:

```
Version: "foo-123"
```

The version of a merger will generally contain multiple strings:

```
Version: "foo-123", "bar-abc"
```

However, it is also possible for a peer to define a no-op event that represents the merger as a single string:

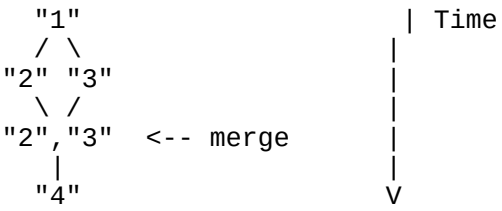
```
Parents: "foo-123", "bar-abc"
Version: "{foo-123, bar-abc}"
```

2.2. Semantics of Versions and Version History

A resource's Version History is the partial order (or DAG) of Versions to its state. Each version marks a unique point in

distributed time: a point when the resource was in a particular state, and its representations had particular values.

For example, this Version History has one fork and one merge:



The semantics of a merge is specified by the resource's Merge-Type. (See [Merge-Types].)

Although the Version History exists conceptually, it might not be known in its entirety by any one peer. Each peer can store and work with different subsets of the version history independently. Also, each peer can merge mutations in different orders, creating a different sequence of intermediate merged versions, and some intermediate merge versions may never have existed on any peer, even though they are possible to reconstruct from known history.

2.3. Using Versioning with HTTP Methods

The Version and Parents headers can be set in requests and responses of methods:

- GET
- HEAD
- PUT
- PATCH
- POST

For each of these methods, if a Version or Parents header exists in the request, it should also exist in the response, and if not, the server may nonetheless specify it in its response.

We now detail the ways in which these headers modify requests and responses for these methods.

2.3.1. GET the current version

If the Version: header is not specified, a GET request returns the current version of the state as usual:

Request:

```
GET /chat
```

Response:

```
HTTP/1.1 200 OK
Version: "ej4lhb9z78"
Parents: "oakwn5b8qh", "uc9zwhw7mf"
Content-Type: application/json
Content-Length: 64

[{"text": "Hi, everyone!",
  "author": {"link": "/user/tommy"}}]
```

The server MAY include a Version and/or Parents header in the response, to indicate the current version and its parents.

Clients can use a HEAD request to elicit versioning history without downloading the body:

Request:

```
HEAD /chat
```

Response:

```
HTTP/1.1 200 OK
Version: "ej4lhb9z78"
Parents: "oakwn5b8qh", "uc9zwhw7mf"
Content-Type: application/json
```

2.3.2. GET a specific version

A server can allow clients to request historical versions of a resource in GET requests by responding to the Version and Parents headers. A client can specify a specific version that it wants with the Version header:

Request:

```
GET /chat
Version: "ej4lhb9z78"
```

Response:

```
HTTP/1.1 200 OK
Version: "ej4lhb9z78"
Parents: "oakwn5b8qh", "uc9zwhw7mf"
Content-Type: application/json
Content-Length: 64

[{"text": "Hi, everyone!",
  "author": {"link": "/user/tommy"}}]
```

2.3.3. PUT, POST, or PATCH a new version

When a PUT, POST, or PATCH request changes the state of a resource, it can specify the new version of the resource, and the parent version that it was based on:

Request:

```
PUT /chat
Version: "ej4lhb9z78"
Parents: "oakwn5b8qh", "uc9zwhw7mf"
Content-Type: application/json
Content-Length: 64

[{"text": "Hi, everyone!",
  "author": {"link": "/user/tommy"}}]
```

Response:

```
HTTP/1.1 200 OK
```

The Version and Parents headers are optional. If Version is omitted, the recipient may assign new event IDs. If Parents is omitted, the recipient may assume that its current version is the version's parents.

2.3.4. GET a range of historical versions

A client can request a range of history by including a Parents and a Version header together. The Parents marks the beginning of the range (the oldest versions) and the Version marks the end of the range (the newest versions) that it requests.

Request:

```
GET /chat
Version: "3"
Parents: "1a", "1b"
```

Response:

```
HTTP/1.1 209 Multiresponse
Current-Version: "3"

HTTP/1.1 200 OK
Version: "2"
Parents: "1a", "1b"
Content-Type: application/json
Content-Length: 64

[{"text": "Hi, everyone!",
  "author": {"link": "/user/tommy"}}]

HTTP/1.1 200 OK
Version: "3"
Parents: "2"
Content-Type: application/json
Merge-Type: sync9
Content-Length: 117

[{"text": "Hi, everyone!",
  "author": {"link": "/user/tommy"}}
 {"text": "Yo!",
  "author": {"link": "/user/yobot"}]
```

Note that this example uses a new "Multiresponse" code, which is currently being drafted [Multiresponse]. See [Braid-HTTP] Section 3 for an earlier draft of the semantics.

2.4. Rules for Version and Parents headers

If a GET request contains a Version header:

- If the Parents header is absent, the server SHOULD return a single response, containing the requested version of the resource in its body, with the Version response header set to the same version.

- If the server does not support historical versions, it MAY ignore the Version header and respond as usual, but MUST NOT include the Version header in its response.

If a GET request contains a Parents header:

- The server SHOULD send the set of versions updating the Parents to the specified Version. If no Version is specified, then it should update the client to the server’s current version.

- If the server does not support historical versions, then it MAY ignore the Parents header, but MUST NOT include the Parents header in its response.

A server does not need to honor historical version requests for all documents, for all history. If a server no longer has the historical context needed to honor a request, it responds with error code 432 Version Not Found.

2.5. Status 432: Version Not Found

If a server does not have a version required of a request, it should respond with status 432 Version Not Found, and include copies of the request header(s) with versions it could not satisfy. For example, if a client does:

```
GET /foo
Version: "alice-44"
Parents: "bob-32"
```

...but the server does not have the versions 'alice-44', it should respond with:

```
432 Version Not Found
Version: "alice-44"
```

Peers can drop history at any time. Clients cannot rely on any particular portion of history existing on a server or intermediary when it makes requests.

2.6. The Current-Version header

While sending historical versions, a server or client can specify its current latest version with the Current-Version header. The other party may desire this information to know when it has caught up with the latest version. This is also used in the resumable uploads example below.

3. Version-Type Header

The optional Version-Type header specifies constraints on the format and interpretation of event IDs. This allows for various optimizations and specialized versioning schemes.

For example:

```
Version-Type: git
```

This could indicate that event IDs will be git-style hashes, branches, or tags. Peers could verify that the entire repository at a given version hashes to the specified ID.

Another example:

```
Version-Type: peer-counter; text-runs
```

Diamond-Types, Automerge, and other algorithms use this format to compress history metadata through run-length encoding of consecutive insertions. This allows a set of 50 inserted characters to be stored as 50 bytes plus one event ID, rather than 50 bytes plus 50 event IDs (each of which takes up multiple bytes).

Implementers may define custom Version-Types to suit specific needs:

```
Version-Type: version-vector
```

A Version Vector Version ID might take the form:

```
Version: "{peerid1: counter1, peerid2: counter2, ...}"
```

Version Vectors enable direct computation of partial order between any two event IDs without examining the full version history graph.

(To know the order between two Version Vectors A and B, one needs only to compare each peer’s counter between A and B. If A dominates across all peers, it is newer. If B dominates, then it is newer. Otherwise, the ordering between the two version vectors is not known, and we can say that they happened in parallel.)

3.1. The peer-counter Version-Type

The "peer-counter" Version-Type specifies that each Event ID is a Lamport timestamp, of the form:

```
Version: "<peer>-<counter>"
```

Or:

```
Version: "<peer>-<counter>"; text-runs
Version: "<peer>-<counter>"; bytestream
```

It is guaranteed that, for any peer, the counter will always increase in subsequent versions. Therefore, any two Event IDs from the same peer can be compared in time just by examining their counters.

3.1.1. The "text-runs" modifier

The modifier "text-runs" further constrains the behavior of the counter to enable run-length encoding. Specifically, for each inserted set of characters, the counter is constrained to be set to the number of unicode codepoints that have been inserted; and for every deleted set of characters, the counter is constrained to represent the number of unicode codepoints that have been deleted.

Furthermore, it defines that intermediate versions can be inferred, between any explicitly transmitted version, by assuming that for any insertion, all characters are inserted one-by-one, left-to-right, by unicode codepoint; and for any deletion, all characters are deleted right-to-left, by unicode codepoint.

3.1.2. The "bytestream" modifier

The "bytestream" modifier specifies that the resource can be considered as an append-only bytestream:

```
Version-Type: peer-counter; bytestream
```

Each <counter> then specifies that the number of bytes in the resource is equal to <counter> at time <peer>-<counter>.

For example, "x82ha-344" indicates "the resource state after peer ‘x82ha‘ appended 344 bytes".

This approach creates a direct correspondence between time and space: each version increment represents one additional byte in the stream.

4. Versioning through Intermediaries

Intermediaries can take advantage of versioning to uniquely reference, store, and serve multiple states/updates across a resource’s history. To distinguish versions, intermediaries must add the "Version" and "Parents" headers to their cache keys -- equivalent to the Vary header:

```
Vary: version, parents
```

Intermediaries SHOULD behave as if the Version and Parents headers have been added to the Vary header in every response passing through them. To support legacy versioning-unaware intermediaries, the origin server is RECOMMENDED to explicitly add or extend Vary with "version, parents" in all its responses, unless it is certain that no legacy intermediaries will process the response.

4.1. Detecting Legacy Intermediaries

In the case that a legacy intermediary *does* process a versioned response without the Vary header, it can be detected by the client noticing that the Version and Parents Event IDs in a client request are not present in the request's response. Here is an example:

Presume we start with two versions:

```
PUT /foo
Version: "1"

Hello

PUT /foo
Version: "2"

Hello World!
```

Now, if someone GETs the old version:

```
GET /foo
Version: "1"
```

Then a versioning-aware origin server can return it:

```
HTTP/1.1 200 OK
Version: "1"

Hello
```

However, a versioning-unaware intermediary will cache this old response as if it is the current state.

This breaks when a client requests the *newest* version:

```
GET /foo
Version: "2"
```

And the cache ignores the Version header and returns what it has most recently seen:

```
HTTP/1.1 200 OK
Version: "1"

Hello
```

To detect this, the client SHOULD check that the Version and Parents headers in the response contain a superset of the Event IDs specified in the request. (A missing Version or Parents header is considered to be an empty set for the version or parents of this superset calculation.) The client can throw a warning to the programmer when encountering a response that does not contain the request's Event IDs, so that he knows to add Vary to the server's responses.

5. Example Uses

5.1. Incremental RSS Subscription

Traditional RSS readers inefficiently poll servers, often downloading entire feeds when only minor changes have occurred. HTTP Resource Versioning enables more efficient incremental updates.

A client can specify its last known version using the Parents header:

Request:

```
GET /feed.rss
Accept: application/rss+xml
Parents: "4"
```

The server can then respond with only the changes since that version:

Response:

```
HTTP/1.1 200 OK
Content-Type: application/rss+xml+patch
Version: "5"
Parents: "4"

<?xml version="1.0" encoding="UTF-8" ?>
<rss version="2.0">
<channel>
 <title>My RSS Feed</title>
 <item>
  <title>This is a new entry</title>
  <description>Incremental update example</description>
  <link>http://www.example.com/blog/post/1</link>
 </item>
</channel>
</rss>
```

This approach significantly reduces bandwidth usage and processing time for both client and server. The specific patch format used can vary; see [Updates] or [Range-Patch] for examples.

5.2. Hosting git via HTTP

We can host a git repository directly through HTTP, where each file corresponds to a resource, and all have a version history.

Git versions are normally specified as a hash. The server can express this with a "Version-Type: git" header:

Request:

```
GET /repo/readme.md
```

Response:

```
HTTP/1.1 200 OK
Content-Type: text/markdown
Version-Type: git
Version: "9531a9702af0d90dd489050ed8e25f87912a9252"
Parents: "3a4c361f8e0349fe4b25c1ff46ebec1cec66e60f"

...
```

Git also allows specifying a version with a short string, like "HEAD", which works for any tag or branch. We can request the latest "development" branch version with:

Request:

```
GET /repo/readme.md
Version: "development"
```

Response:

```
HTTP/1.1 200 OK
Content-Type: text/markdown
Version-Type: git
Version: "9e26e8837a4f6a4445e74eed744fe8af85efd0c2"
Parents: "1d5f89f8843b33b91d62bf95877e46b23fd86741"

...
```

One can also request the files from release tagged "1.3.5" using:

Request:

```
GET /repo/readme.md
Version: "1.3.5"
```

One can clone a repo by asking for all versions from the root to HEAD:

Request:

```
GET /repo/readme.md
Version: "HEAD"
Parents: "ROOT"
```

Response:

```
HTTP/1.1 209 Multiresponse

HTTP/1.1 200 OK
Content-Type: text/markdown
Version-Type: git
Version: "9e26e8837a4f6a4445e74eed744fe8af85efd0c2"
Parents: "1d5f89f8843b33b91d62bf95877e46b23fd86741"
Content-Length: 190

...

HTTP/1.1 200 OK
Content-Type: text/markdown
Version-Type: git
Version: "1d5f89f8843b33b91d62bf95877e46b23fd86741"
Parents: "1cf6ab4ed836d4d7308ac93edbc6fd18a69ef88f"
Content-Length: 192

...
```

In fact, git itself already supports two HTTP protocols: a "dumb" and a "smart" protocol. The dumb protocol uses plain HTTP, but doesn't support incremental updates -- each pull re-downloads the entire pack file. The smart protocol allows the client to specify the version it has, and the version it wants:

```
0054want 31f1c37dfa1bf983e4d67e06fac28e8e6f
00093bd7884 HEAD@{1}
0032have e68fe437718c37155c7e3e5f4a3ff17c4f476940
0000
```

We can express this with HTTP Versioning as:

Request:

```
GET /repo/readme.md
Version: "31f1c37dfa1bf983e4d67e06fac28e8e6f"
Parents: "e68fe437718c37155c7e3e5f4a3ff17c4f476940"
```

This expresses aspects of the "smart" git protocol over plain HTTP.

5.3. Resumable uploads protocol

Resource Versioning semantics enable efficient implementation of resumable uploads, providing an alternative perspective to [Resumable Upload].

To initiate an upload, the client specifies the Version-Type and the expected final version using the Current-Version header:

Request:

```
PUT /something
Current-Version: "abwejf-900"
Version-Type: peer-counter; bytestream
Content-Length: 900

<binary data of length 900>
```

For a successful upload, the server responds as usual:

Response:

```
200 OK
```

If the upload is interrupted, the client can query the server's current state:

Request:

```
HEAD /something
Parents: "abwejf-0"
```

The server's response determines the client's next action:

A. Upload complete:

Response:

```
200 OK
```

```
      Parents: "abwejf-0"
      Version: "abwejf-900"
```

B. Partial upload:

```
   Response:

      206 Partial Content
      Parents: "abwejf-0"
      Version: "abwejf-400"
```

C. No upload progress:

```
   Response:

      416 Range Not Satisfiable
```

Based on the response, the client proceeds as follows:

- Case A: Upload is complete, no further action needed.
- Case B: Resume the upload from the last received byte:

   Request:

      PUT /something
      Current-Version: "abwejf-900"
      Parents: "abwejf-400"
      Content-Range: bytes 400-900/900
      Content-Length: 500

      <binary data from 400-900>

- Case C: Restart the upload from the beginning.

This protocol leverages general version semantics, allowing servers implementing HTTP Resource Versioning with the "bytestream" Version-Type to inherently support resumable uploads.

5.4. Distributed collaborative editing

This versioning system can also support full CRDT and OT collaborative editing features (when used with other extensions such as [Braid-HTTP]), allowing every URL to gain the functionality of Google Docs.

The [Braid-Text] project implements a very efficient style of this. When you first load a resource, a server provides it as a single version:

Request:

```
GET https://braid.org/test
Accept: text/plain
Subscribe: true
```

Response:

```
HTTP/1.1 209 Multiresponse

HTTP/1.1 200 OK
Version: "2agvvzgccrq-5"
Version-Type: peer-counter; text-runs
Merge-Type: simpleton
Content-Length: 12

Hello world!
```

Updates are expressed as a stream of patches:

```
   Response (continued):

      HTTP/1.1 200 OK
      Version: "4590r8uwm63-18"
      Parents: "2agvvzgccrq-5"
      Content-Length: 1
      Content-Range: text [12:12]

      :

      HTTP/1.1 200 OK
      Version: "4590r8uwm63-19"
      Parents: "4590r8uwm63-18"
      Content-Length: 1
      Content-Range: text [13:13]

      )
```

This versioning system supports multiple [Merge-Types], and they can even co-exist simultaneously for the same resource. For instance, braid-text supports two merge-types simultaneously:

- The "simpleton" merge-type requires the server to rebase all edits for the client

- The "dt" merge-type uses a fully peer-to-peer merge algorithm called Diamond-Types

Clients can connect with either merge-type, and can even change merge-type on-the-fly -- the version history itself can be re-used.

5.5. Improved Header Compression with runs

A header compression scheme can leverage Version-Type patterns to significantly compress messages. For example, consider the following scenario where 4 characters are inserted into a collaborative editor, requiring 479 bytes when uncompressed:

```
PUT /some.txt
Version: "3f84786c-57"
Parents: "3f84786c-56"
Content-Length: 1
Content-Range: text 471:471

a

PUT /some.txt
Version: "3f84786c-58"
Parents: "3f84786c-57"
Content-Length: 1
Content-Range: text 472:472

s

PUT /some.txt
Version: "3f84786c-59"
Parents: "3f84786c-58"
Content-Length: 1
Content-Range: text 473:473

d

PUT /some.txt
Version: "3f84786c-60"
Parents: "3f84786c-59"
Content-Length: 1
Content-Range: text 474:474

f
```

Huffman Encoding (e.g. in HTTP’s HPACK and QPACK) can compress these headers by identifying repeated strings and assigning them codes:

```
   U = "PUT /some.txt"
   W = "Version: \"3f84786c-{ }\""
   X = "Parents: \"3f84786c-{ }\""
   Y = "Content-Length: 1"
   Z = "Content-Range: text { }"
```

This compression reduces the messages to:

```
   U
   W{57}
   X{56}
   Y
   Z{471:471}

   a

   U
   W{58}
   X{57}
   Y
   Z{472:472}

   s

   U
   W{59}
   X{58}
   Y
   Z{473:473}

   d

   U
   W{60}
   X{59}
   Y
   Z{474:474}

   f
```

This initial compression reduces the 479 bytes to approximately 123 bytes.

Further compression is possible by recognizing that the numeric parameters in each header increment by 1 from the previous header. This allows us to take advantage of run-length encoding; specified as "Version-Type: peer-counter; text-runs". We can thus compress the entire run of inserts using a new compression function, RUN(a, b, c):

```
RUN(a, b, insertions) =
  for i, char in insertions
    return ‘
      U
      W{a+i}
      X{a+i-1}
      Y
      Z{b+i:b+i}

      char
    ’
```

Where the parameters are:

```
a: Initial version number
b: Initial content range
insertions: String of characters to be inserted
```

We could then compress the 479 bytes down to a single 20-byte message:

```
RUN(57, 474, ’asdf’)
```

For even more compact representation, sacrificing human-readability, this can be encoded in 12 bytes:

```
R57,474,asdf
```

Further compression is possible by using variable-length integer encoding (such as LEB128 or Protocol Buffers’ varint) for the numeric values. This approach could potentially reduce the encoding to 7 bytes for these 4 insertions, or just 1.46% of the uncompressed 479 bytes.

5.6. Run-Length Compression without Header Compression

In practice, run-length compression can also be applied without relying on header compression. By specifying:

```
Version-Type: peer-counter; text-runs
```

...any peer can receive a series of N insertions as a single update and infer that the string in the update was composed of N separate insertions. For instance, the previous example of four PUT requests can be compressed into a single update without any header compression:

```
PUT /some.txt
Version: "3f84786c-60"
Parents: "3f84786c-56"
Version-Type: peer-counter; text-runs
Content-Length: 4
Content-Range: text 471:474

asdf
```

This single request expresses the change from version 3f84786c-56 to 3f84786c-60, implicitly including the three intermediate versions (3f84786c-57, 3f84786c-58, and 3f84786c-59). Any client or server that understands "Version-Type: peer-counter; text-runs" can infer these intermediate versions when necessary by slicing the content "asdf" to the appropriate length for each version.

For example:

- Version 3f84786c-57 would contain "a"
- Version 3f84786c-58 would contain "as"
- Version 3f84786c-59 would contain "asd"
- Version 3f84786c-60 would contain the full "asdf"

This approach significantly reduces the number of HTTP requests and amount of data required for a series of small, consecutive inserts, while still allowing for precise version control and the ability to reconstruct any intermediate state.

6. Acknowledgements

This is derived from prior draft [Braid-HTTP] with authors:

- Michael Toomim
- Greg Little
- Raphael Walker
- Bryn Bellomy
- Joseph Gentle

And incorporates additional ideas from:

- Rahul Gupta
- Duane Johnson
- Mitar Milutinovic
- Paul Kuchenko

7. Conventions

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

8. IANA Considerations

8.1. HTTP Header Registrations

This document defines the following new HTTP headers, which should be added to the "Permanent Message Header Field Names" registry at <https://www.iana.org/assignments/message-headers>:

Header Field Name: Version
Protocol: http
Status: standard
Reference: This document

Header Field Name: Parents
Protocol: http
Status: standard
Reference: This document

Header Field Name: Version-Type
Protocol: http
Status: standard
Reference: This document

Header Field Name: Current-Version
Protocol: http
Status: standard
Reference: This document

Additionally, this new status code should be added to the "HTTP Status Codes" registry at <https://www.iana.org/assignments/http-status-codes>:

Value: 432
Description: Version Not Found
Reference: This document

8.2. Version Type Registry

The "Version Type Registry" defines the namespace for the version type names and refers to their corresponding specifications. The registry will be created and maintained at <http://www.iana.org/assignments/version-types>.

8.2.1. Procedure

Registration of a Version Type MUST include the following fields:

- Type name
- Required parameters
- Optional parameters
- Description
- Security considerations
- Interoperability considerations
- Obsolete/Non-obsolete status
- Published specification

- Person & email address to contact for further information

Values to be added to this namespace require IETF Review (see [RFC8126], Section 4.1).

8.2.2. Version Type Registrations

[todo: put initial version-type registrations here]

8.2.3. Comments on Version Type Registrations

Comments on registered Version Types may be submitted by members of the community to the IANA at iana@iana.org. These comments will be reviewed by the Version Types reviewer and then passed on to the "owner" of the Version Type if possible. Submitters of comments may request that their comment be attached to the Version Type registration itself; if the IANA, in consultation with the Version Types reviewer, approves, the comment will be made accessible in conjunction with the type registration.

8.2.4. Change Procedures

Once a Version Type has been published by the IANA, the owner may request a change to its definition. The same procedure that would be appropriate for the original registration request is used to process a change request.

Version Type registrations may not be deleted; Version Types that are no longer believed appropriate for use can be declared OBSOLETE; such Version Types will be clearly marked in the list published by the IANA.

Significant changes to a Version Type's definition should be requested only when there are serious omissions or errors in the published specification. When review is required, a change request may be denied if it renders entities that were valid under the previous definition invalid under the new definition.

The owner of a Version Type may pass responsibility to another person or agency by informing the IANA; this can be done without discussion or review.

The IESG may reassign responsibility for a Version Type. The most common case of this will be to enable changes to be made to types where the author of the registration has died, moved out of contact, or is otherwise unable to make changes that are important to the community.

9. Copyright Notice

10. Security Considerations

XXX Todo

11. Authors' Addresses

For more information, the authors of this document are best contacted via Internet mail:

Michael Toomim
Invisible College, Berkeley
2053 Berkeley Way
Berkeley, CA 94704

EMail: toomim@gmail.com
Web: https://invisible.college/@toomim

12. References

12.1. Normative References

[RFC5789] "PATCH Method for HTTP", RFC 5789.

[RFC9110] "HTTP Semantics", RFC 9110.

[Braid-HTTP] draft-toomim-httpbis-braid-http-04

[Merge-Types] draft-toomim-httpbis-merge-types-00

[Updates] draft-toomim-httpbis-updates-[TBD]

[Multiresponse] draft-toomim-httpbis-multiresponse-[TBD]

[Range-Patch] draft-toomim-httpbis-range-patch-00

[Resumable Upload] draft-ietf-httpbis-resumable-upload

12.2. Informative References

[XHR] Van Kesteren, A., Aubourg, J., Song, J., and R. M. Steen, H. "XMLHttpRequest", September 2019. <https://xhr.spec.whatwg.org/>

[SSE] Hickson, I. "Server-Sent Events", W3C Recommendation, February 2015. <https://www.w3.org/TR/2015/REC-eventsource-20150203/>

[Braid-Text] <https://github.com/braid-org/braid-text>