The Immutable Web

A building block for a decentralized, resilient web.

Robert Kaye, July 2018
feedback@immutableweb.org

Disclaimer: While this paper borrows currently very popular topics involving blocks and chains, this paper will refuse to connect those two words, in a effort to keep a concerted distance to the complete nonsense that is evolving elsewhere.

Name disclaimer: So far no clear winners for a name have emerged. The current name is a very geeky and apt description of what we are building, but it probably isn't a good name for general consumption.

Introduction

The web ecosystem of today is not what Tim Berners-Lee had envisioned for his creation. A lot of user data is controlled by too few corporations that likely not have the best interest of their users at heart. Users are not in control of their own data and worse, if someone with power or money doesn't want you to keep some data, they can shut you down. Doing serious investigative journalism into government corruption is utmostly challenging today because the current state of tools require journalists to also be extremely computer savvy.

The dynamic web we love today also suffers from the problem that any data can be changed at any time. Very few resources such as the Internet Archive's Wayback Machine provide a glimpse into what really did happen in the past. Sadly, the Wayback Machine isn't enough to build a solid digital society where we can collectively protect sensitive data, remain in control of our own data or collect data on bad actors who wish not to be disturbed in their nefarious business.

The immutable web builds a very low level distributed, resilient data storage layer that combines with social features to distribute, secure and protect data. Ideally this system would prevent spammers and bad actors from diluting or taking down data. Ideally governments, corporations, the mafia and other bad actors should face a near impossible time eradicating data from the immutable web. And no one should be able to change the data without compromising the whole data stream.

The goals of this storage include:

1. Store and distribute data in a manner that makes it difficult to remove from the Internet.
2. Store data in streams that can be appended to over time.
3. Have the option to store the data anonymously.
4. Easily verify data integrity
5. Prevent spammers from diluting data streams
6. Prevent bad actors from inserting bogus data into streams
7. Allow the people who host this storage to recover their costs, or even run a business on the basis of hosting these streams.
8. Super easy deployment in forms of deployable containers or SD cards suited for Raspberry Pi computers connected to home routers.

The anti-goals of this storage include:

1. Prevent the bad guys from using these tools. Governments and banks may very well end up using these services, but we can also expect people like child pornographers and organized crime to also make use of them.
2. Prevent the creation of massive digital piracy dark networks.

Concepts

The basic concepts of this storage include Streams of blocks that are stored on Servers.

Streams

Streams will always be publicly readable and publicly accessible, but only the creator of a stream can write to the stream or otherwise the integrity of the steam is compromised. If the creator of a stream desires for the stream to be private/confidential then the creator should choose a form of encryption that is suited to their needs. Stream encryption and the key management of these encrypted streams are outside of the scope of this project and lie one layer above this data layer.

Data streams will be identified by a UUID, contain a creation timestamp and will contain metadata about the stream. The very basic elements of metadata will only serve to make the stream functional -- there are no requirements to describe the stream or to identify the owner of the stream.

Streams will consist of blocks, which will be described in the next section. Each block in the stream will contain a cryptographic hash of its contents and then be digitally signed by the stream creator. This will prevent bad actors from tampering with individual blocks.

Each block will also include the hash value of the previous block, to ensure that people who host the streams cannot insert unauthorized blocks of data into the stream. Anyone who mirrors

a stream should verify the integrity of the stream when accepting a new block into the stream. If a block cannot be verified via the cryptographic signature of the stream, the block should be rejected.

Each stream starts with block 0, the manifest block. The manifest block is a JSON encoded document that must contain an the identifying UUID of the stream and the public key that will be used to digitally sign each block and to verify the integrity of the stream.

All the remaining types of blocks in a stream either contain content that the stream owner wants to place in the stream or directive blocks. Directive blocks indicate a change in the stream; directives include:

1. Stream end: The stream is complete and ends here.
2. Stream fork: This stream will continue after this directive block, but the directive block will contain one or more UUID streams that are related, but entirely different streams. It is up to the consumer to decide how to act on this directive.
3. Stream merge: This stream is ending, but the consumer is suggested to continue consuming at the merge destination stream.
4. Stream redirect: This stream ends, but the consumer is redirected to another stream that will continue this stream.
5. Related to: This stream is related/relevant to another stream. This stream will continue.
6. Replay of: This stream is an exact replay of another stream and it may add more blocks than the original stream.

All directives that refer to other streams can optionally specify a cardinal block offset. No new blocks are allowed after stream end, stream merge or stream redirect directives. If blocks appear after these directives, the stream should be considered corrupt.

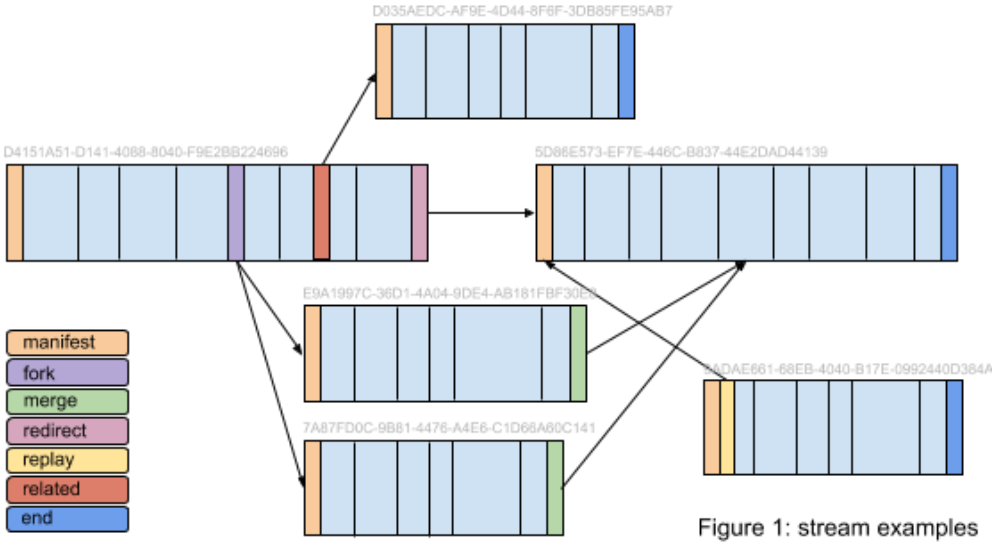

Figure 1: stream examples

Addressing streams

To refer to a stream we can construct the following URI:

> **iw://[ip_address[:port]/]uuid[/block offset]**
> iw://154.34.129.91:4589/6E725903-DC4F-417F-AB61-740EC97631E1/2

Ip_address/port are optional, but certainly required until a DHT is working. UUID is the only required part and the last optional part of the URI is the block offset.

blocks

Each block in the stream must contain:

1. A 64 bit integer, describing the size of the block.
2. The sha256 hash of the previous block, 0 if this is the first block.
3. Metadata: A optional JSON document about information relating to this block. It may specify a public key for the encryption used for this block, for instance.
4. Content, a series of bytes. For manifest and directive blocks the format for these blocks is defined below. For content blocks, the format is free form and left up to the user to decide.
5. A sha256 cryptographic hash of everything in the block up to this hash.
6. A digital signature of all the content in the block, including the hash above.

The concept of encrypted streams lives on the logical level above the blocktree. If a user wanted to encrypt the whole stream, the user could provide the public key in the manifest block. If the user wanted per block encryption, it would be up to the user to manage the contents of the block to provide data and the needed keys.

Manifest block

The manifest block must contain the UUID of the stream and public key that will be used to digitally sign each block in the stream and an identifier indicating which encryption software is used for signing the stream.

All of the other elements in the JSON manifest will be optional:

1. URL of for more information about the stream.
2. Contact info of the stream creator: web address, email, IM handles, twitter accounts.

3. Cryptocurrency wallet information associated with the stream. The presence of a wallet id does not indicate that the wallet is associated with a given stream. (e.g. a New York Times journalist may give the EFF donation wallet's address)
4. A user defined metadata section, where the creator may store whatever other data should accompany the stream. This could be a well defined piece of information, a JSON document or an undefined blob.

Directive blocks

Directive blocks contains only a JSON document as its content. This document requires the following keys:

type: one of: fork, merge, redirect, end, replay.
uri: an immutable web stream URI (for type merge, redirect)
uri-list: a list of immutable web stream URIs (for type fork)

A merge, redirect or end directive closes the stream. If any additional data appears at the end of the stream, it should be considered corrupt. A redirect points the reader of the stream to a new stream to follow. A fork indicates that another stream has been started forking off this one and a merge indicates that two more more streams are converging into this stream. A replay directive indicates that this stream is a replay, an exact copy of a stream using a different stream ID.

Servers

This system relies on a collection of servers that will host data streams. Servers host streams of data and to serve them to anyone who requests them. Servers will only accept new blocks for streams if they continue the integrity to the stream; this effectively allows anyone to attempt to write to a stream, but only new blocks that are properly signed by the stream owner will be accepted by the server.

Servers should be easy to deploy ranging from Docker (or similar) containers for easy deployment or SD Cards for Raspberry Pi's that individuals can connect to their home routers. Given this ad-hoc nature of servers, server lifetime cannot be relied upon, therefore many servers should be in existence to replicate streams. Having streams redundant across many servers in many geographic locations should make these streams resilient, both from data loss, bad actors and nations.

If someone creates a new stream, no one will know that this stream should be replicated. To address this, trusted organizations and entities such as the EFF or the Guardian should publicly list a set of UUIDs of streams that they believe are worthy of being replicated.

How these organizations determine this, is entirely left up to them. Good samaritans who wish to help can instruct their servers to replicate the streams that their trusted organizations wish to replicate. Anyone can host a set of stream and let the world know that they are hosting streams and they should be replicated. Whether or not anyone trusts a server really depends on the operator of the node operating in a trustworthy manner.

Furthermore, a server may expose a cryptocurrency wallet ID where random people can deposit money, optionally with stream IDs to sponsor, in order to help cover hosting costs. Some form of reliability of the host will need to be established, otherwise hosters can collect money and never host streams; this work is TBD. If someone is just starting out with a data stream that may have no established value and/or doesn't want to share the context of the stream with anyone, the stream creator may pay a someone to host the stream, until an organization can endorse the stream to achieve greater redundancy.

There is no concept of stream ownership or that a server has authority for a given stream. Ownership of a stream lies solely in who possesses the keys to sign new packets to be appended to the stream.

Streams over HTTP

Classic HTTP is perfectly suited for accessing streams. For a given stream at the /<UUID> location, the following queries are valid:

> **GET**: Fetch a stream, block or range of blocks.
> **PUT**: Create a new stream. This is the only operation that needs to authorization for access, otherwise spammers create many new meaningless streams.
> **POST**: Add a new block or blocks to an existing stream. Any block or blocks POSTed to a stream that isn't signed by the same digital signature as the stream will be rejected.
> **HEAD**: Get summary information about the stream.

Performing a GET on /streams , allows the caller to fetch a list of streams in JSON format that are being hosted on this server

Stream discovery & mirroring

Each server participates in a Distributed Hash Table (DHT) or P2P network (details still TBD) that serves as the equivalent distributed DNS for discovering streams and recent write activity to streams. Each server advertises its list of streams and the hash and timestamp of their latest blocks in the DHT and answers requests for steam discovery of other streams.

When a server discovers that one of its mirrored streams is out of date, it will fetch and verify any new blocks and append them to its local copies. Once the stream is updated, the server will update the latest block's hash and timestamp in the DHT.

A stream owner may write to any instance of the stream (as long as the owner has write permission which not all server may choose to provide). As long as the owner only ever writes to one stream instance, no mirroring conflicts will arise. Should the owner of the stream decide to write to a different instance of a stream, it is the owner's responsibility to ensure that previous writes have fully propagated before writing to a different stream instance. If the user fails to adhere to this rule and causes a collision, then the conflicting blocks will be rejected and the stream and all its replicas will be closed.

Example use cases

The following progressive use cases illustrate how the Immutable Web might be put to simple use:

1. Storing a static web page in an Immutable Web stream

To store a complete web page in an IW steam a program would need to collect all of the required files for the site (HTML, CSS, JS, image, etc) and store them into some archive format such as tar or zip. This zip file could be written to the stream in one block with subsequent versions of the same page stored in further blocks.

To read this web page, an IW aware stream reader in Javascript would be loaded and pointed to a stream ID. The javascript program could fetch the latest block, possibly decrypt, and unzip the contents of the file and then render the page in the browser.

The software to read the stream is fully decoupled from the data being stored -- the javascript based stream reader/page renderer could be used in many different contexts and could easily be stored in a CDN. Similar to a BitTorrent client, this client can be used to access "illegal content" but it itself is not illegal.

2. Storing a blog in an Immutable Web stream

One can extend #1 and store static versions of a blog. For each new blog post a new blog post page and new index files would be written — only the latest files in the stream would be considered to be part of the most recent state of the blog. All of the latest files together make up an accurate version of the latest state of a blog. Also, storing a machine readable copy of the blog data/content would allow the original blog to be restored from the Immutable Web stream.

3. Storing a blog with comments in an IW stream

This extends #2, but comments are also written to the stream. This allows dynamic pages to be stored as static snapshots in the Immutable Web. However, at this point we can clearly see how this would not be very efficient — parsing through a massive blog with tons of comments would be very very slow. But having a robust archive of your blog can be off massive value!

4. Using IW as a permanent archival master for dynamic sites

Instead of dynamically loading a whole blog for a casual reader, one could use an encrypted IW as a master archival copy of the dynamic web page/blog. The actual use of a blog would continue to happen using existing software (e.g. wordpress) and as users contribute to the site, more blocks are written to the IW stream. Should an active blog be taken down for whatever reason (e.g. government intervention, ISP account closure, etc) it can be revived on a new server by loading the blog from the IW.

The uses cases can be extended to further provide automatic blog re-creation in case a blog gets taken down and even having a sort of meta-DNS that always knows the current working location of a blog, so that users would always be directed to the latest and greatest stable instance of the blog, regardless of when and where someone attempts to take down the blog.

The blog software is clearly only a simple example -- many more complex web sites and data sets can be stored in the Immutable Web. Furthermore the code that runs the site (blog) can be made generic and to work in a data driven manner, so that an IW stream can provide the configuration, look and feel and content of a site.

Advanced example

To illustrate how this resilient storage system could work in real life, consider the following workflow for investigative journalism:

A journalist begins working on an investigation and will collect data from a number of sources. In order to keep the collected data safe and to distribute the data to other collaborators (consumers), the journalist starts a data stream. To digitally sign the stream of data, the journalist also creates a GPG key for signing the data blocks. The stream is started using this public GPG key on a server hosted by a trusted third party. The IW server will associate the UUID of the created stream with the provided GPG key to sign the data blocks. The journalist

may also submit a (different) GPG key to actually encrypt the documents before adding them to the stream.

The journalist then starts generating/collecting data points, documents and audio clips from a number of sources. A collection of data points, such as when a given person was seen in a particular location or documenting a money transfer for example, will need to be collected into a document. This document along with other documents such as sound clips and images will then be encrypted and written to the data stream, one document to a block. Each block is signed digitally.

The journalist then finds another journalist working on the same investigation and they decide to team up and to join forces in collecting data. The second journalist starts their own stream and provides that UUID to the first journalist. The first journalist then forks the first stream, providing the URI for the second stream. This indicates that there are now two independent, but related streams of data being created. A new stream may be added for each additional journalist involved in the case.

This process may continue for quite some time until a body of data has been built up that can be examined for patterns and other pieces of information. When this time arrives, the journalist may provide the stream ID to the person or persons who will analyze the data. If the journalist knew who would be consuming the documents, then the journalist could have encrypted the documents with the public keys of the recipients -- if not, the journalist will need to provide the private keys to the recipients so they may decrypt the documents.

At this point the journalists and the analyst may spend time analyzing the collected data and deriving conclusions from the data. These new conclusions could form a new stream of data and the journalists could then redirect their streams to the new stream of data or they could relate their streams to this new stream of data using a related to directive. The flow of streams may look like Figure 2:

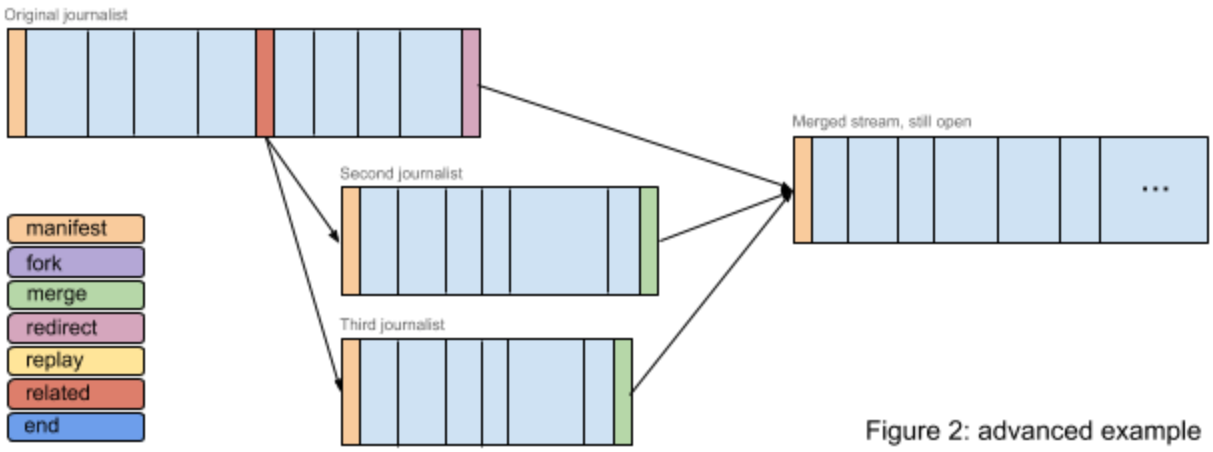

Figure 2: advanced example

The journalists may continue to write new blocks as new data arrives and analyzed data may continue to be written to the new stream. The journalists and analyzers can continue their work, creating, merging and forking streams as needed as their work evolves. Anyone who has the keys to decrypt their streams, if they are encrypted, can follow along and consume the data. To mitigate against data loss in the case of being discovered by the target, the journalists should provide the UUIDs of streams and the encryption keys to the streams to third parties. These third parties can be instructed to release the UUIDs/keys in case the journalist ceases to check in with the third party. The process of disclosing data becomes much easier since only UUIDs/keys are needed (and not the data itself) -- this small amount of data could be printed onto a postcard!

Ideally the streams will be replicated across many machines, across many jurisdictions, making it very difficult to shut down the streams of data. If a stream is replicated in 15 countries on 100 servers, it would take a serious and concerted international effort to remove one stream. And the publicity this effort receives would cause the creation of many more stream copies making the eradication of the streams effectively impossible.

Open questions & Concepts needing further development

1. Add the concept of one server following another, so that streams from one server automatically get created on following servers.
2. Define best practices for getting streams replicated. (blocks not too big, streams not too long)
3. How are streams that are no longer needed disposed of? Removing the stream ID from a list of streams to be replicated can help, but older streams could continue to linger.

References

1. https://www.wired.com/2017/04/tim-berners-lee-inventor-web-plots-radical-overhaul-creation/
2. https://www.mozilla.org/en-US/internet-health/decentralization/
3. https://ruben.verborgh.org/blog/2017/12/20/paradigm-shifts-for-the-decentralized-web/