Internet-Draft
Expires: Jan 8, 2020
Intended status: Proposed Standard

M. Toomim
Invisible College
R. Walker
Invisible College

July 8, 2019

The Braid Protocol: Synchronization for HTTP
draft-toomim-braid-00

Abstract

Braid is a proposal for a new version of HTTP that transforms it from a state *transfer* protocol into a state *synchronization* protocol. Braid puts the power of Operational Transform and CRDTs onto the web, improving network performance and robustness, and enabling peer-to-peer web applications.

At the same time, Braid creates an open standard for the dynamic internal state of websites. Programmers can access state uniformly, whether local or on another website. This creates a separation of UI from State, and allows any user to edit or choose their own UI for any website's state.

We have a working prototype of the Braid, and have deployed it with production websites. This memo describes the protocol, how it differs from prior versions of HTTP, and a plan to deploy it in a backwards-compatible way, where web developers can opt into the new synchronization features without breaking the rest of the web.

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

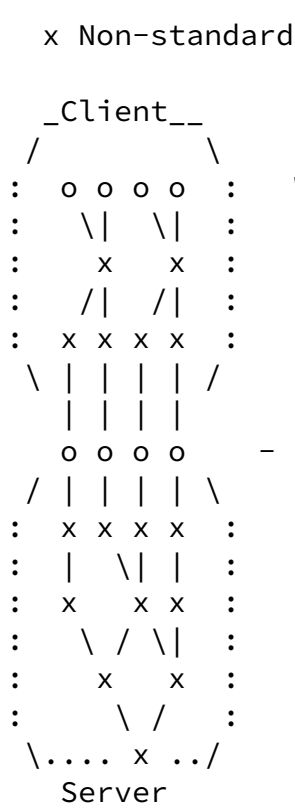
1. Introduction

HTTP was initially designed to transfer static pages. If a page changes, it is the client's responsibility to issue another GET request. This made sense when pages were static and written by hand. However, today's websites are dynamic, generated from databases, and continuously mutate as their state changes. Now we need state *synchronization*, not just state *transfer*.

Unfortunately, there is no standard way to synchronize. Instead, programmers write non-standard code; wiring together custom protocols over WebSockets and long-polling XMLHTTPrequests with stacks of Javascript frameworks. The task of connecting a UI with data is one that every dynamic website has to do, but there is no standard way to do it.

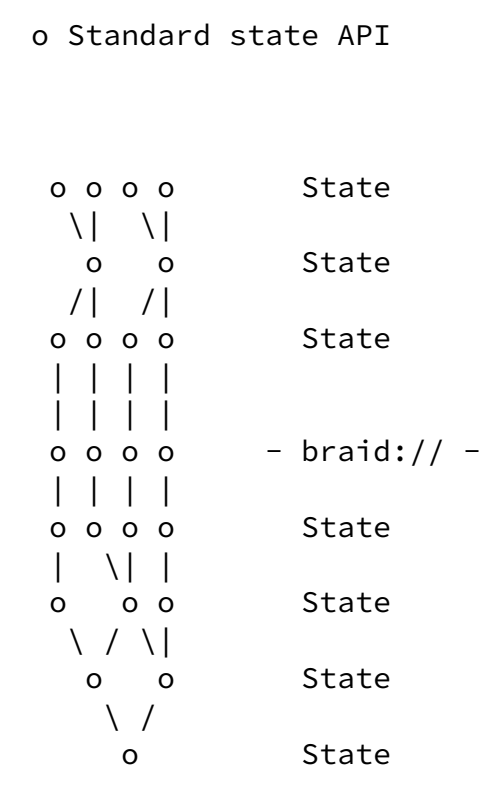
======= HTTP Websites =======	====== Braid Websites ======
Today's websites are generated from multiple layers of state across multiple computers. Each	Braid generalizes HTTP and REST into a uniform standard that synchronizes state within and between dynamic

layer has a different API.



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.

websites.



On the braid, each piece of state (o) has a URL; whether public, or internal to a client or server. Any state can be a function of other state, and dynamically recomputes when its dependencies change. Braid guarantees the network will synchronize.

As the web becomes more dynamic and data-driven, the complexity of the non-standard Javascript stack grows, and an increasing amount of data is inaccessible to the open web. The result is a web which is open on the surface, but closed internally: websites can link to each other's *pages*, but cannot easily share each other's internal *state*.

We can solve this by generalizing HTTP into a *synchronization* protocol, which replaces the complex Javascript stack, while providing new features, and making website internal state accessible anywhere desired, and realtime synchronized by default.

We have a working prototype of the Braid protocol, and have deployed it with production websites. The prototype is implemented as a polyfill library, which adds Braid features to existing browsers and

servers.

This document describes the new protocol, how it differs from prior versions of HTTP, and a plan to deploy it in a backwards-compatible way, where web developers can opt into the new synchronization features without breaking the rest of the web.

2. Synchronization

Braid incorporates the abilities of Operational Transform and CRDTs. These are approaches to solving *synchronization*.

Synchronization is a problem that occurs whenever two or more computers or threads access the same state. Synchronization code is tricky to write, and can result in clobbers, corruptions, or race conditions.

This is a challenging problem, which has seen a number of partial attempts in HTTP, such as e-tags, cache control, PATCH, JSON-diff, and SSE.

Luckily, a set of maturing synchronization technologies (such as Operational Transform and CRDTs) can now automate and encapsulate synchronization within a library. They can synchronize arbitrary JSON data structures across an arbitrary set of computers that make arbitrary mutations, and consistently merge their edits into a valid result, without a central server, in the face of arbitrary network delays and dropouts. In other words, it is now possible to interact with state stored anywhere on a network as if it is a local variable, and program as if it is already downloaded and always up-to-date.

Unfortunately, each synchronizer implements a different protocol, with a different set of features and tradeoffs. Braid proposes a common language for synchronizers, so that they can interoperate, and implements it as an extension to HTTP.

This lets multiple synchronizers interoperate, if they agree on a way to consistently resolve ambiguities -- a *merge type*. We have run tests that succesfully interoperate a CRDT and OT system over the common Braid protocol.

When applying synchronization to the web, we see the power of synchronization manifest in these concrete ways:

- Caches update automatically and instantly, because servers promise to push changes to their subscribers. This obsoletes the `cache-control` and `refresh` headers, and the `max-age`

heuristic. Users never need to force-clear their cache.

- Updates go over the network as diffs, which can be much smaller than the resources they modify, significantly reducing network usage.

- Web apps get an offline mode for free. Edits from multiple clients merge automatically once they come online. Network failures recover transparently.

- *Reload* buttons in browsers become unnecessary, and can be removed for braid sites. Browsers automatically discover and display the most recent version on their own.

- Web apps require roughly 70% less code to build (in our experiments), because programmers do not need web frameworks or custom logic to wire together a stack of server-state and client-state. This work is automated by the protocol.

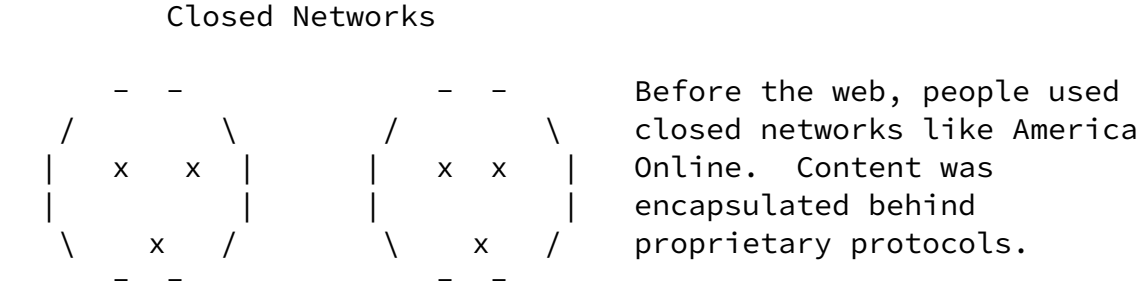
- Every <textarea> can become a collaborative editor (like Google Docs) for free.

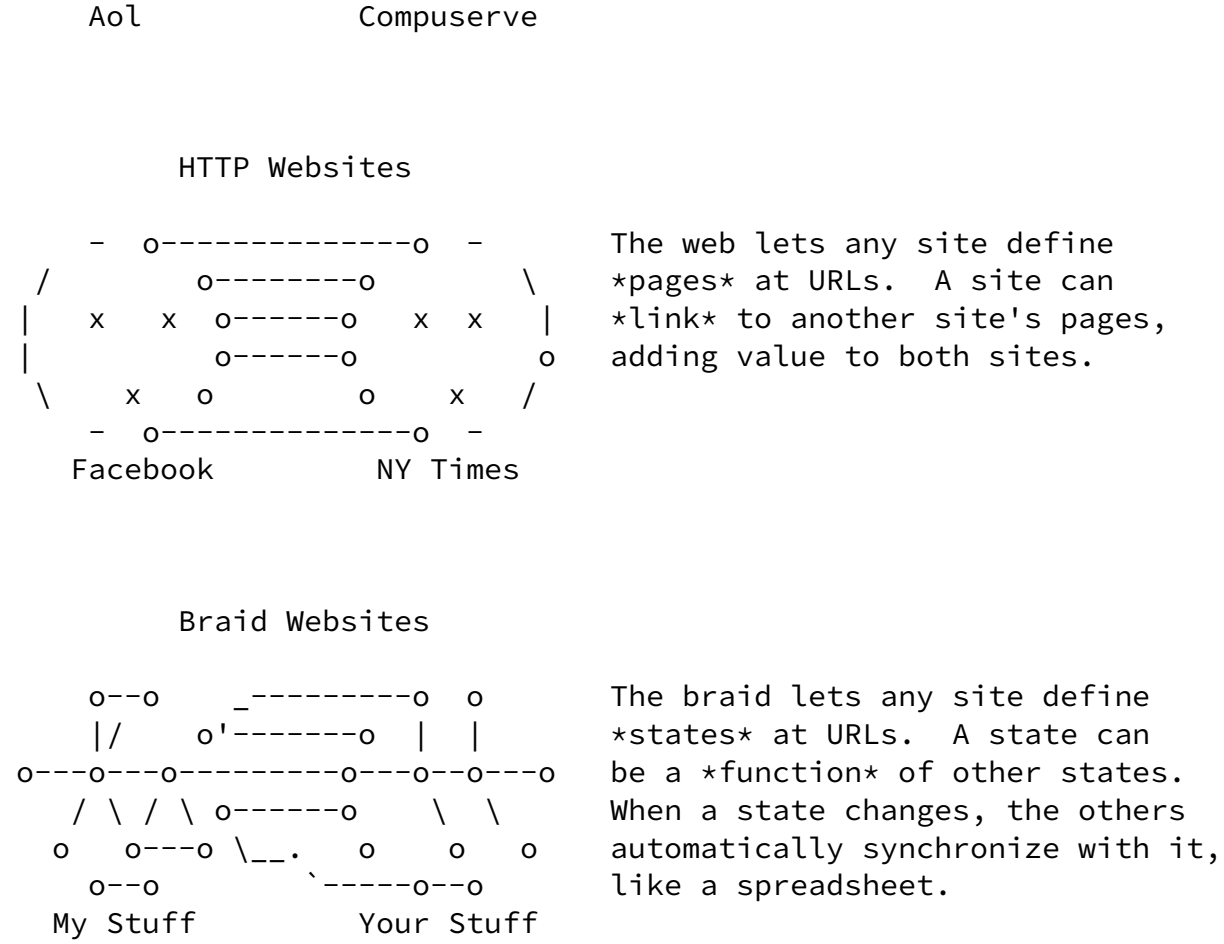
- Servers become optional. Many apps can function without a server, because peers can synchronize with one another directly over the protocol.

- In standardizing synchronization, we implicitly create a standard for state. This allows state to be shared between different sites without the need for an extra API.

- Standardizing the representation of *state* allows us to separate the representation of UI and state. Most HTTP websites *inject* the state that they receive into templates representing UI components. Braid sites can instead understand UI components as lenses through which to view state. This improves the semantics of UI rendering, allows state to be inspected by clients directly, and makes it easier to build multiple alternative UIs for a single site.

Standardized state and synchronization allows the topology of content on the web to be less centralized.





3. Deployment and Upgrade Plan

Braid makes fundamental changes to HTTP and REST, which creates an opportunity to unify a number of disparate features (SSE, E-tags, Cache-control, PATCH, JSON-Diff) within a simple integrated design.

Rather than shoe-horn these changes into the existing HTTP semantics, we propose a new simpler layer, with a backwards-compatible mapping to HTTP's existing semantics. This allows existing applications to interoperate, but new applications can opt-in to a simpler web API that provides more synchronization features.

We can deploy these semantics in two phases:

1. The first phase requires no changes to existing web browsers, making it easy for users to experiment with the protocol's semantics in existing websites.

   In this phase, browsers fetch the initial HTML page over the existing HTTP protocol, which includes Javascript code that initiates a WebSocket connection that runs the Braid protocol. This WebSocket version of the Braid protocol provides the full

synchronization functionality, but is less performant -- requiring an extra round-trip to initiate.

2. If and when the WebSocket protocol stabilizes and achieves real-world adoption, we can add the Braid semantics into HTTP itself, in a new layer via an HTTP Upgrade header [RFC 2616, section 14.42].

Both versions of the protocol can maintain backwards-compatibility with existing HTTP clients and servers. Any client accessing a Braid server via HTTP 1, 2, or 3 will be able to GET, PUT, and POST Braid state, but without full synchronization capabilities. Likewise, Braid clients can access state on HTTP servers, but will have to poll the server for updates.

4. Proposed Changes to HTTP

Braid is composed of a set of opt-in changes that any browser or server can implement.

First, whereas HTTP is explicitly client/server, Braid is capable of running peer-to-peer. To do this, it generalizes the explicit request/response pattern of HTTP into a set of common messages, opened over a persistent two-way connection.

Additionally, Braid specifies a new content-type for a resource: Linked JSON. This provides a standard format for dynamic state, akin to how HTML specifies a standard for the presented content of a webpage.

This section first describes Linked JSON, and then the changes to HTTP networking methods, and finally the versioning features that ensure a peer-to-peer network of edits converge to the same version.

4.1. Linked JSON

Whereas HTML defines a common format for the presented content of web pages, braid defines a common format for their internal data, or state: Linked JSON. This extends standard JSON with two additional datatypes:

- A *link*, which lets one piece of JSON, at one URL, to link to another piece of JSON at another URL

- A *binary blob*, which lets one encode a binary file (such as an image) as a value.

We encode links within JSON like:

```
{
  "foo": 3,
  "bar": 5,
  "something else": {"link": "braid://foo.com/something"}
}
```

Any object with a field named "link" is special, and interpreted as a link. To encode an actual field named link, you prefix it with an underscore, like:

```
{"_link": "this is not a link"}
```

To encode an underscore, you use two underscores, like "__", and so on.

Links allow programmers to combine data across multiple services, even on multiple websites, and to represent non-tree data as JSON, such as circular graphs and relational tables. Foreign keys can be specified as links to other queries. Cycles can be specified as a link back to the root of an object.

Binary data is encoded similarly, as:

```
{"binary": "<base64-encoded string>", "content-type": "<type>"}
```

The content-type is optional.

4.2. Generalized request/response

In HTTP, a client sends a *request* to the server, and that request is met with a *response*. By contrast, a Braid connection is two-way, so messages can be initiated by either party. Rather than giving a response to a message, a Braid server sends a separate message that acts as the response. It turns out that a GET response message has the same effect on a peer as a SET request message-- both set the state on the recipient.

HTTP	Braid	Meaning
Get Request	Get message	"I want this"
Get Response	Set message	"This is the current version"
Put Request	Set message	"This is the current version"
Put Response	Ack message	"I accept this version"

Subscriptions

In the Braid model, whenever a client requests some state, it also subscribes to new versions of that state. This requires only minor changes to the semantics of HTTP.

In the Braid protocol, a GET message not only returns the current value of state, but also *subscribes* to future updates. The subscription continues until the client sends a FORGET. Finally, Braid unifies the PUT, POST, and PATCH methods in to a single SET method, which is able to both create state and change state.

HTTP method	Braid method	What's new
Get	Get	Also subscribes to future updates
- n/a -	Forget	Ends a "Get" subscription
Put/Post/Patch	Set	Also updates all subscribers
Delete	Delete	Also updates all subscribers

The traditional distinction between PUT and POST is that PUT requests are idempotent, allowing them to be cached and retried, whereas POST is often not. However, the Braid protocol allows idempotence to be distinguished by re-using a version on Set -- if two messages set a state to the same version, they are idempotent, and equivalent to a PUT.

In their simplest forms, these messages are otherwise semantically identical to their corresponding HTTP methods. When robust synchronization is required, these messages will include optional *versioning* features.

4.4. Versioning

Even though there are many synchronizers, it is possible for them to communicate in a common language. Different synchronizers use different data structures internally, and have different network messages-- however, the *information* they send can all be represented with a common set of objects:

- VERSIONS define points in time, irrespective of space
- LOCATIONS define points of space, irrespective of time
- PATCHES replace regions of space, across spans of time

These three objects are enough to represent any type of change to a JSON data structure. We have verified this experimentally, by

implementing a translation algorithm that converts the network messages of ShareDB (an Operational Transform synchronizer) and Automerge (a CRDT synchronizer) into these objects, and back again, and verifying that the synchronizers still work in fuzz testing.

However, synchronizers also differ in how they resolve conflicting changes to the same region of state. We can generalize the behavior of these resolvers by defining *merge types*:

- MERGE TYPES define how edits to the same location resolve

If a synchronizer expresses state changes using versions, locations, and patches, and specifies its merge types, then it can synchronize with any other braid synchronizer implementing the same merge types, no matter their internal implementation.

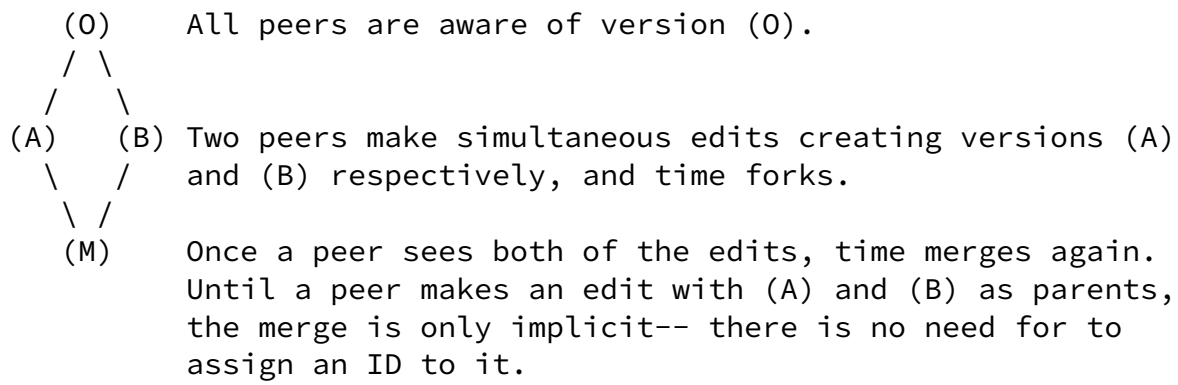
Finally, synchronizers also broadcast *acknowledgements* of the versions they have received, in order to tell their peers that they have moved forward in time, and will no longer refer to old history when sending patches. This allows their peers to prune their history logs, and free up unused memory:

- ACKNOWLEDGEMENTS of versions allow peers to prune historical memory

The rest of this section explains how these concepts work together.

4.4.1. VERSIONS

Time on a network is ambiguous as a result of latency. If multiple peers edit the same state at the same time, we cannot say that one happened before the other, and time forks. When they communicate their changes over the network, they merge their edits, and time merges.



All peers are aware of version (O).

Two peers make simultaneous edits creating versions (A) and (B) respectively, and time forks.

Once a peer sees both of the edits, time merges again. Until a peer makes an edit with (A) and (B) as parents, the merge is only implicit-- there is no need for to assign an ID to it.

Thus the shape of time is not a line but a Directed Acyclic Graph-- a DAG. Every change to state creates a new version. This version has *parents*: all recent (leaf) versions that the client had seen when it made its edit. The edit becomes the *child* of its parents. We can

say that one version came before another only if the first version is an *ancestor* of the second.

Every version is identified by a unique ID. There is no requirement on the format of the ID, only that it be a unique string.

When a peer makes an edit, it broadcasts the edit's version ID, its parents version IDs, and a patch from its parents state to its state.

4.4.2. Patches

When a peer changes some state, it encodes that change as a patch. A SET message includes 'patches' as an array of patches. All patches are *replace* operations, that replace one region of space with a new value. The region being replaced is specified as a start and end index of the previous state, which is computed by merging all the parents of the version using its merge type.

Insertions are implemented as replacing a zero-length region with a non-zero-length region, and deletes replace a non-zero length region with a zero-length one.

Here are some example patches targeting object 'obj'
(ie, {set: 'obj', patches: [...]})

```
  .foo[0].bar = null       # Replace obj.foo[0].bar with null

  [3:3] = "asdf"           # Insert the string 'asdf' at index 3 of
                             string obj.  Illegal if array.
  [3] = "a"                # Set char 3 of string obj to 'a'
                             or element 3 of array obj to 'a'
  [3] = "asdf"             # Illegal if string.
                             If array, set obj[3] to 'asdf'
  [3:4] = [1, 3, 5]        # Splice [1,3,5] into array obj, replacing
                             element 3.  Illegal if string.
  [4:4] = [{msg: "hi"}]    # Insert an object at the end of array obj

  [3:10] = ""              # Delete characters 3-10 in string obj

  = false                  # Set the entire object obj to false

  .foo[0].bar = undefined  # Delete obj.foo[0].bar
```

4.4.3. Merge Types

Different applications want to resolve conflicts in different ways. For instance, strings in a collaborative text editor will want to

merge clobbering edits by inserting everything typed, and deleting everything deleted, and breaking ordering ties arbitrarily; but if two debits to a bank account balance occur in parallel, we will want to merge the debits by adding the differences together.

A "merge type" specifies how any two edits made in parallel are merged together. If two synchronizers implement the same merge type for some state, they will converge to a consistent result after arbitrary merges. Merge types are specified by unique strings, such as "sync9-string" or "sharedb-rich-text".

```
LWW(vid)       # Last-write-wins, sorted by version ID
text(vid)      # Merges text edits like Google Docs
counter        # Merges additions and subtractions by summing
```

All peers synchronizing with a piece of state will need to implement the same merge types. We envision a handful of merge types will likely cover most situations on the web. Each field in a JSON object can merge using a different merge type. One way to specify merge types of JSON objects is with a schema:

```
{
    id: <string>             : LWW(vid)
    body: <string>           : text(vid)
    authors: <array>         : text(vid) [
        author_id: <string> : LWW(vid)
    ]
    likes: <int>             : counter
}
```

However, we have not yet implemented configurable merge types in our prototype, or settled on a way to communicate them in the protocol. Thus, we do not specify merge types in the network examples given later.

4.4.4. Acknowledgements

In order merge two versions, a synchronizer generally needs enough history to trace a path of time back from both versions through a common "fork point." Thus, in order to synchronize perfectly, peers need to store historical versions in time back to any fork point from which they expect another peer to send an edit.

To allow other peers to prune history, any peer thus needs to inform them that it no longer intends to base edits onto versions from the past. A general way to do this is for peers to agree to always make edits to the most recent versions they have. Then, a peer will be able to prune old history as long as it knows which versions of history all other peers have seen. Once all peers have seen a more

recent version, a peer can know that they will not base an edit on one of its ancestors.

In the Braid protocol, peers can communicate which versions they have seen using ACK messages. Each ACK specifies the URL and version of the state that the peer has seen. When a peer sends an ACK, it means "everyone who I have sent this version to has also acknowledged its receipt." Then, once the original sender has received all acknowledgements from all peers, it sends out a final "ack-complete" message, which communicates "everyone in the entire network has acknowledged receipt of this version." This is enough information for any peer to then know that no peer will be sending an edit based on a prior ancestor.

However, this spec does not yet handle the case where a peer goes on and offline in a peer-to-peer network. We are currently working on implementing a solution to this and will update this draft when the spec has been finished and tested.

5. Network Messages

We present some examples of the Braid protocol, from the perspective of a client communicating with a server.

Basic session with no versioning:

```
Send: {get: "text"}                                    [1]
Recv: {set: "text", val: "Hello"}                      [2]
Send: {set: "text", val: "Hello, World!"}              [3]
Send: {forget: "text"}                                 [4]

[1]: The client requests the most recent version of "text"
[2]: The server sends the client the value of "text"
[3]: The client changes the value to "Hello, World!"
[4]: The client unsubscribes from edits to "text"
```

Basic session with versioning:

```
Send: {get: "text"}                                    [1]
Recv: {set: "text", val: "Hello", version: "v1"}       [2]
Send: {set: "text", version: "v2",                     [3]
       patches: ['[5:5] = ", World!"'],
       parents: ["v1"]}
Send: {forget: "text"}                                 [4]

[1]: The client requests the most recent version of "text"
[2]: The server sends the client the value of "text"
[3]: The client adds ", World!" on to the end of "text"
```

```
  [4]: The client unsubscribes from edits to "text"
```

JSON locations and more complex versioning:

```
  Send: {get: "user/fred", parents: ['62347']}           [1]
  Recv: {set: "user/fred",                               [2]
         patches: ['.name[0] = "F"'],
         version: '2h38a',
         parents: ['62347']}
  Send: {ack: "user/fred", version: "2h38a"}             [3]
  Send: {set: "user/fred",                               [4]
         patches: [".name[4:4] = \" Wilson\""],
         version: "36x02",
         parents: ["2h38a"]}
  Recv: {ack: "user/fred", version: "36x02"}             [5]
  Send: {forget: "user/fred"}                            [6]

  [1]: The client requests the current version of "/current_user",
       as a patch based on version 62347, which it has in cache.
  [2]: The server responds with a SET containing a patch.
  [3]: The client acknowledges receipt of the new version.
       This enables the server to prune its history of old versions.
  [4]: The client updates the current user's name to "Fred Wilson".
  [5]: The server acknowledges receipt of the new version.
       This enables the client to prune its history.
  [6]: The client is done with current_user, and unsubscribes.
```

GET semantics:

```
  Send: {get: "val"}                                     [1a]
  Recv: {set: "val", val: ..., version: "vX"}            [1b]
  ...
  Send: {get: "val", version: "vX"}                      [2a]
  Recv: {set: "val", version: "vX", val: ...}            [2b]
  ...
  Send: {get: "val", parents: ["vPA", "vPB"]}            [3a]
  Send: {get: "val", parents: ["vPA", "vPB"],            [3b]
         version: "vX", patches: ...}
  ...
  Send: {get: "val", version: "vX",                      [4a]
         parents: ["vPA", "vPB"]}
  Send: {get: "val", version: "vX",                      [4b]
         parents: ["vPA", "vPB"], patches: ...}

  [1a]: The client requests the document "val".
  [1b]: The server sends the client the most recent version, giving
        the explicit value of the document as well as the ID of the
```

most recent version. The server also subscribes the client to new updates.

[2a]: The client requests version "vX" of document "val".
[1b]: The server sends the client version "vX", giving the explicit value of the document at version "vX" as well as the ID "vX". The server does NOT subscribe the client to new updates.

[3a]: The client requests the document "val" as a patch based on parent versions "vPA" and "vPB".
[3b]: The server sends the client the most recent version, giving its value as a patch against the implicit merge of the given parents "vPA" and "vPB", even if these are not the original parents of version "vX". The server also subscribes the client to new updates.

[4a]: The client requests version "vX" of document "val" as a patch based on parent versions "vPA" and "vPB".
[3b]: The server sends the client version "vX", giving its value as a patch against the implicit merge of the given parents "vPA" and "vPB", even if these are not the original parents of version "vX". The server does NOT subscribe the client to new updates.

6. Security Considerations

Although this protocol enables and encourages web programmers to make more internal available and shared, it has the same fundamental security model as HTTP. State at any URL can have access control controlling who can access it.

However, additional work will be needed to analyze the security concerns of specific uses of the protocol.

7. IANA Considerations

This document has no actions for IANA.

8. Copyright Notice

include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

9. Author's Address

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

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

Rafie Walker
Invisible College, Berkeley
2053 Berkeley Way
Berkeley, CA 94704

Email: slickytail.mc@gmail.com