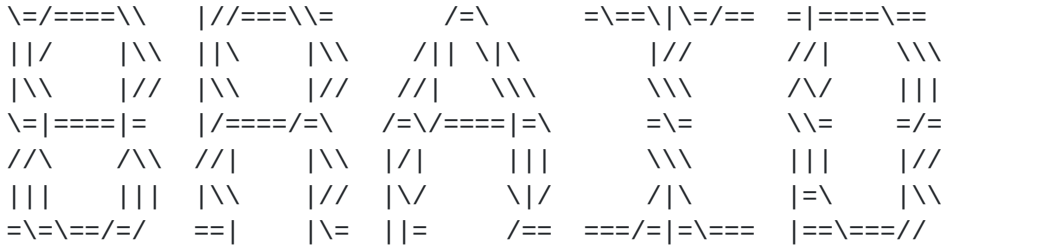
Internet-Draft
Expires: Mar 10, 2024
Intended status: Proposed Standard

M. Toomim
Invisible College
G. Little
Invisible College
R. Walker
Invisible College
B. Bellomy
Invisible College
J. Gentle
Invisible College

Nov 19, 2023



Braid-HTTP: Synchronization for HTTP

draft-toomim-httpbis-braid-http-04

Abstract

Braid is a set of extensions that generalize HTTP from a state *transfer* protocol into a full state *synchronization* protocol.

Braid is composed of four independent extensions to HTTP:

1. VERSIONING of resource history
2. UPDATES sent as patches
3. SUBSCRIPTIONS to updates over time
4. MERGE-TYPES that specify OT or CRDT behavior

Each extension provides a distinct value for a stand-alone use-case. However, they can compose together to support the full power of CRDTs and Operational Transforms on web resources. This allows multiple writers to make simultaneous mutations to arbitrary content-types, under arbitrary network delays and partitions, while guaranteeing consistency across multiple clients and servers. This improves web caching and network performance, and enables natively peer-to-peer, collaboratively-editable, local-first web applications.

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 http://datatracker.ietf.org/drafts/current/.

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

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

1.1. HTTP applications need state Synchronization, not just Transfer

HTTP [RFC9110] transfers a static version of state within a single request and response. If the state changes, HTTP does not automatically update clients with the new versions. This design satisficed when webpages were mostly static and written by hand; however today's websites are dynamic, generated from layers of state in databases, and provide realtime updates across multiple clients and servers. Programmers today need to *synchronize*, not just *transfer* state, and to do this, they must work around HTTP.

The web has a long history of such workarounds. The original web required users to click reload when a page changed. Javascript and XMLHTTPRequest [XHR] made it possible to update just part of a page, running a GET request behind the scenes. However, a GET request still could not push server-initiated updates. To work around this, web programmers would poll the resource with repeated GETs, which was inefficient. Long-polling was invented to reduce redundant requests, but still requires the client to initiate a round-trip for each update. Server-Sent Events [SSE] finally created a standard for the server to push events, but SSE provides semantics of an event-stream, not an update-stream, and SSE programmers must encode the semantics of updating a resource within the event stream. Today there is still no standard to push updates to a resource's state.

In practice, web programmers today often give up on using standards for "data that changes", and instead send custom messages over a WebSocket -- a hand-rolled synchronization protocol. Unfortunately, this forfeits the benefits of HTTP and ReST, such as caching and a uniform interface [REST]. As the web becomes increasingly dynamic, web applications are forced to implement additional layers of non-standard Javascript frameworks to synchronize changes to state.

1.2. Braid-HTTP is four extensions to HTTP

State synchronization implementations come in many forms. The simplest perform one-off transfers of state from one computer to another, but as implementations advance, they may develop support for pushed updates, delivery guarantees, multiple writers, multiple types of edits, expressed as diffs or patches, with automatic conflict resolution, offline modes, merge semantics over multiple content types, and/or function over different network topologies and conditions. Different systems have different needs, and implementations today come in multiple variations, with different network protocols.

Fortunately, it turns out that any implementation's network protocol for these features can be decomposed into the same simple set of four extensions to HTTP semantics:

1. Versioning (Section 2)

   Each resource has a history of changes, ordered in time.

2. Updates as Patches or Snapshots (Section 3)

   Each resource can express updates as either *patches* or *snapshots*; in bidirectional client->server and server->client messages.

3. Subscriptions (Section 4)

   A Subscribe header can be added to GET requests. The server responds by pushing updates to the client while the request is open.

4. Merge Types [MERGE-TYPES]

   If multiple clients and servers simultaneously mutate the same resource, they can guarantee a consistent resulting state by implementing the same Merge Type.

The first three extensions are improvements to existing HTTP features, which are valuable in HTTP on their own, aside from from state synchronization. (For examples, see Section 6.) However, by composing these extensions together, HTTP can genrealize into a full synchronization protocol, and ReST into a synchronization architecture:

```
HTTP:       HyperText  *Transfer* Protocol
becomes:    HyperState *Synchronization* Protocol

ReST:       Representational State *Transfer*
```

becomes: Representational State *Synchronization*

Together, they allow an arbitrary set of clients and servers to make arbitrary mutations to arbitrary resources, under arbitrary network delays and partitions, and merge all mutations consistently, receiving updates as soon as they reconnect. This enables caches to support dynamic content, web applications to feature an offline mode, and textareas to support collaborative editing.

The extensions are explained in the following sections, in turn.

2. Versioning for Resources

Each Braid resource has a current version, and a version history. Versions are specified as a set of one or more strings (called "version IDs") in the Structured Headers [RFC8941] format. Each version ID must be unique, to differentiate distinct changes at distinct points in time.

To specify the version of content in a request or response body, a Version header MAY be included in a request for a PUT, PATCH or POST, or in the response to a GET:

```
Version: "dkn7ov2vwg"
```

Every version also has a set of parents, denoting the version(s) immediately before the version, that it derives from. Any version can be recreated by first merging its parents, and then applying the its update onto that merger. Parents are specified with a Parents header in a PUT/PATCH/POST request or GET response:

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

The full graph of parents forms a Directed Acyclic Graph (DAG), representing the partial order of all versions. A version A is known to have occurred before a version B if and only if A is an ancestor of B in the partial order. Braid time is a DAG, rather than a line.

For any two versions A and B that are specified in a Version or Parents header, A cannot be a descendent of B or vice versa. The ordering of versions in the list carries no meaning.

A Version header is also allowed to contain multiple IDs, to describe the version of a merger:

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

However, any single mutation SHOULD create only a single version ID, and mergers themselves need not be announced over the network when created. They only need to be referenced after the fact, in some situations.

If a client or server does not specify a Version for a resource it transfers, the recipient MAY generate and assign it new version IDs. If a client or server does not specify a Parents header when transferring a new version, the recipient MAY presume that the most recent versions it has (the frontier of time) are the parents of the new version. It MAY also ignore or reject the update.

2.1. Comparison with ETag

The Version header is similar to an ETag, but has two differences:

1. ETags are sensitive to Content-Encoding. If you send the same version with a GZip Content-Encoding, it will have a different ETag, but the same Version.

2. A Version marks a unique point in causal graph time -- not unique content. If a resource is changed from version A to B, and then to C, such that the contents at A are the same as the contents at C, then it is possible for versions A and C to have the same ETag, even though they have different Versions. This can break a CRDT or OT merge algorithm.

Versions can be used in a variety of requests, as we explain next.

2.2. PUT a new version

When a PUT request changes the state of a resource, it can specify the new version of the resource, the parent version IDs that it was directly built upon, and the way multiple simultaneous changes should be merged (the "Merge-Type"):

Request:

```
PUT /chat
Version: "ej4lhb9z78"                              | Update
Parents: "oakwn5b8qh", "uc9zwhw7mf"                |
Content-Type: application/json                     |
Merge-Type: sync9                                  |
Content-Length: 64                                 |
                                                   |
[{"text": "Hi, everyone!",                         | | Snapshot
  "author": {"link": "/user/tommy"}}]              | |
```

Response:

```
HTTP/1.1 200 OK
```

Merge-Types are specified in [MERGE-TYPES]. The Version and Parents headers are optional. If Version is omitted, the recipient may assign new version IDs. If Parents is omitted, the recipient may assume that its current version is the version's parents.

As will be explained in Section 3, we call the set of data that updates a resource from one version to another an "update". An update consists of a set of headers and a body. In this example, the update includes a snapshot of the entire new value of the resource. However, one can also specify the update as a set of patches.

2.3. 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"                                | Update
Parents: "oakwn5b8qh", "uc9zwhw7mf"                  |
Content-Type: application/json                       |
Merge-Type: sync9                                    |
Content-Length: 64                                   |
                                                     |
[{"text": "Hi, everyone!",                           | | Snapshot
  "author": {"link": "/user/tommy"}}]                | |
```

2.4. 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/1.1 200 OK
   Current-Version: "3"

   Version: "2"                                          | Update
   Parents: "1a", "1b"                                   |
   Content-Type: application/json                        |
   Merge-Type: sync9                                     |
   Content-Length: 64                                    |
                                                         |
   [{"text": "Hi, everyone!",                            | | Snapshot
     "author": {"link": "/user/tommy"}}]                 | |

   Version: "3"                                          | Update
   Parents: "2"                                          |
   Content-Type: application/json                        |
   Merge-Type: sync9                                     |
   Content-Length: 117                                   |
                                                         |
   [{"text": "Hi, everyone!",                            | | Snapshot
     "author": {"link": "/user/tommy"}}                  | |
    {"text": "Yo!",                                      | |
     "author": {"link": "/user/yobot"}]                  | |
```

To express a range of updates, the response body contains a sequence of updates; each with its own content-length. The format of this sequence is defined in the upcoming (Section 4.2) on Subscriptions.

2.5. Rules for Version and Parents headers

If a GET request contains a Version header:

- The Subscribe header (Section 4) MUST be absent.

- 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 request contains a Subscribe header, then it SHOULD additionally leave the request open and subscribe the client to future updates. Otherwise, it should close the connection after sending the updates.

- 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 MAY refactor or rebase the version history that it provides to a client, so long as it does not affect the resulting state, or the result of merges using the history. (Rules for specifying constraints on such rebases are out of scope for this draft.)

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 may respond using an error code that will be defined in a subsequent version of this draft.

3. Updates as Patches or Snapshots

Whereas today's HTTP sends the current version of a resource as a "snapshot" in the body of a GET response or PUT request, and allows clients to send a "patch" in a PATCH request, a general state synchronization protocol needs updates that travel in both directions (both server->client and client->server) and across multiple methods.

This section describes a general form for bidirectional updates. Updates can be sent as snapshots or patches. When sent as patches, a single update can contain a single patch, or multiple patches. Unlike the PATCH method, these updates can be sent idempotently by including versioning information -- a client or server that receives the same update twice, for the same version, can discard the second update, and thus maintain idempotence.

There are two reasons to send an update as a patch:

- Patches can be smaller and more efficient

- Patches articulate *how* changes occur, which enables Merge-Types to intelligently merge, e.g. in collaborative editing.

There are two ways to express patches:

- Custom patch Content-Types: As defined in HTTP PATCH [RFC5789], a custom patch format can be specified as a Content-Type. Any such patch can be included in any Braid-HTTP update, by adding its Content-Type to its update or patch header.

- Range Patches: If a Content-Range header is specified on the

update or patch, then it defines the region of the document that is being replaced by the content, as specified in [RANGE-PATCH].

Every patch MUST include either a Content-Type or a Content-Range.

Finally, it is possible to include multiple patches within a single update by including a "Patches: N" header, and setting the body to a concatenation of that many patches.

These scenarios are elaborated below.

3.1. PUT an update as a patch

A single Range Patch (see [RANGE-PATCH]) can be constructed as a Partial PUT [RFC9110] -- a PUT with a Content-Range header:

Request:

```
PUT /chat
Version: "g09ur8z74r"                              | Update
Parents: "ej4lhb9z78"                              |
Content-Type: application/json                     |
Merge-Type: sync9                                  |
Content-Length: 53                                 | | Patch
Content-Range: json .messages[1:1]                 | |
                                                   | |
[{"text": "Yo!",                                   | |
  "author": {"link": "/user/yobot"}]               | |
```

Response:

```
HTTP/1.1 200 OK
```

Note that Partial PUTs can result in data loss if sent to a server that does not support them. Legacy servers ignore headers they do not understand, and will interpret the patch as the entire resource, replacing the resource's state with just the region being patched.

To avoid this, only use Partial PUTs on servers that you know support them, or do feature detection (to be defined in a subsequent draft) to determine what they support. You can also reformulate a Partial PUT with Patches: 1 as in (Section 3.4) below, as a safe alternative.

3.2. GET an update as a patch

A GET response can also express a patch, using Content-Range:

Request:

```
GET /chat
Version: "g09ur8z74r"
Parents: "ej4lhb9z78"
```

Response:

```
HTTP/1.1 200 OK
Content-Type: application/json                    | Update
Merge-Type: sync9                                 |
Content-Length: 53                                | | Patch
Content-Range: json .messages[1:1]                | |
                                                  | |
[{"text": "Yo!",                                  | |
  "author": {"link": "/user/yobot"}]              | |
```

Or a custom patch type, using Content-Type:

Response:

```
HTTP/1.1 200 OK
Merge-Type: sync9                                 | Update
Content-Length: 53                                | | Patch
Content-Type: application/json-patch+json         | |
                                                  | |
[                                                 | |
 {"op": "test", "path": "/a/b/c", "value": "foo"}, | |
 {"op": "remove", "path": "/a/b/c"},              | |
 {"op": "add", "path": "/a/b/c", "value": []},    | |
 {"op": "replace", "path": "/a/b/c", "value": 42}, | |
 {"op": "move", "from": "/a/b", "path": "/a/d"},  | |
 {"op": "copy", "from": "/a/d", "path": "/a/d/e"} | |
]                                                 | |
```

3.3. PUT an update as a set of patches

To format an update as a set of patches, include a header called "Patches" and assign it to the number of patches included, and format those patches in the body as a sequence separated by blank lines:

Request:

```
PUT /chat
Version: "g09ur8z74r"                             | Update
Parents: "ej4lhb9z78"                             |
Content-Length: 189                               |
```

```
      Content-Type: application/json                      |
      Merge-Type: sync9                                   |
      Patches: 2                                          |
                                                          |
      Content-Length: 53                                  | | Patch
      Content-Range: json .messages[1:1]                  | |
                                                          | |
      [{"text": "Yo!",                                    | |
        "author": {"link": "/user/yobot"}]                | |
                                                          |
      Content-Length: 40                                  | | Patch
      Content-Range: json .latest_change                  | |
                                                          | |
      {"type": "date", "value": 1573952202370}            | |
```

Response:

```
      HTTP/1.1 200 OK
```

To distinguish the boundaries between patches in an update, each patch MUST include the following header:

```
      Content-Length: N
```

This length determines where each patch ends, and next begins.

3.4. Using Patches: 1 for safe Partial PUTs

The "Patches" header also provides a safe alternative to Partial PUTs. Instead of specifying the patch in the request body of the PUT, the request can:

- set the "Patches: 1" header
- move the "Content-Length" and "Content-Range" headers into patch headers on the request body

For example, the Partial PUT example of Section 3.1 would translate like this:

Request:

```
      PUT /chat
      Version: "g09ur8z74r"                               | Update
      Parents: "ej4lhb9z78"                               |
      Content-Type: application/json                      |
      Merge-Type: sync9                                   |
      Patches: 1                                          |
```

```
                                                    |
   Content-Length: 53                               | | Patch
   Content-Range: json .messages[1:1]               | |
                                                    | |
   [{"text": "Yo!",                                 | |
     "author": {"link": "/user/yobot"}]             | |
```

Response:

```
   HTTP/1.1 200 OK
```

Because this PUT request has no Content-Length, legacy servers will not know where the request body ends, and will not recognize a complete PUT. (Any server that accepts a PUT before it is complete also risks data corruption from network failures during a PUT.) Servers that understand Patches: 1, on the other hand, know to look at the patch inside to find the true Content-Length, and will be able to complete the request.

3.5. PUT an update with a custom patch-type

Since PATCH is not idempotent, a client may want to send a patch idempotently using a PUT. The client SHOULD include a Version and Parents header to ensure idempotency. The server SHOULD discard duplicate patches (for the same Version) to satisfy idempotence.

Request:

```
   PUT /chat
   Version: "up12vyc5ib"                            | Update
   Parents: "2bcbi84nsp"                            |
   Content-Length: 371                              |
   Merge-Type: sync9                                |
   Patches: 1                                       |
                                                    |
   Content-Length: 288                              | | Patch
   Content-Type: application/json-patch+json        | |
                                                    | |
   [                                                | |
    {"op": "test", "path": "/a/b/c", "value": "foo"}, | |
    {"op": "remove", "path": "/a/b/c"},             | |
    {"op": "add", "path": "/a/b/c", "value": []},   | |
    {"op": "replace", "path": "/a/b/c", "value": 42}, | |
    {"op": "move", "from": "/a/b", "path": "/a/d"}, | |
    {"op": "copy", "from": "/a/d", "path": "/a/d/e"} | |
   ]                                                | |
```

Response:

```
HTTP/1.1 200 OK
```

4. Subscriptions for GET

If a GET request includes the Subscribe header, the server can "subscribe" the client to the resource, which means it promises to keep it up-to-date with the updates necessary to describe new versions as it learns about them.

The server will first send the current version, and then stream the updates for future versions. Each update can express the new content either as a snapshot, or a set of patches, as in Section 3.

Request:

```
GET /chat
Subscribe:
```

Response:

```
HTTP/1.1 209 Subscription
Subscribe:

Version: "ej4lhb9z78"                              | Update
Parents: "oakwn5b8qh", "uc9zwhw7mf"                |
Content-Type: application/json                     |
Merge-Type: sync9                                  |
Content-Length: 64                                 |
                                                   |
[{"text": "Hi, everyone!",                         | | Snapshot
  "author": {"link": "/user/tommy"}}]              | |

Version: "g09ur8z74r"                              | Update
Parents: "ej4lhb9z78"                              |
Content-Type: application/json                     |
Merge-Type: sync9                                  |
Patches: 1                                         |
                                                   |
Content-Length: 53                                 | | Patch
Content-Range: json .messages[1:1]                 | |
                                                   | |
[{"text": "Yo!",                                   | |
  "author": {"link": "/user/yobot"}]               | |

Version: "2bcbi84nsp"                              | Update
Parents: "g09ur8z74r"                              |
Content-Type: application/json                     |
Merge-Type: sync9                                  |
Patches: 1                                         |
```

```
                                                    |
Content-Length: 58                                  | | Patch
Content-Range: json .messages[2:2]                  | |
                                                    | |
[{"text": "Hi, Tommy!",                             | |
  "author": {"link": "/user/sal"}}]                 | |

Version: "up12vyc5ib"                               | Update
Parents: "2bcbi84nsp"                               |
Content-Type: application/json                      |
Merge-Type: sync9                                   |
Patches: 1                                          |
                                                    |
Content-Length: 288                                 | | Patch
Content-Type: application/json-patch+json           | |
                                                    | |
[                                                   | |
 {"op": "test", "path": "/a/b/c", "value": "foo"},  | |
 {"op": "remove", "path": "/a/b/c"},                | |
 {"op": "add", "path": "/a/b/c", "value": []},      | |
 {"op": "replace", "path": "/a/b/c", "value": 42},  | |
 {"op": "move", "from": "/a/b", "path": "/a/d"},    | |
 {"op": "copy", "from": "/a/d", "path": "/a/d/e"}   | |
]                                                   | |
```

Note the blank line after the "Subscribe:" header. This allows subscriptions to be smuggled through existing HTTP clients (such as browser fetch() APIs), which will interpret the sequence of updates as a long response body of unspecified length. Application programmers can then parse the updates via polyfill libraries on top of existing client libraries. In future versions of this draft, it could be advantageous to remove the blank line for upgraded clients, which can interpret the initial response headers and body as the headers and body for the first update, directly.

It is RECOMMENDED that updates do not change the Merge-Type for a resource, because there is no defined semantics for merging updates of different Merge-Types. If a client observes a change in Merge-Type from a server, it is suggested to reload the resource.

4.1. Creating a Subscription

A client requests a subscription by issuing a GET request with a Subscribe header:

```
Subscribe: <Parameters>
```

<Parameters> may be blank, set to "true", or contain arbitrary data, and is reserved for future use.

This header modifies the normal GET method's semantics, to request a subscription to future updates to the data, rather than only returning the current version of the representation data.

A server implementing Subscribe MUST include a Subscribe header in its response. The server then SHOULD keep the connection open, and send updates over it.

In general, a server that implements subscriptions promises to keep its subscribed clients up-to-date by sending changes until the connection is closed. Once closed, a subscription can be resumed by the client issuing a subsequent GET request on the same document.

4.2. Sending multiple updates per GET

To send multiple updates, a server concatenates multiple updates into a single response body. Each update MUST include headers and a body, and MUST specify the end of its body by including at least one of the following headers:

- Content-Length: N
- Patches: N

The body may be zero-length. A server MAY separate each update with one or more blank lines. These lines do not count towards Content-Length. They can by used to visually separate updates, or to guide the behavior of certain proxies or clients:

1. Certain clients or proxies close inactive connections. A server signal that a connection is still active by periodically sending additional blank lines between updates.

2. Some clients (e.g. Firefox) only flush incoming data after a receiving a chunk of a certain size. A server can ensure small updates get flushed by padding them with blank lines.

4.3. Continuing a Subscription

Once closed, a Braid subscription may be restarted by the client issuing a new subscription request.

When the client reconnects, it may specify its last known version using the Parents header. The server SHOULD then send only the updates since that version.

Example:

Initial request:

```
   GET /chat
   Subscribe:

Initial response:

   HTTP/1.1 209 Subscription
   Subscribe:

   Version: "ej4lhb9z78"                                | Update
   Content-Type: application/json                       |
   Content-Length: 64                                   |
                                                        |
   [{"text": "Hi, everyone!",                           | | Snapshot
     "author": {"link": "/user/tommy"}}]                | |

<Client disconnects>

Reconnection request:

   GET /chat
   Subscribe:
   Parents: "ej4lhb9z78"

Reconnection response:

   HTTP/1.1 209 Subscription
   Subscribe:

   Version: "g09ur8z74r"                                | Update
   Parents: "ej4lhb9z78"                                |
   Content-Type: application/json                       |
   Merge-Type: sync9                                    |
   Patches: 1                                           |
                                                        |
   Content-Length: 53                                   | | Patch
   Content-Range: json .messages[1:1]                   | |
                                                        | |
   [{"text": "Yo!",                                     | |
     "author": {"link": "/user/yobot"}]                 | |
```

4.4. Signaling "all caught up"

When starting or resuming a subscription, the server can indicate which version is current by specifying a "Current-Version" header before starting the stream of versions. This should contain the frontier of time -- the leaves of the currently-known time DAG. The client can use this information to determine when it has caught up with the server's version at the time it received the client's request.

```
Request:

   GET /chat
   Subscribe:

Response:

   HTTP/1.1 209 Subscription
   Subscribe:
   Current-Version: "ej4lhb9z78"              <-- Current Version

   Version: "b9z78ej4lh"                     | Updates
   Content-Type: application/json            |
   Merge-Type: sync9                         |
   Content-Length: 2                         |
                                             |
   []                                        |
                                             |
   Version: "ej4lhb9z78"                     | <-- Current Version
   Parents: "b9z78ej4lh"                     |
   Content-Type: application/json            |
   Merge-Type: sync9                         |
   Content-Length: 64                        |
                                             |
   [{"text": "Hi, everyone!",                |
     "author": {"link": "/user/tommy"}}]     V
                                               <-- Now caught up
```

4.5. Errors

If a server has dropped the history that a client requests, the server can return a 410 GONE response, to tell the client "sorry, I don't have the history necessary to synchronize with you."

5. Design Goals

This spec is designed to be:

1. Backwards-compatible with existing HTTP

2. Easy to implement simple synchronizers with. For instance, it should be easy to write a read-only synchronizer for an append-only log.

3. Possible to implement arbitrary synchronization algorithms. For instance, these extensions support any Operational Transform or CRDT algorithm.

6. Example Use Cases

The first three Braid extensions (Versions, Patches, and Subscriptions) are useful independently -- without Merge-Types, and with or without each other.

The first section 6.1. gives examples of independent uses. Then 6.2. puts them together into more advanced combinations.

6.1. Basic Examples

First, we give examples of Subscriptions, Versions, and Patches used individually.

6.1.1. Subscribing to the current temperature

Subscriptions are useful independently of versioning, patches, or merging. Suppose that a web server hosts the current temperature:

```
Request:

  GET /temperature

Response:

  HTTP/1.1 200 OK
  Content-Length: 4

  70 F
```

Braid Subscriptions enable a client can stay up-to-date as the temperature changes:

```
Request:

  GET /temperature
  Subscribe: true

Response:

  HTTP/1.1 209 Subscription
  Subscribe: true

  Content-Length: 4

  70 F

  Content-Length: 4

  72 F
```

```
Content-Length: 4

73 F

Content-Length: 4

71 F
```

6.1.2. Versioning of source code

Source code is often hosted at URIs with embedded versions, such as:

https://code.jquery.com/jquery-3.7.1.js

However, doing so sacrifices having a universal name for the resource, as each version gets its own URI. there is no standard way to know which URIs are the same resource.

Braid versions can instead be specified with the Version: header:

Request:

```
GET /jquery.js
Version: "3.7.1"
```

This provides a standard way for clients to access versions of a resource programmatically. One advantage of this approach is that the same URI can be used by the programmers to edit the code, via full-featured version control (over Braid), as is used to import the code as library into other code.

6.1.3. Patches can append to server logs

Patches are useful for a variety of stand-alone use-cases. For instance, imagine a server in a datacenter that holds system logs. A client might want to append a notification to the log, and could do so with a simple Range-Patch like so:

Request:

```
PUT /logs
Content-Range: lines -
Content-Length: 53
Content-Type: text/plain

Notification: process 7726 OOM error on host "buster"
```

This range-patch uses the "lines" unit (specified in [RANGE-PATCH]) at the special position "-" which represents the new line at the end of the file.

6.2. Combination examples

6.2.1. Resumeable uploads

Braid semantics are expressive enough to implement resumeable uploads. This provides an alternative approach to the work in [draft-ietf-httpbis-resumable-upload].

6.2.1.1. Version-Type: bytestream

The uploading resource can be considered an append-only bytestream. We can declare this with a header:

```
Version-Type: bytestream
```

"Bytestream" versions look like:

```
Version: "x82ha-344"
```

Which refers to "the resource after appending 344 bytes by agent `x82ha`".

This creates a direct correspondence to time and space -- at each timestep, the size of the document is one byte longer.

6.2.1.2. Protocol for resuming uploads

A client starts an upload by simply specifying this special Version-Type, along with the version it will reach when it's complete, via the Current-Version header:

```
Request:

  PUT /something
  Current-Version: "abwejf-900"
  Version-Type: bytestream
  Content-Length: 900

  <binary data of length 900>
```

Now if the upload succeeds, the server will give a 200 OK as usual:

```
Response:

  200 OK
```

However, if the upload fails mid-stream, the client will want to re-initiate the upload. It first asks the header how much has been received by issueing a HEAD request, and seeing what version of the bytestream the the server is at:

Request:

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

(A) If the server has received everything, it will report a 200 OK at the final version:

Response:

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

(B) If server has only received a part, it will respond with a version partway through the bytestream:

Response:

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

(C) If the server has received nothing, it will not have any versions of the bytestream starting with the version "abwejf-0" initiated by the client:

Response:

```
416 Range Not Satisfiable
```

The client now responds accordingly:

- In case (A), the client is done.
- In case (B), the client resumes the upload as follows:

Request:

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

<binary data from 400-900>
```

- In case (C), the client restarts the upload from scratch.

This simple protocol allows the functionality of resumeable uploads to be implemented on top of existing synchronization semantics and libraries. A server that implements Braid-HTTP along with the "bytestream" version-type can support resumeable uploads for free.

6.2.2. Dynamic Resources: Animating a PNG

Braid allows resources to become inherently dynamic -- able to change over time. You can use this to make a resource animate.

In this example, a server streams changes to a PNG file in a sequence of patches. When the client renders the new state of the PNG after each patch, a new frame of animation is displayed.

Request:

```
GET /animated-braid.png
Subscribe
```

Response:

```
HTTP/1.1 209 Subscribe
Subscribe
Content-Type: image/png          | Update
Content-Length: 170763           |
                                 |
<binary data>                    | | Snapshot

Content-Type: image/png          | Update
Patches: 2                       |
                                 |
Content-Length: 1239             | | Patch
Content-Range: bytes 100-200     | |
                                 | |
<binary data>                    | |
                                 |
Content-Length: 62638            | | Patch
Content-Range: bytes 348-887     | |
                                 | |
<binary data>                    | |
```

6.2.3. Dynamic proxies and caches

Since updates aren't pushed, today's web often uses timeouts to trigger a cache becoming stale. Unfortunately, sometimes the timeout is wrong, and caches become out-of-date, and we have to wait for an unknown cache to timeout before we can see the new version of

something. As a result, programmers have learned to force-reload pages habitually, and caches become less efficient than necessary.

A cache supporting the Braid extensions, however, will automatically update whenever a change occurs. If a client starts a GET Subscription with a proxy, the proxy will then start and maintain a GET Subscription with the origin server for that resource. The origin server will promise to send the proxy updates over its GET Subscription, and the proxy will then relay these changes to all connected clients. If a set of clients and servers all support Braid, they will never need to force-reload caches for any data amongst them.

6.2.4. Serverless chat example

A Braid web application can operate offline. A user can use the app from an airplane, and their edits can synchronize when they regain internet connections. Additionally, the Braid protocol can be expressed over peer-to-peer transports (e.g. WebRTC) to support a a a a peer-to-peer synchronization without a server. For example, a chat application might be served and synchronized on Braid-HTTP, while also establishing redundant peer-to-peer connections on WebRTC, and translating all Braid-HTTP messages over the WebRTC connections, and vice versa. The server could then be shut down, and users of the chat app could continue to send messages to one another.

Imagine the server serves the current set of trusted clients' IP addresses at the /peers state. Each client then subscribes to the /peers state with:

```
GET /peers
Subscribe:
-------
[ {ip: '13.55.32.158', pubkey: 'x371...8382'},
  {ip: '244.38.55.83', pubkey: 'o2u8...2s73'},
  ...
]
```

Each peer can then choose a set of those peers with whom to establish a WebRTC connection. It will then exchange Braid messages with those peers over that connection.

7. Related Work

7.1. Existing IETF Standards

A number of IETF specifications already standardize aspects of synchronization for specific domains. IMAP [RFC9051] provides synchronization of email. WebDAV provides the synchronization of "collections" [RFC6578], and has been extended specifically for

calendar data in CalDAV [RFC4791], and vCards in [RFC6350]. More recently, JMAP [RFC8620] provides an updated method of synchronization, supporting mail, calendars, and contacts.

7.2. IETF Work in Progress

We wish to integrate this work with the excellent related efforts already underway:

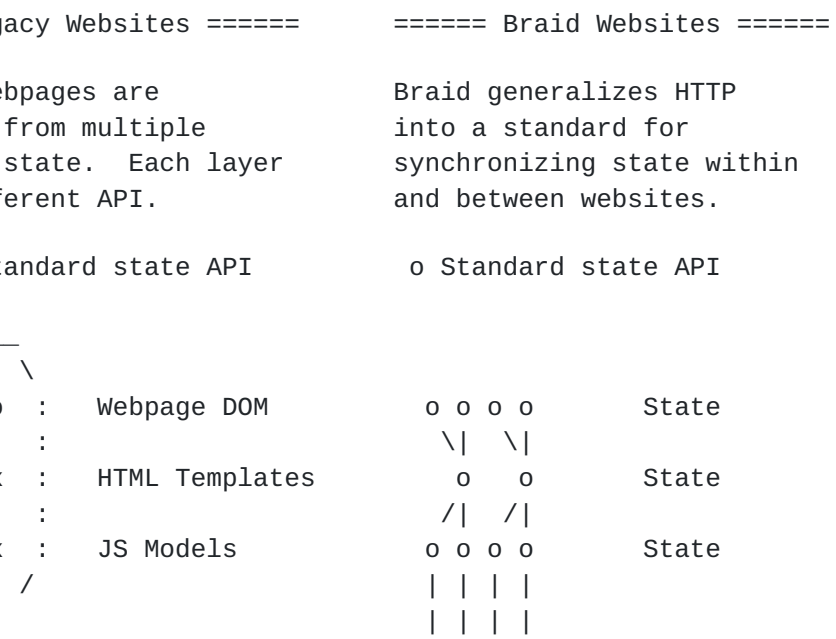
- Per Resource Events [draft-gupta-httpbis-per-resource-events] also provides subscriptions for HTTP resources, and also provides a mechanism to define types of updates, including patches.

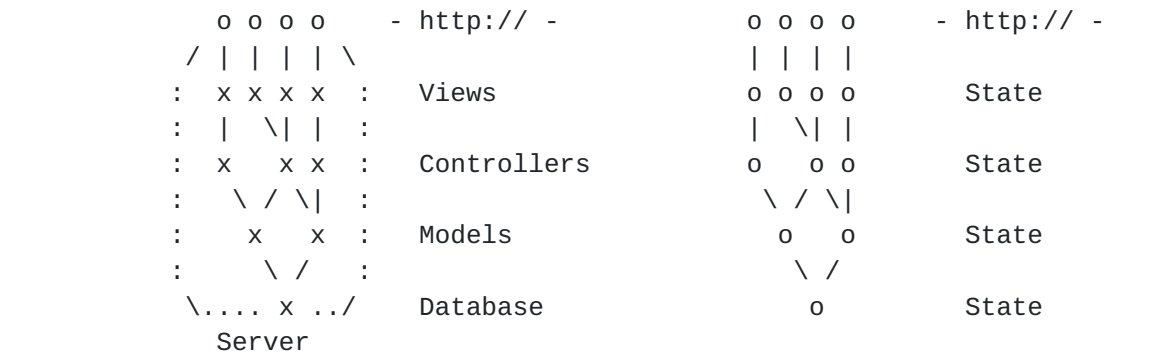
- Resumable Uploads [draft-ietf-httpbis-resumable-upload] can be expressed in Braid-HTTP as a sequence of patches, from client to server. Each patch can specify a version in an "uploading" branch of time. Once the upload is complete, the branch can be "merged" back into the main version history. A subsequent version of this draft will provide examples.

- Byte-Range-Patch [draft-wright-http-patch-byterange] also enables general patches using Content-Range, and has useful mechanisms to avoid Partial PUTs on legacy servers.

7.3. Web Frameworks

Web applications typically synchronize the state of a client and server with layers of models, views, and controllers in web frameworks. By automating synchronization within HTTP, programmers have to write fewer layers of code on top of it.





Today's programmers have to learn each API, and wire them together, making sure that changes to shared state synchronize across all layers and computers.

Each piece of Braid state (o) has a URI; whether public or internal. State can be a function of other state, and and automatically recompute when its dependencies change. Braid guarantees network synchronization.

8. IANA Considerations

8.1. Header Field Registration

HTTP header fields are registered within the "Hypertext Transfer Protocol (HTTP) Field Name" registry maintained at <https://www.iana.org/assignments/http-fields>.

This document defines the following HTTP header fields, so their associated registry entries have been updated according to the permanent registrations below (see [BCP90]):

Header Field Name	Status	Reference
Version	experimental	Section 2
Parents	experimental	Section 2
Merge-Type	experimental	Section 2.2
Patches	experimental	Section 3.3
Subscribe	experimental	Section 4
Current-Version	experimental	Section 4.4

The change controller is: "IETF (iesg@ietf.org) - Internet Engineering Task Force".

9. Security Considerations

XXX Todo

10. 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].

11. Copyright Notice

12. References

12.1. Normative References

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

[RFC9110] "HTTP Semantics", RFC 9110.

[RFC9111] "HTTP Caching", RFC 9111.

[RFC9112] "HTTP/1.1", RFC 9112.

[RFC8941] "Structured Field Values for HTTP"

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[MERGE-TYPES] draft-toomim-httpbis-merge-types-00

[RANGE-PATCH] draft-toomim-httpbis-range-patch-00

12.2. Informative References

[XHR] Van Kestern, 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/>

[REST] Fielding, R. "Architectural Styles and the Design of Network-based Software Architectures" Doctoral dissertation, University of California, Irvine, 2000.

[RFC9051] Melnikov, Ed., Leiba, Ed., "Internet Message Access Protocol - Version 4rev2", RFC 9051, DOI 10.17487/RFC9051, August 2021, <https://www.rfc-editor.org/info/rfc9051>.

[RFC6578] Daboo, C., Quillaud, A., "Collection Synchronization for Web Distributed Authoring and Versioning (WebDAV)", RFC 6578, DOI 10.17487/RFC6578, March 2012, <https://www.rfc-editor.org/info/rfc6578>.

[RFC4791] Daboo, C., Desruisseaux, B., Dusseault, L., "Calendaring Extensions to WebDAV (CalDAV)", RFC 4791, DOI 10.17487/RFC4791, March 2007, <https://www.rfc-editor.org/info/rfc4791>.

[RFC6350] Perreault, S., "vCard Format Specification", RFC 6350, DOI 10.17487/RFC6350, August 2011, <https://www.rfc-editor.org/info/rfc6350>.

[RFC8620] Jenkins, N., Newman, C., "The JSON Meta Application Protocol (JMAP)", RFC 8620, DOI 10.17487/RFC8620, July 2019, <https://www.rfc-editor.org/info/rfc8620>.

[RFC6902] Bryan, P., Nottingham, M., "Javascript Object Notation (JSON) Patch", RFC 6902.

[RFC9110] Fielding, R., Nottingham, M., Reschke, J., "HTTP Semantics", RFC 9110

[BCP90] Klyne, G., Nottingham, M., and J. Mogul, "Registration Procedures for Message Header Fields", BCP 90, RFC 3864, September 2004.

13. Acknowledgements

In addition to the authors, this spec contains intellectual contributions from the following people:

- Mitar Milutinovic
- Sarah Allen
- Duane Johnson
- Travis Kriplean
- Marius Nita

- Paul Pham
- Morgan Dixon
- Karthik Palaniappan

We thank the following people for key feedback on previous drafts:

- Austin Wright
- Martin Thomson
- Eric Kinnear
- Olli Vanhoja
- Julian Reschke
- Chris Lemmons
- Rahul Gupta

We also thank Mark Nottingham, Fred Baker, Adam Roach, and Barry Leiba for facilitating a productive environment within the IETF.

14. 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

Greg Little
Invisible College, Berkeley
2053 Berkeley Way
Berkeley, CA 94704

EMail: glittle@gmail.com
Web: https://glittle.org/

Rafie Walker
Bard College

EMail: slickytail.mc@gmail.com

Bryn Bellomy
Invisible College, Berkeley
2053 Berkeley Way
Berkeley, CA 94704

EMail: bryn@signals.io
Web: https://invisible.college/@bryn

Joseph Gentle
Invisible College, Berkeley
2053 Berkeley Way
Berkeley, CA 94704

EMail: me@josephg.com
Web: https://josephg.com/