Messiah: A decentralized peer-to-peer blockchain-based zero-trust information-sharing network

Jim John Johnson Jones Junior II

Liberta Veritás

Abstract

Freedom of speech is the definitive barrier to humanity's future and must be preserved at any cost.

This is a response to the centralized control of information distribution and content moderation from companies, governments, and individuals wishing to control how it flows around the human race to control its freedom of life.

The only weapon existing for individuals against the monopoly of the state is the entropy of the universe by using the information-sharing dilemma:

"Once the information has been shared, there is no way to get the system back to its previous state."

This paper proposes a way of sharing information without relying on single points of failure, by using existing technologies.

Keywords: blockchain, information sharing, cryptography, censorship

1. Introduction

Messiah is a decentralized information-sharing network that works on file distribution mechanisms, based on Bitcoin's blockchain concept (Nakamoto, 2008), anyone can download these files, authenticate them based on the public-key infrastructure[1] embedded in the file, also validate their contents within a level of trust defined by the user, by trusting available public keys and it's certificate chain.

If at least one copy of any file still exists online, anyone can read its contents and verify it with the information embedded in it, shared content

[1] Public-key Infrastructure

can refer to other previously shared content as well, so a user wishing to validate its content can authenticate it's integrity and authority by going all the way up in the public certificate chain using the digital signature present in each data entry of the shared file(s).

A public key infra-structure (Housley et al., 2002) defined in the file is used to maintain consistency and deal with everything that is needed to authenticate its contents using the digital signatures in each entry. No dependency on external information when authenticating a file shall be preferred as much as possible: a zero-trust environment without external dependencies.

Once the content file is published and shared online, due to the decentralized nature of the network, there is no mechanism to go back and remove it, there is no central control of what can or cannot be shared. On the other hand, there are different levels of trust in the content based on who published it and how many users referred to its content and shared new files about it. It shall be validated as trustable content if other nodes of the network hold its content in high availability within the peer network or any source of files.

Each content can be signed by other parties as trusted, along with metadata that may help other users sort through the files and find what they might be looking for, the metadata and referred content are also shared as new files within the public network as long as the digital signature mechanism is intact and can be independently checked.

Also, any content can refer to other content in the network by using a pre-defined hashing mechanism, a file carrying a referral from other content should, in principle, carry the original content as well so a user accessing it can read all the content without relying on downloading more files as they might not be available all times. If a file needs to be split, it is important to refer to the previously known file and provide any available known sources to help other users find its original content.

The distribution of those files is not dependent on a single transmission protocol, they can be shared through a standard web server, libp2p[2], bittorrent[3] magnetic link, or any protocol that is available at the time the content is being shared online, the author of the file can list the public repository link it was originally posted online if he wants as well, within the file itself.

Referrals to other files should preferably be linked to decentralized dis-

[2] libp2p
[3] BitTorrent

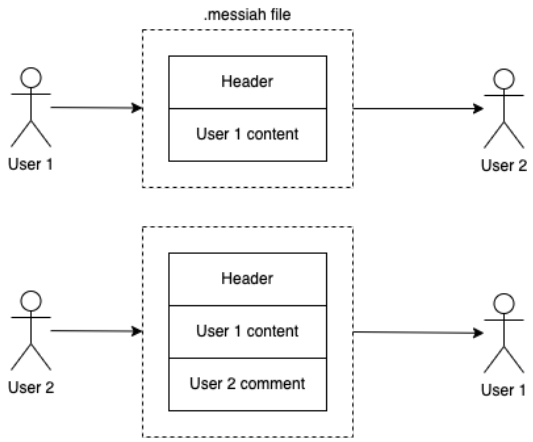

Figure 1: Basic file format diagram

tribution networks such as libp2p, bittorrent magnetic linking, BTFS, IPFS, and any future decentralized storage protocols to avoid censorship sanctions from other parties.

2. File format

The file format is JSON based (Bray, 2014), and each line of the file is a stand-alone JSON that is both human-readable content and easy for any software to parse, validate, and authenticate content, conforming to the JSON Lines format[4].

Binary content is encoded in BASE64[5] format and embedded in the entry along with metadata to identify what kind of data is encoded, following the standard media content type naming[6].

The basic structure of a Messiah file is this:
[header] *line_break*
[entry_1] *line_break*
[entry_2] *line_break*
[entry_n] *line_break*
[index_list] *line_break*

[4] jsonlines.org
[5] RFC1521
[6] IANA media-types

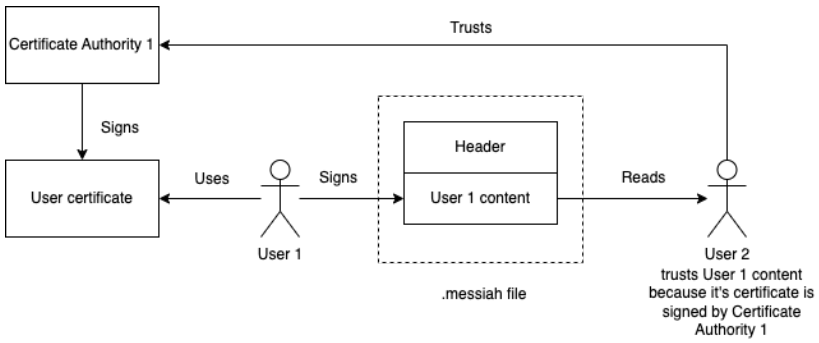

Figure 2: PKI diagram

Each file ***must*** begin with a header, in this header, there is basic information about how to validate the file contents, a certificate chain is usually the basic mechanism, along with a JSON schema [7] version that should be used as the detailed specification of all the entries in that file.

New entries can be added by any user in the network, as long as the previous entries are kept with the same payload and can be validated independently, the user wishing to add a new entry will attach an entry at the end of the entry list and update the index_list to add the final hash of that entry to easy the step of referring to individual entries in other files, see 1.

3. File naming

As Messiahs files are based on a zero-trust environment, it is necessary to check if the file contents have been completely transferred before any checks on its contents, for this to happen as easily as possible, the file name is used to provide the hash of it's final content, the hashing algorithm is SHA-256[8], and the format is:

`<hash-256>.messiah`

As an example, an empty JSON file ({}), should be named:
ca3d163bab055381827226140568f3bef7eaac187cebd76878e0b63e9e442356.messiah

4. Signature

To publish a new entry in a file, be it a new file or an entry in an existing file previously validated, the publisher of the content must sign the entry to help other parties validate that the contents are not tainted in any way, to do that, the user needs to generate an RSA-PSS[9] key-pair and sign the content with his private key, including the signature in the *signature* field of the new entry.

The public key must also be inserted in the same entry so that the entry can be self-validated independently, this key must be included in *PEM BASE64 encoded format*[10], or in an X.509[11] compliant certificate with additional information about the publisher.

This certificate can be signed by a trusted party and will provide additional trust to the publisher of the information. If the certificate is signed by a trusted party, the publisher must add the complete certificate chain up to the self-signed certificate authority public key in the header entry, appended to the existing root certificate chain as the X.509 specification (Housley et al., 2002) requires to validate certificate chains and any additional intermediate certificates must be included as well.

The validation of this signature field by other parties can provide in a single process some **irrefutable** conclusions:

- That the contents of the entry have not been modified in the transmission, because the signature of the file matches the contents and has been signed with the public key published in that entry;
- That the public key that was used is valid and has a valid private key used to sign it;
- If the public key has been published within an X.509 certificate, the publisher certificate can be verified with the certificate chain up to the certificate authority, validating that this authority has indeed signed it;
- If the user trusts the certificate authority that signed the certificate, the publisher of the information can be trusted as well: ***chain of trust***.

[7] json-schema.org
[8] SHA-2
[9] RFC4055
[10] RFC7468
[11] X.509

(See Figure 2)

For the signature to be checked in the entry payload, the user must first remove the signature from the JSON, leaving it empty:

```
{ ...,
  signature: ""
}
```

Then, with the signature, it must look for either one of the two fields: *user_public_key* or *user_public_certificate*, and compute to check if the signature matches the content and the public key. If it doesn't, that entry should be discarded as tainted content and not republished in any way, as the next user will do the same if it encounters an invalid entry and can ignore anything beyond that Messiah file.

5. Entry linking

To interact with content already shared by other parties, a user must be able to refer to previously shared content to provide other users the ability to look for this missing piece of content and understand the thread of interactions since its inception.

To refer to other contents, the user must use an SHA-256 hash of the full entry payload, most of the time this will be available at the index_list entry at the bottom of the transmitted file it is just a case of checking if it is indeed valid by calculating the hash of the entry and refer to it on the new entry to be appended to a new messiah file.

On the recipient side, once a user finds a referral to other content, the first thing he needs to do is look in the index_list of this file to see if it is already present on the same file, if it is there, it will be the entry in the same position index of the array in the index_list, all it needs to do is get the entry and compute the hash, if it is a match, then it is indeed the content that he is looking for.

If the content is not there, the user should start looking for this content in the network for this specific entry hash.

Although hash collisions are possible, with the SHA-256 hashing algorithm it is highly improbable that two entries will ever have the same hash in available Messiah files active in the network. Over time, as files get older and the data becomes unused, old hashes should not be a problem also because clients will stop sharing old files in the network.

6. Entry format

Every entry must have this format for other parties to understand and check it correctly.

When the publisher has **no signed certificate** yet:

```
{
  "timestamp": "...",
  "user_public_key": "...",
  ...
  "signature": "..."
}
```

When the publisher already has a **signed certificate**:

```
{
  "timestamp": "...",
  "user_public_certificate": "...",
  ...
  "signature": "..."
}
```

The *timestamp* must be in ISO-8601[12] format with date, hour, second, and **without** timezone information like this:

2023-04-01T16:32:57.714Z

7. Control Entries

There are entries in Messiah files that have other purposes than sharing user content, those entries are used to build trust between publishers of the network, similar to how followers are the proof of stake in social networks.

7.1. ***User Certificate Sign Request***

Once a user wants to publish a Messiah files, the first entry after the header, as long as his certificate is not signed yet, should be a BASE64 encoded, X-509 certificate sign request (CSR). This will provide the possibility

[12] ISO-8601

for other parties to sign it's certificate and share it's trust level across the network with the new user.

At the time the user finds a signed certificate that he has the corresponding private key, he should start using this certificate as his certificate entry in any newly posted content, as this will, in principle, greatly improve his trustability in the network and it's content engagement:

```
{
  "timestamp": "...",
  "user_public_key": "...",
  "user_certificate_sign_request": "...",
  "signature": "..."
}
```

7.2. ***User Certificate Signed***

Once a certificate signer (authority) finds a certificate sign request it wants to share it's reputation with, it must return the signed certificate to the network, the step to do that is to publish a new entry in an existing file or a brand new file with:

- The full certificate authority chain, also in BASE64, the user must adopt to verify the certificate authority trust chain, to be used in header entries;
- The signed user certificate in PEM BASE64 encoded that the user should start using from now on;

```
{
  "timestamp": "...",
  "certificate_authority_chain": "...",
  "user_signed_certificate": "...",
  "user_public_certificate": "...",
  "signature": "..."
}
```

The certificate authority also needs to add it's self-signed certificate in the *user_public_certificate* to ease the checking process by other parties, if it is not there, the user can use the certificate_authority_chain as the source of the public key to verify the signature against it as well.

7.3. ***User Certificate Revoked***

Sometimes a Certificate Authority can find a certificate he signed, misbehaving in the network and might compromise it's reputation, as a countermeasure, it can revoke the certificate, once this certificate is published as revoked, every party that finds a revoked certificate must store the certificate public key fingerprint in it's internal database to avoid giving voice to bad actors in the network:

```
{
  "timestamp": "...",
  "user_revoked_certificate": "...",
  "certificate_authority_chain": "...",
  "user_public_certificate": "...",
  "signature": "..."
}
```

In this entry, the certificate authority must also publish it's entire certificate chain to prove it has the power to revoke this previously signed certificate.

7.4. ***Data Source***

As the entire network is completely decentralized, users might want to share sources of information that might hold the file being shared or even additional files that might be needed in the future.

The entire system is built on top of existing protocols, so any valid data source can be shared online, like:

- HTTP / HTTPS addresses
- SFTP / FTP addresses
- Links to standard BitTorrent files
- Magnet BitTorrent URLs

Additional addresses can be added as long as they are supported by other clients, even specific purpose protocols like TOR[13] and IPFS[14] can be used to hold data sources.

The format of a data source entry is pretty simple:

[13] TOR
[14] IPFS

```
{
  "timestamp": "...",
  "data_source": "...",
  "user_public_certificate": "...",
  "signature": "..."
}
```

7.5. ***Reference***

Sometimes is not possible to add more entries to existing files, be it because the file is already too large, or any other reason, for this purpose the publisher has the option to include a reference to an existing published Messiah file that should be used as the source of content to new entries in the current file being built.

The purpose of this reference entry is to show what files the peer reading this file should look at next to fulfill the sequence of contents posted online about a topic, the *reference* is an SHA-256 hash of the complete referenced file.

There might also be an optional *data_source* entry that can be used to help other peers easily find those files, this field has the same specification as the **Data Source** entry.

```
{
  "timestamp": "...",
  "reference": "...",
  "data_source": "...",
  "user_public_certificate": "...",
  "signature": "..."
}
```

7.6. ***Alternate Reference***

Sometimes a publisher wants to store data in the network that is just a backup if something goes missing online, for example, he is talking about a website that might go down somewhere in the future. It doesn't make sense for him to include it in the current Messiah file because the website is still accessible, but it is a good practice to build another file that has the website backed up and refer to it as an alternate reference.

To do that, he registers in the current file that if that resource becomes unavailable, this is the backup file the reader should look for in the future.

- The *original_content* field must contain the reference used in the files that might go missing, for example, the website URL.
- The *alternate reference* field must contain the SHA-256 hash of the backup file.

The format is:

```
{
  "timestamp": "...",
  "original_content": "...",
  "alternate_reference": "...",
  "user_public_certificate": "...",
  "signature": "..."
}
```

8. Content Entries

Content can be anything that the publisher wants to share, it will be embedded in the entries and identified by the standard content type naming convention. If it is binary content, like an image or video, it needs to be BASE64 encoded as well.

Every content need to have at least one item in the array content with:

- *content*: text or base64 encoded
- *content_type*: IANA content type naming
- *content_size*: size in bytes of the content

Multiple content can be shared in the same entry, simply adding to the content array.

8.1. ***Public Content***

Public content is by definition readable by any recipient in the network. In addition to the 3 common content entries, it may also have a title entry to provide better context on what is being published. Also, the *title* enables the client interface to organize content in a thread-like hierarchy.

```
{
   "timestamp": "...",
   "title": "...",
```

```
   "content": [{ content: "...",
                 content_type: "...",
                 content_size: ...
               },
               { content: "...",
                 content_type: "...",
                 content_size: ...
               }],
   "user_public_certificate": "...",
   "signature": "..."
}
```

8.2. ***Private Content***

Private content in a blockchain-based network can only be achieved by encryption, a publisher wanting to share something that only some parties can understand will have to add a modified encoded content entry.

First, he needs to tell whoever is the recipient of the message that it is addressed to him, it should be inserted in the *destinatary* field, and the text identifier can be anything it wants as long as the recipient can recognize it as addressed to him. Also, this can be left out if there is not really a recipient for that message.

The rest of the content fields are prefixed with *encoded_* to help the network understand this entry is not plainly readable as every other content.

```
{
   "timestamp": "...",
   "destinatary": "...",
   "encoded_title": "...",
   "encoded_content": [{ content: "...",
                         content_type: "...",
                         content_size: ...
                       },
                       { content: "...",
                       content_type: "...",
                       content_size: ...
                       }],
   "user_public_certificate": "...",
   "signature": "..."
}
```

8.3. ***Private Content Key***

A content publisher might want to publish sensitive content and after the content is shared across the network, it wants to make it publicly readable by everyone. An example use case would be a whistleblower wanting to publish some sensitive content from a risky location and after that, he wants to make it readable as soon as he in a safe condition.

To provide the keys for the content to be decrypted, it needs to add this entry to a public Messiah file, preferably at the end of the previous entry with the encoded content, off course in some conditions it might be impossible.

- *encoded_content_origin* is the SHA-256 hash of the encoded entry.
- *encoded_content_encryption_algorithm* is the encryption algorithm used to encode it
- *encoded_content_decryption_key* is the key to decrypting the content using the encryption algorithm.

```
{
   "timestamp": "...",
   "encoded_content_origin": "...",
   "encoded_content_encryption_algorithm": "...",
   "encoded_content_decryption_key": "...",
   "user_public_certificate": "...",
   "signature": "..."
}
```

9. Poll

Let's say the population of a country wants to make a public poll without relying on the government polls due to possible corruption in the voting process. If this country has an open public key infra-structure in place[15] that issues certificates for citizens using biometric[16] validation, all they need to do is include the full certificate-chain[17] of this public infra-structure in the header entry of a messiah file, and create a poll like this:

[15] ICP-Brasil
[16] ICP-Brasil Biometrics
[17] ICP-Brasil Certificate Chain

9.1. ***Poll creation***

The user wishing to launch the poll, creates the poll entry with:

- *poll_name* name of the poll.
- *poll_deadline* the end of the election, as a reference.
- *option_xx* each option of the poll.

```
{
   "timestamp": "...",
   "poll_name": "...",
   "poll_deadline": "...",
   "option_1": "description of the candidate 1",
   "option_2": "description of the candidate 2",
   "option_3": "description of the candidate 3",
   "user_public_certificate": "...",
   "signature": "..."
}
```

9.2. ***Vote***

When a person wants to publish its vote, all it needs to do is add a entry in the file like this:

- *poll* SHA-256 hash of the entry of the poll.
- *vote* the selected vote, one of **option_xx**.

```
{
   "timestamp": "...",
   "poll": "...",
   "vote": "option_1",
   "user_public_certificate": "...",
   "signature": "..."
}
```

9.3. ***Poll finished***

The same user that published the original creation of the poll, can and has the authority to sum all the valid polls before the deadline and publish a entry like this, at the end of the file, including all the votes before this entry to be independently validated by any party:

- *poll* SHA-256 hash of the entry of the poll.
- *result_option_x* the sum of all the votes computed in option_xx.

```
{
   "timestamp": "...",
   "poll": "...",
   "result_option_1": total_amount_option_1,
   "result_option_2": total_amount_option_2,
   "result_option_3": total_amount_option_3,
   "user_public_certificate": "...",
   "signature": "..."
}
```

10. Interactions

Information-sharing networks are based on user interaction over existing content, the following entries describe how the user can publish their interactions as well.

10.1. ***Republish***

Republishing content is the basic interaction a user can do with existing data, in fact, what he is doing is sharing his trust with existing publishers, the same behavior can be achieved with the digital signature structure in Messiah's files:

Publisher A shares interesting content, and his public key might be trustable or not in the network depending on who signed his certificate, however, Publisher B might want to give it some relevance, to do that, he simply republishes the content with his public key, that might be signed by a more popular certificate authority and give the content more interest on the network just by doing that.

To achieve that, he simply shares the original content wrapped in a republish entry: - *origin* is the original SHA-256 hash entry of that content. - *original_content* is the complete original content entry as it was first published, with all the data along with it to be independently verifiable as well.

```
{
   "timestamp": "...",
   "origin": "...",
```

```
   "original_content": { ... },
   "user_public_certificate": "...",
   "signature": "..."
}
```

10.2. ***Reaction***

Reactions work similarly, but besides just re-sharing existing content, the publisher wants to express himself on top of the content, there are two ways of doing this in the protocol.

The first is using the *reaction* entry as a simple text reaction, with keywords:

- liked
- disliked
- loved
- hated
- ...

The entry is as simple as this:

```
{
   "timestamp": "...",
   "origin": "...",
   "reaction": "like",
   "signature": "..."
}
```

The second method is to add a small icon to be used as the reaction in the customer client, the icon must be of any image-type format and with a pre-defined max squared width and height to fit in the user interface without too much image resizing as it will inevitably make the picture distorted.

In cases where the user embeds a custom reaction, the *reaction* field of the JSON is a content object with the 3 minimum required entries to correctly display the data:

- content_type
- content_length
- content

This is what it will look like:

```
{
   "timestamp": "...",
   "origin": "...,
   "reaction": {
    "content_type": "...",
    "content_length: ...,
    "content": "..."
   },
   "signature": "..."
}
```

10.3. ***Comment***

The next interaction is adding comments on top of existing content, this works as a way of putting away more information on top of existing data, the way to do it is to refer to the original content, preferably attached to the same Messiah file, and then add the *comment* entry, the comment itself can also be of any type, text, image, anything the user thinks is the better way to express his comment.

```
{
   "timestamp": "...",
   "origin": "...",
   "comment": {
       "content_type": "...",
       "content_size": ...,
       "content": "..."
    },
   "signature": "..."
}
```

10.4. ***Tag***

The next interesting thing to do with existing content helps other users find what it is about by using tags, the tags are a list of keywords that can be used to help in searching by other peers on the network.

The tagging process is made by adding an array of text-based content prefixed with the # sign, as follows:

```
{
  "timestamp": "...",
  "origin": "...",
  "tags": [ "#...", "#...", "#..." ],
  "signature": "..."
}
```

11. Nested entries

To reduce the number of entries a user need to insert in a file to publish content, he can choose to nest content from multiple entries in a single one, of course, some entries don't make any sense to be nested like content and a republish, but in principle, there are no restriction of entries that should be nested together as long as the data is the same.

There is also a benefit in computing power to validate the entry as this process must be done on each one.

```
{
  "timestamp": "...",
  "content_type": "...",
  "content_size": ...,
  "content": "...",
  "reaction": "...",
  "tags": [ "#...", "#..." ],
  "user_public_certificate": "...",
  "signature": "..."
}
```

12. Monetization

Due to de decentralized nature of the Messiah network, the ideal mechanism to monetize content is to publish, along with any entry, information about a financial transaction that can be done with the publisher of the information.

There are interesting use cases that can be implemented:

- A publisher asks for donations over published content;

- A publisher requests payment to send the decryption key of private content to be sent after the payment. Here the decryption entry can be sent readable by everyone, or encrypted to a single user, usually the payor of the transaction.

In order to do that, the publisher of the content must add to the entry the *wallet* address to where the money must be sent to, along with the *cryptocurrency* symbol[18] and, if this is a payment and not a donation, the desired *amount* to be paid:

```
{
  "timestamp": "...",
  "content_type": "...",
  "content_size": ...,
  "content": "...",
  "payment": {
    "wallet": "...",
    "cryptocurrency": "...",
    "amount": ...
  }
}
```

It is important to note that, as this is not a smart contract, there is no guarantee that the user will receive whatever he is paying for.

13. Distribution Channels

The distribution of Messiah files should be based on existing protocols, leveraging the large availability of working file-sharing mechanisms and networks.

Publishers are encouraged to focus on distributing files over peer-to peer-networks such as Libp2p[19], BitTorrent[20], Inter Planetary File System[21], Freenet[22] due to the uncensorable nature of those protocols.

[18] Coinmarket Symbol list
[19] libp2p
[20] BitTorrent
[21] IPFS
[22] Freenet

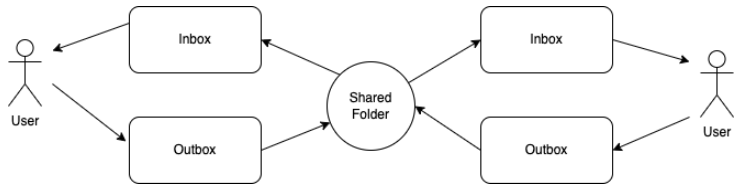

Figure 3: Filesystem distribution channel diagram

But in reality, there are two types of channels that can be used by publishers, centralized and decentralized.

On the **centralized** model:

- Filesystem
- FTP, FTPS and SFTP
- HTTP and HTTPS

On the **decentralized** model:

- Libp2p
- BitTorrent & WebTorrent
- IPFS
- Freenet

Also, for censorship mitigation, clients can connect over the TOR[23] network to obtain data source URLs to look for additional files on the open network as a dial-home mechanism.

13.1. ***Filesystem***

The simplest distribution channel can be a shared folder between a group of users, in order to do that they need to:

- Synchronize continuosly a shared folder between each user, independent of the protocol, it can be a SMB[24], NFS[25], Google Drive[26], Dropbox[27], or any other service;

[23] TOR
[24] Server Message Block
[25] Network File System
[26] Google Drive
[27] Dropbox

- Create a folder that will be used to store the files that need to be sent te shared folder, the **outbox**;
- Create a folder that will be used to store the files that the user has received, the **inbox**;
- Continuosly monitor the shared folder for new files, using a filesystem event mechanism like inotify[28], FSEvents[29] or similar and copy files that are not present in the outbox folder (files not built by this user) to the inbox folder;
- Continuosly monitor the outbox folder for new files and copy them to the shared folder.

Once this process is estabilished (Figure 3), the distribution channel will work without the need for a specific purpose server.

Obviously this isn't scalable and pose security risks to users that don't trust each other, but the basic principle can be extended to other file sharing protocols and services, denoting the adaptability of the network over existing data sharing protocols, mimicking how the distribution of printed papers worked in the real world over the centuries.

13.2. ***FTP/FTPS/SFTP servers***

Using FTP[30], FTPS[31] and SFTP[32] servers are interesting alternatives for data storage because they support:

- upload and download of files of any type;
- anonymous authentication (FTP/FTPS) and public key authentication (SFTP);
- directory file structure.

To implement indexation in the directory file structure (Figure 4), clients can organize the files based on dates or even common name(CN) entry of certificates that are popular in the network.

[28] inotify - monitoring filesystem events
[29] File System Events API
[30] File Transfer Protocol
[31] Securing FTP with TLS
[32] SFTP Protocol

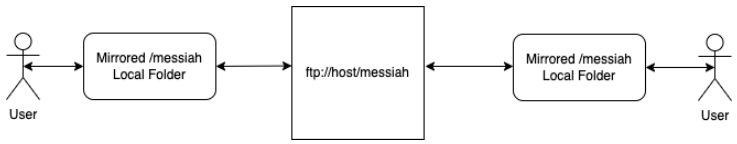

Figure 4: FTP distribution channel diagram

Multiple servers can be referred as *Data Sources* entries inside Messiah files themselves to improve data availability beyond libp2p peers or BitTorrent Magnet URLs.

Besides being centralized distribution channels, public FTP/FTPS servers can be hosted by anyone as long as files can be uploaded and downloaded freely, in fact, an open, anonymous server can host files to the entire network.

It even doesn't matter if the server stops working in the future, as long as it distributed files to other readers, it already helped the network spread new data to other peers in a specific point in time, as new entries can always be appended to existing Messiah's files, new FTP file repositories are easy to be shared among the network anytime.

13.3. ***HTTP/HTTPs servers***

Web servers are *de facto* the standard for web2 services, a implementation needs at least 3 endpoints to correctly implement distribution of Messiah's files:

A webservice needs to implemented with 3 endpoints:

- List files - example: GET /files
- Download a specific file - example: GET /file/filename.messiah
- Upload a file - example: POST /files/filename.messiah

Once the server receives a new file, his first action should be to validate the contents based on the public keys listed in the file itself, only after that the file should be listed to other users.

13.4. ***Libp2p***

The Libp2p[33] is a spin-off of the IPFS[34] project, the developers of the platform realized their peer to peer stack could be used in any application

[33] libp2p
[34] IPFS

and separated the implementation of the data exchange stack to the file system stack.

The library supports many protocols and peer discovery methods, two of them are quite interesting as a distribution mechanism:

- Gossipsub[35], a channel oriented broadcast[36] messaging system that can be used to announce new files available in the Messiah network, the peer subscribes to a named channel and receives any announcement sent to that channel by any other peer on the network.
- Dial[37] mechanism is a direct peer to peer connection that can be used to transfer files between them over the choosen transport layer (TCP, Websocket, WebRTC) and the choosen encryption protocol. The process of determining the right end-point information is abstracted by the libp2p implementation and can even use a routing peer when two peers trying to exchange data don't have means of communicating between themselves (when both are under a firewall for example).

See figure 5 example diagram below.

The peer discovery mechanism can use different strategies like a initial seed of known nodes on the network, random walking valid ip addresses, mdns for local networks, etc. As a last resource the application can fallback to connect to an TOR endpoint and ask for a list of available peers to use as seed if none of the previous strategies resulted in discovering remote peers.

13.5. ***BitTorrent & WebTorrent***

BitTorrent[38] is the most successful peer-to-peer file-sharing protocol[39] ever built, designed to optimize transfer between peers without consuming too much bandwidth from a central server, today it is used in almost every content distribution network on large scale.

Originally built to use trackers servers as repositories of peers' information (IPs and ports) that had available pieces of files, currently, the DHT[40]

[35] Gossipsub
[36] Pubsub
[37] Peer Dialing
[38] Original Specification
[39] BitTorrent Specification Improved
[40] DHT Extension

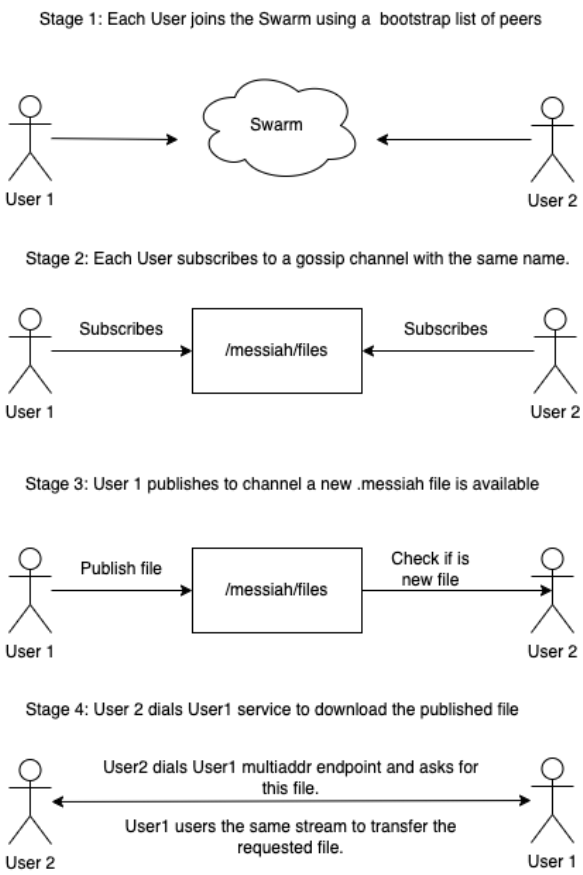

Figure 5: libp2p diagram

extension to the protocol enables peers to behave like trackers and share node lists among themselves using Kademlia (Peter Maymounkov, 2002).

In the previous versions, it was necessary to share .torrent files with the info-hash list of each piece of a file. The Magnet[41] URLs extension made it possible to share files with specially built URLs that can be easily shared across peers.

Unfortunately, a client running the BitTorrent protocol will still need to find a list of available Magnet URLs online to know what files to ask for from other nodes on the available torrent network, be it a legacy tracker or a DHT node[42].

Possible solutions:

- An extension for the torrent protocol to enable nodes to show lists of available info-hashes that are currently active in the network.
- Use of libp2p Gossipsub[43] protocol to broadcast files using magnet URLs.
- A database of HTTP/HTTPS/FTP/SFTP servers that can provide lists of magnet URLs to the client.
- A TOR endpoint might need to be used if none of the above options work.

WebTorrent is another possible solution to enable web clients to download files without a central server, A Messiah web client can support Webtorrent[44] protocol to share files between web browsers using only WebRTC.

This enables any website that hosts a web-based client to start sharing the Messiah's files stored in the client's computers, effectively using the publisher's computers for storage and distribution and the web server hosted on any URL as an index of available info-hashes to be shared among different customers.

Even multiple instances of the same web based client, running on different web servers will eventually share nodes and magnet URLs among them because of the decentralized nature of the BitTorrent protocol.

[41] Magnet URLs
[42] Storing arbitrary data in the DHT
[43] Gossipsub
[44] WebTorrent

14. Conclusion

Trust the control of what can or can't be shared online to private tech companies was an inevitable path until now, however with the use of cryptography, blockchain, peer-to-peer data transfer, and near unlimited data storage and transfer, there is a real possibility to turn the tide from an oppressive society towards his citizens based in information control, to a new era of true freedom of speech.

This document is a initial step in that direction.

I will release a complete working opensource prototype sometime in the future, hoping others will join this battle against censorship.

If you wish to be notified when this happens, go to my TOR website at: http://n5zefucqctip5ujzcokmvdavdddsnvfk74thdwfhqgexkiwrw3xp2rad.onion and sign-up.

15. Proof of Authority

To check if this paper is indeed written by Jimjo, below is my public key. There is a PDF signature on the first page of this paper that can be verified with this key.

```
-----BEGIN PUBLIC KEY-----
MIIEIjANBgkqhkiG9w0BAQEFAAOCBA8AMIIECgKCBAEAzJ0Rk5yvowMn4471dFpR
69R5k+/8pIA+J/VEArG7zFvwJ07o49acNHiwjuCCT0YWu6PL4fVSRE5/iV3jo5Jk
hnrBlTpL9hMcnorPOePatpWK72Xfr+iu6mZDwcGm2m44lMa/W9m4XEHJ/p1D/e1a
HNROasCEF9u64PHyiU89vfqrc6PDyZWQ+NRbvVwKs9W/ofBYtLqw2HwKk6tJRWKZ
cbd4O+YCGBAy8ah3EkpjUBsGEf1xxxgnLdPxMsaYCLv3lfOK1nZuQeYTaUCFJgHp
HtvnkMk2tJAEdesCD//Mh1l7/GFLIwqeTL6LiD9WY9qu1WHDxa3OgFOHenjGiYb1
r8ZUay1K6Wnb9f7OZ3EViuDILZAxVP9KmXGfJzuOdVsqAzHzUFmBRv+vgnynPt9N
Ijpp79QpgqKFYD8z0T7lZFoxAYxnpTH9EntKunktr5qhMGrj5lIHmfvE9rwAyr/p
7clwDR250BgGZMt6pCycALcC9MBQqzuoXwVNySerAR3ASGefOnxaU+FvId/7ZXxL
SSGeeFNtMU9wS4NvWGjWmuytZSXvYgEdSnMUejwyJOMEO0zvzfX7A974ZJMidBGF
jvpr7lp7F3ew+oiOhRIGihOTOcOvfbJ/iPZNmPNIeGBAfZI32pQJWgNseYVYa32T
SziBUxqEkEbJessMPQf7VMhndFmPs0ocOMC8Ez4ahgAj/EadmaYqgJWya5bzKpiC
PrMXw8fxuDYlASs1q9IL0ShTX4YV1/An4XHpvFg25gObgksXoUv5S3mVajbpchzC
U4JETPET5awzGG2gE7TU07zgBLs4GZG36qHqZc83kpheF4NZChrhj7l+9MkldwuA
t9s7dof6t2xhu+xir8D3YY24mA9xBXxN4Ox2HHhdmJRFx22VPpOZXOnQD1ufoFja
CkEl3DSuMh3gxXbrVonsJZMAG43PIizWPSDAYyXAAQADeponNiBnbfTy54WanGCs
```

7Niewa6SlIJncQ8c7966PUNNGGoqkYRwmwzOF5Bl5Hn5cDo3JYtiLpR9CTaLKx5Y
t7U3r3HEj/EWdSx5/AgyadB16Ij1aoBPZKCvRYsiLhe0p0r452RSMJF9k6bAm6/T
SQB7O1ucV4L3mo9I3N3bAEX+zbyLs6ymeL1Gq0cyW2ESPqQ21wLooCMU1IAnmH3M
rJfvEGKKDP5jk/n+uj2e5cDHJUChJ5MUopmpgikHT2wa6OUcXUkiK2DtYEJDz886
OhNkJJPYBlEMVJ8RqmwVOjH/ZcGROGKpcwCWx1dhC/6e3iRETtDhUXG7oc+/bJkX
M3vJO6A3wSNKGOmQV2T+YlrOW05g2DiI3WbXCiZ/QcmmHYsQtgxuixnfqbKZixuL
6wIDAQAB
-----END PUBLIC KEY-----

References

Bray, T., 2014. The javascript object notation (json) data interchange format .

Housley, R., Polk, T., Ford, D.W.S., Solo, D., 2002. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 3280. URL: https://www.rfc-editor.org/info/rfc3280, doi:10.17487/RFC3280.

Nakamoto, S., 2008. Bitcoin: A peer-to-peer electronic cash system URL: https://bitcoin.org/bitcoin.pdf.

Peter Maymounkov, D.M., 2002. Kademlia: A peer-to-peer information system based on the xor metric. IPTPS URL: http://www.cs.rice.edu/Conferences/IPTPS02/109.pdf.