Libri: opinionated, high-performance decentralized storage suitable for health data

Drausin Wulsin *

September 24, 2018

Abstract

We describe Libri, a decentralized storage network suitable for heterogenous healthcare data. Libri does not depend on any blockchain or distributed consensus protocol. This simplifies its implementation and allows its performance to scale very well with horizontal cluster size. Libri documents are automatically (re-)replicated across network peers to ensure robustness to planned and unplanned peer outages. The Libri API exposes simple `Put`/`Get` endpoints and a streaming `Subscribe` endpoint for storage notifications across the entire network. We describe initial experiments measuring request load and cluster size on request latency, rate, and throughput. In particular, a modest 64-peer cluster with 1 CPU and 6 GB RAM per peer easily handles over a million documents uploaded per day with median `Get` and `Put` latencies of 6 ms and 22 ms, respectively. Request latency and throughput scale well with horizontal cluster size increasing from 8 to 64 peers. These initial results give us confidence that Libri can scale to efficiently handle production-level load for healthcare use cases.

1 Introduction

The problem of health data aggregation and sharing presents distinct challenges and opportunities. Health data is extremely sensitive and highly regulated. At the same time, the US healthcare system is very decentralized – each patient has many providers over the course of their lives or even a single acute episode, and therefore many sources and stores of data. This data fragmentation is costly to providers who bear high operational costs of error-prone point-to-point coordination with other providers conducted by fax transmissions and physically couriered CDs. This data fragmentation is also very costly to patients, whose health outcomes bear the direct cost of any information transmission lapses, and are at least burdened by the inability to exercise their rights to access and share health data.

*drausin@libri.io

Libri responds to these challenges by envisioning a cooperative pooled data asset, enabled by a novel approach to end-to-end encrypted decentralized storage. A pooled data asset facilitated by Libri would not be controlled by a single entity (corporate or otherwise), would run on open source software, would enforce strong encryption and security, and would be both performant and durable. Moreover, in appropriate domains such as healthcare where non-monetary incentives for data-pooling exist, Libri is able to deliver on this promise without reliance on blockchains, smart contracts, and decentralized identity management, which add considerable technical and operational costs.

We describe Libri[1] in detail in the rest of this paper. Section 2 covers a high-level overview of related decentralized storage technologies, the current state of health data sharing technology, and mentions some other decentralized healthcare network efforts. Section 3 introduces the high-level Libri responsibilities and architecture. Section 4 addresses some considerations resulting from this design. Section 5 describes some implementation details around how we envision organizations deploying and managing their fleet of Libri nodes. Section 6 discusses some initial experiments and promising performance results. Finally, Section 7 gives our high-level vision for the Libri community and what technical work we see before us.

2 Related work

2.1 Decentralized storage

Decentralized storage has existed in various forms for decades, but BitTorrent [1, 2], released in the early 2000s, was one of the first networks to gain widespread participation. BitTorrent makes peer-to-peer file storage efficient and contains incentives (between "seeders" and "leechers") in order to keep the network healthy. Kademlia [3] is the distributed hash table (DHT) used by almost all contemporary decentralized storage system (including BitTorrent, which now uses it for decentralized peer tracking). Kademlia provides a simple, efficient protocol for peers to use when finding and storing values within the network.

InterPlanetary File System ("IPFS") [4] uses much of the same design as BitTorrent—peer-to-peer file sharing, possible caching of popular documents, tit-for-tat incentive accounting—but it addresses the stored data by a hash of its content instead of a filename. This content-addressing combined with a simple link structure allows it to behave as a Merkle DAG, giving it great flexibility in being able to store and model many forms of data, from simple blobs to files to whole filesystems. At its heart, though, IPFS looks very much like BitTorrent: peers host data that others may optionally copy and host as well, and the addresses of the peers hosting each object are stored in a Kademlia DHT.

[1] https://github.com/drausin/libri

Peer-to-peer data storage and sharing networks are powerful, but the networks themselves are unable to offer any guarantees about the data stored in them. If the only peer hosting some data goes offline, that data is lost from the network. Absent sufficient incentives to operate a storage node, networks are unable to offer durability and performance guarantees required for most production-level storage needs. Over the last few years, blockchain-based networks such as Sia [5], Storj [6], BigchainDB [7, 8], Ethereum Swarm [9, 10], and Filecoin [11] have been developed in an attempt to solve the incentives problem: why should any peer dedicate storage space, bandwidth, and machine resources? These different networks each tackle this problem in slightly different ways, but they all rely on an economic incentive for peers to store (and keep storing) data. This economic incentive requires monetary transactions between parties, and those transactions require a blockchain.

We expect most readers to be familiar with at least the high-level concepts of a blockchain. Transactions are gossiped among a network of peers, which work together to order and package blocks of those transactions together. Transactions and blocks are immutable. Each block points to a previous block, forming a chain. So, if I have some data that I want to store in a decentralized network, instead of having to host it myself, I can pay another peer (or peers) to do so instead. These decentralized storage economies are still in their early periods with major technical and economic details still being worked out.

While the incentives that a blockchain allows add an important feature to a storage network, they also saddle it with a slow, complex component. Ethereum has a current cap at 15 transactions per second, with a block time of about 15 seconds. These limits define the request rate and response time for any storage solution using Ethereum (e.g,. even via an ERC20 token). To anyone who has worked with commodity data stores (e.g., MySQL, PostgreSQL, DynamoDB, Cassandra) to build real-world applications, a data store capable of no more than 15 transactions per second and an eventual consistency window of up to a minute[2] represents an impractical step backward.

Blockchains and the incentive systems they support also add non-trivially to the maintenance and coordination complexity of the storage system. When new storage features and bugs fixes are designed, implemented, and deployed, developers must account for how these changes will (or will not) interact with the incentive system. These considerations slow development of the core storage capabilities.

While financial transactions do justify this complexity, many different domains do not necessarily. In the particular domain of health data storage, we make the following assumptions:

- the community of organization running peers is gated (often for regulatory

[2]Since it usually takes a few blocks to be sealed for a user to have confidence that their transaction is still on the longest chain, the total time for a confirmed transaction at 15 seconds/block is perhaps between 30 and 60 seconds.

compliance);

- organizations run peers in low-variance cloud infrastructure;
- the benefits of read/write access to the shared data repository outweigh the costs of running the peers.

Coupling these assumptions with a narrow, storage-only (i.e., no processing) feature set allows us to avoid the complexity of a blockchain.

2.2 Health data sharing

Today, if patients have any access to their electronic health records (EHRs), it is most often through a health system's patient portal, where patients authenticate into a site showing things like lab results, appointments, and prescriptions. While these portals are indeed better than nothing, they rarely contain the more detailed EHR documents (e.g., clinical notes), and they often omit EHRs from whole groups within the system that aren't yet integrated with it. Many small-to-medium sized practices don't have any patient portal. These patient portals are also read-only: patients cannot upload their EHRs from portal A into portal B. Some portals are part of regional health information exchanges (HIEs), where some doctors can log in and see records from others, but like the portals, these HIEs have incomplete EHR and doctor coverage even within a large metropolitan regions. Some health systems are exposing RESTful APIs (often in the form of the HL7 FHIR standard) that allow 3rd parties like Apple Health to query and aggregate them. While these APIs are good progress, the vast majority of doctors and EHRs are not accessible via them. Like the patient portals, these APIs are also generally read only for clinical data and thus do not fully solve the problem of transferring a subset of EHRs from doctor A to doctor B.

In recent years, a bevy of blockchain-based EHR storage proposals have appeared, including DokChain [12], Coral Health [13], Embleema [14], MedicalChain [15], MediBloc [16], MedRec [17], and Patientory [18]. Many use ERC20 tokens [13, 16], direct Ethereum smart contracts [17, 18], or a private Hyperledger blockchain [12, 15]. Those papers [12, 13, 16] that discuss storage in any concrete terms mention only in passing using IPFS without any regard for the durability or performance challenges we discuss in the previous section. While we are glad to see similar ideas taking shape across the healthcare ecosystem, most of these efforts are still very much in their infancy (and some are mere proposals), often with very few technical details available to the public.

2.3 Libri's place

Libri focuses only on storing and sharing data in a decentralized manner. We believe this decentralization is necessary to include as many organizations in the healthcare ecosystem as possible. Higher-level needs like authentication, identity, and EHR-integrations will be built on top of Libri but most likely as conventional, centralized applications communicating with the shared Libri

network. Healthcare business logic is hard enough to manage when building in a centralized world; we don't see the need to complicate higher-level logic as well by prematurely decentralizing it. A decentralized storage core is sufficient.

3 Architecture

Peers in the Libri network are called Librarians, and clients of these peers are called Authors. Librarian peers never see the plaintext content of a document and deal only with encrypted chunks of one. Author clients convert a plaintext document into these encrypted chunks and back again. This distinction means that data stored in Libri and all data encountered by Librarian peers is fully end-to-end encrypted.

Librarians are responsible for the following:

- storing/retrieving documents,
- maintaining routing tables of other peers in the network,
- gossiping storage events to other peers,
- re-replicating data if it becomes under-replicated.

The storing/retrieving and routing table behave quite similarly to a standard Kademlia [3] distributed hash table (DHT). The storage event gossiping and re-replication responsibilities add two important capabilities on top of it. Section 3.1 below describes the Librarian API in more detail. Each Librarian exposes the same synchronous API for other Librarians and Author clients to make requests against.

Each Librarian and Author has an identity defined by the 256-bit public key of an secp256k1 ECDSA key-pair. Librarians and Authors sign each request with their private key (see Section 3.3 below for more details), and each Librarian's public key also defines its location on the Kademlia hash ring. Using a public key for peer identity and request signing closely follows the approach in S/Kademlia [19], though the node ID is just the public key rather than a hash of it as in S/Kademlia.

Authors are clients responsible for uploading, downloading, and sharing content in the Libri network. In these capacities, they handle the following:

- compression/decompression,
- pagination/concatenation,
- end-to-end encryption/decryption,
- Libri network Put/Get requests.

Author clients by default compress all data with GZIP unless otherwise specified. This compressed stream is then split up into Pages (or chunks) of 2 MB or less. Each Page is individually encrypted and uploaded (via a `Put` request) to Libri. When retrieving data from Libri, the Author client is also

responsible for the reverse process of downloading, decrypting, concatenating, and decompressing the chunks into the original byte stream that was uploaded. Currently, the Author client is implemented as a librarian/SDK, akin to other client-side storage libraries for services like AWS S3.

Both Librarians and Authors may subscribe to the gossiped storage events of other Librarians. Usually each Librarian or Author will want to receive a complete log of all events. In a blockchain-based system, there is usually a single log (i.e., the "consensus"). In Libri, each peer maintains its own log, which with very high probability captures all storage events. The individual logs of each peer have exactly the same events in them with high probability, but very small differences in ordering and timing (< 1s) will exist due to gossip path differences. We believe these small differences are well worth the performance benefits and implementation simplicity of avoiding a blockchain or consensus protocol. Organizations running fleets of nodes may post consolidated versions of these store logs for the public to download, compare, and query if they desire.

3.1 Librarian API

Each Librarian exposes a synchronous service with the following endpoints:

- `Introduce` receives a peer ID and IP address and returns a random sample of other peers from the routing table.
- `Find` receives a document key and returns either the value for that key if present or the peers closest to it the routing table.
- `Verify` receives a document key and an HMAC key and returns either the HMAC of the value for that key, if present, or the peers closest to it from the routing table.
- `Store` receives a key-value pair to store at that peer.
- `Get` receives a document key and returns the corresponding value, if it exists, managing the recursive `Find` operations for the client.
- `Put` receives a document key-value pair and stores the value with the appropriate peers, managing the recursive `Find` and final `Store` operations for the client.
- `Subscribe` receives public key Bloom filters and (continually) streams `Publications` matching the filters to the client.

The `Find` and `Store` endpoints follow the standard Kademlia protocol. `Introduce` is used when a new peer is created and needs to bootstrap some initial other peers into its routing table. `Verify`, used by peers to ensure sufficient replication of the documents they contain, behaves very similarly to `Find` except that it returns an HMAC instead of the value. `Get` and `Put` are mostly intended for Authors to call (rather than other Librarians), especially since they involve the receiver of the call making a number of `Find` and/or `Store` calls. In Section 4.1, we discuss authorization and rate limits for endpoints between peers and

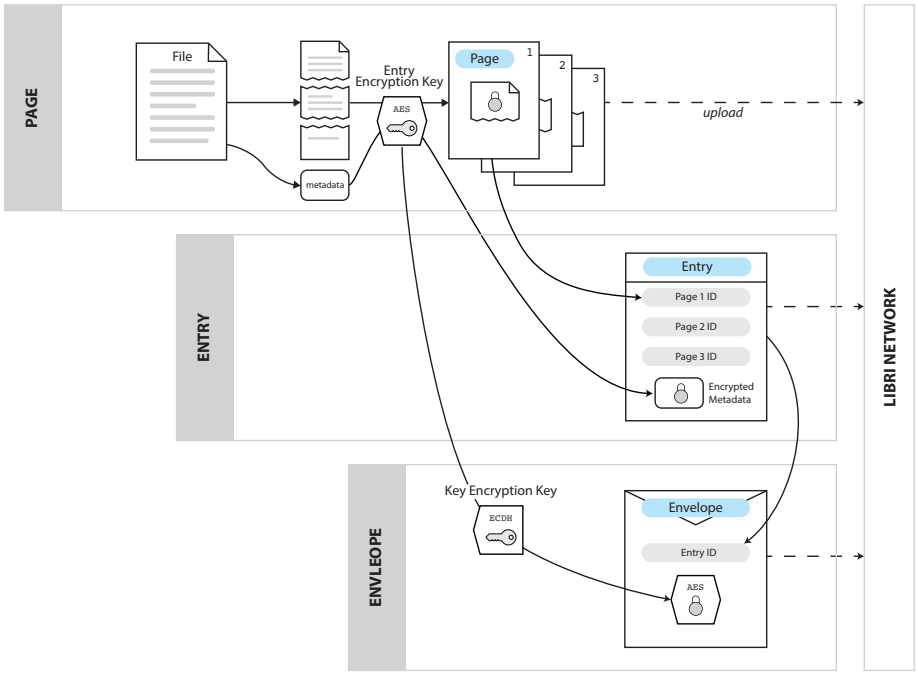

Figure 1: Libri documents take one of three forms: `Page`, `Entry`, or `Envelope`. The contents are split into one or more `Page`s and encrypted with an AES-256 Entry Encryption Key (EEK). The IDs of each of these pages are stored in an `Entry` along with encrypted metadata. The EEK is encrypted with a ECDH Key Encryption Key (KEK) and stored with the `Entry` ID in an `Envelope`. Multiple `Envelope`s are usually generated for a single `Entry` since one often wants to share an `Entry` with multiple recipients.

clients.

`Subscribe` allows peers to listen to the publications emanating from or relayed through other peers. Librarians will usually subscribe to the $O(10)$ other Librarians, receiving almost all of each subscribed peer's publications, meaning that each peer would receive at least one publication notification for every true publication event with very high probability. Authors not interested in the full publication log would instead subscribe to $O(10)$ Librarians with Bloom filters for specific sets of author and/or reader public keys that it is interested in.

3.2 Libri Documents

Libri documents take one of three forms:

- A `Page` holds a particular sequential chunk of the (compressed) document content.

- An `Entry` holds the encrypted contents of the document, either as a single `Page`, or as a list of keys to separate `Page` documents.
- An `Envelope` contains the entry encryption key (EEK) between a specific author and reader.

Documents are serialized to a binary representation for storage in Libri. The key of any document is the SHA-256 hash of its value bytes. Figure 1 shows an overview of these three documents. Below we discuss some particulars of each document type.

3.2.1 Page

A `Page` contains some or all of the content for a single document. The maximum `Page` content size is 2 MB[3], from our assumption that at least 50% of documents will have sizes less than 2 MB. For now, our pagination strategy is very simple: we just split the (compressed) plaintext of the content into consecutive 2 MB chunks. We expect that this straightforward approach, combined with a reasonable replication strategy (e.g., 3-5x) will be sufficient to ensure clients always have access to the requisite chunks required to reconstruct their document. If this assumption proves problematic, we can always implement more sophisticated chunking via erasure coding in the clients. The Libri server code is agnostic to the chunking strategy.

Each `Page` document has

- the public key of the author[4] that created it,
- the index of the particular `Page` to define the order in which subsequent `Page`'s plaintext content should be concatenated,
- the ciphertext of the `Page`'s portion of the content,
- the ciphertext MAC.

3.2.2 Entry

An `Entry` defines the content of the document. If the total content can fit on a single `Page`, the `Entry` contains that single `Page` within it, and that `Page` is not stored separately in Libri. If the content is large enough to require splitting across multiple `Page`s, the `Entry` just contains the relevant `Page` document keys. An Entry also contains the public key of the author, a creation timestamp, and encrypted metadata and a corresponding MAC.

`EntryMetadata` contains attributes like the media/MIME type of the data,

[3]This maximum page size conveys our assumption that large clinical records are split up into small, granular documents. For example, a patient's clinical history with a doctor might be split up into individual records for each encounter.

[4]We distinguish between Author clients of Libri, which have their own public key ID, and author public keys, which are used to indicate the creator of a particular document. A user will typically have many author public keys and many reader public keys. Section 3.4 describes these different sets of keys in more detail.

compression codec, full byte size of the entire document (across all `Page`s), and content schema references. See Appendix B for more details.

A random 108-byte entry encryption key (EEK) is generated whenever an `Entry` is created. The EEK contains four sub-keys:

- 32-byte AES-256 key for `Page` and entry metadata encryption,
- 32-byte `Page` block cipher initialization vector (IV) seed,
- 32-byte HMAC-SHA-256 key for ciphertext MACs,
- 12-byte metadata block cipher IV.

The contents in each `Page` and the `Entry` metadata are encrypted via an AES-256 GCM block cipher. `Page` i's plaintext content is encrypted using the EEK AES-256 key and a per-page IV generated from the first 12 bytes of `HMAC-SHA-256(IV seed, i)`, and a MAC for this ciphertext is calculated from the EEK HMAC key via `HMAC-SHA-256(HMAC Key, Page Ciphertext)`. The `Entry` metadata is serialized to its Protobuf binary representation and encrypted with the EEK AES-256 key and 12-byte metadata IV.

3.2.3 Envelope

Envelopes exist solely for the purpose of sharing EEKs from the author of a document to a reader (each identified here just by one of their public keys). The 108 byte EEK is encrypted in the `Envelope` using the key encryption key (KEK) derived from the shared ECDH secret between the author and reader keys. The KEK contains three sub-keys:

- 32-byte AES-256 key for EEK encryption,
- 12-byte block cipher IV,
- 32-byte HMAC-SHA-256 key for ciphertext MAC.

This 76-byte KEK is derived from a SHA-256 hash-based key derivation function (HKDF) initialized with the x-coordinate of the shared ECDH secret. The 108 bytes of the EEK are encrypted via an AES-256 GCM block cipher using the AES-256 key and block cipher IV in the KEK. A MAC of the resulting ciphertext is also calculated via `HMAC-SHA-256(HMAC Key, EEK Ciphertext)`.

A complete `Envelope` contains

- document key of entry whose EEK this envelope encrypts,
- author public key,
- reader public key,
- EEK ciphertext,
- EEK ciphertext MAC.

Since usually an author wants to be able to later read a document they create, they first send an `Envelope` to themselves, using another one of their public keys

as the reader public key. This self-share allows clients to avoid locally storing the EEK plaintext, since they can always decrypt their self-shared `Envelope` if/when they want to share the `Entry` with someone else.

When they do want to share a document, they create a new `Envelope` re-encrypting the EEK using the KEK derived from the shared ECDH secret between one of their author key-pairs and one of the reader. One must then only monitor the `Envelope` publications for those with author or reader public keys that they know they own in order to see what `Entry`s they can decrypt.

3.2.4 Publication

`Envelope` storage events are gossiped between Librarian peers in the form of `Publication` messages, which are exchanged via the streaming `Subscribe` request from one peer to another. A `Publication` contains a subset of the fields of the `Envelope`:

- envelope document key
- entry document key
- author public key
- reader public key

Each Librarian constructs its own `Publication` stream, which it populates from `Subscribe` requests to other Librarians and forwards to Librarians that are subscribed to it. Because these `Publication`s are gossiped, one Librarian may have a slightly different `Publication` order than another, but the differences should be quite small (< 1s). Organizations running Librarian peers may wish to set up Author clients `Subscribe`d to them so that the clients can save these `Publication`s to some more durable storage like a database or message queue.

3.3 Identity

Like most other cryptographic systems, Author and Librarian identity relies on elliptic curve (secp256k1) public keys. Each request to the Libri API contains metadata with a unique, random 32-byte request ID and the public key of the requester. Requesters create a JSON web token (JWT) containing a single claim—the SHA-256 hash of the Protobuf binary message—and sign it with their private key.

Organizations will typically run a fleet of Librarians, perhaps 8 or 16. Each Librarian will have its own distinct ID, but the organization may wish to identify all of them as belonging to the same organization. When this is the case, the organization generates its own public-private key pair and securely distributes that key pair to each of its Librarians. Each Librarian then includes the organization public key in the request metadata and also includes the same JWT signed with its organization public key.

When Librarians receive requests, they first verify the peer public key signa-

ture of that request. If an organization signature is present, they verify that as well. Organization IDs in particular allow a Librarian to segment the requests it receives into tiers of trust. Organization A could configure its Librarians to trust requests from organizations B, C, & D, whereas those from others may be treated more skeptically. We discuss the authorization and rate limit results of these differentiated trust tiers below in Section 4.1.

3.4 Authors

Authors are the clients of the Libri network: they write and read documents. To convert binary content into the documents to be written to the network, authors follow a basic process of compression, pagination, and encryption.

Compression is optional but recommended, since its performance cost is usually low and can result in much smaller files that need to be uploaded and stored. The compression codec used is included in the `Entry` metadata. The compressed content is then split up—paginated—into chunks of at most 2 MB. Each of these chunks is then individually encrypted using the EEK, which is generated randomly for each content. The EEK is encrypted by the KEK generated from the shared author-reader ECDH secret and is used to create the `Envelope`. When small content requires only a single `Page`, that `Page` is stored within the `Entry` for the content. The Author then `Put`s the `Envelope` and `Entry` into the network via calls to one or more Librarians, which store each document with the appropriate other Librarians close to its key in the Kademlia hash ring.

Downloading a document follows a similar pattern in reverse. The Author client first `Get`s the `Envelope` for a given document key and confirms that it has the private key indicated by the reader public key in the `Envelope`. Assuming it does, it constructs the KEK and uses it to decrypt the EEK ciphertext. It then `Get`s the `Entry` indicated by the `Entry` key in the `Envelope` followed possibly by the additional `Page`s indicated in the `Entry` and uses the EEK to decrypt the `Entry` metadata, which gives—among other things—the compression codec used. `Page`s are iteratively decrypted, concatenated, and decompressed to for the final binary content.

When one author wants to share a document with another (which we designate the reader), they only need to create a new `Envelope`, since the reader just needs to receive the (encrypted) EEK rather than the whole document re-encrypted. The author gets one of the reader's public keys [5] and samples one of its own key-pairs to construct the KEK. The author then encrypts the EEK with the new KEK and includes its public key, the reader's public key, the key of the `Entry` it is sharing, and the EEK ciphertext in the newly constructed `Envelope`.

Each client maintains two sets of key-pairs: author keys and reader keys.

[5] The author gets one of the reader's public keys either directly via email or QR code or indirectly via a 3rd party that the Reader has registered some of its public keys with

Technically, each client could just have a single key-pair they use to send and receive documents. But if every entity using Libri only had a single key-pair identifying them, one could potentially re-identify patients based on their (data-sharing) relationships with doctors. Similar re-identification attacks have been shown to be successful on "anonymized" credit card transactions. Since even the knowledge that a patient visits a particular doctor is protected health information (PHI), clients must have more than one key-pair. For a further layer of anonymity, the keys clients use to send documents (the author keys) are distinct from those they use to receive them (the reader keys). This distinction means that knowing one of they keys a doctor uses to receive documents a patient might share with them does not let the patient see even a subset of the documents that doctor is sharing with other patients. In our initial Author client implementation, each Author has 64 author key-pairs and 64 reader key-pairs. Each key-pair is individually encrypted via scrypt [20] using a master password.

4 Considerations

4.1 Incentives & Authorization

A key result in Libri's avoidance of a blockchain and tokens is the lack of any economic incentives for peers to participate in the network. This absence of economic incentives in the face of real economic costs [6] requires that the organizations get some benefit to offset the cost. We believe this benefit is unmediated, read/write access to what will become a massive repository of encrypted healthcare data. While $5000 per year of infrastructure costs certainly is not nothing, is it small relative to the current alternatives: paying humans to mediate this data exchange through more analog methods (fax, snail mail, CDs in padded envelopes, or even wrangling email attachments). These economics do not make much sense for individual hobbyists in the way they do for other decentralized systems, but they do for larger organizations that have much to gain from simple, efficient access to what will be a vast repository of health data. We hope that government organizations will participate as well, especially since a durable repository of all health data should be considered a public good and thus worth supporting for the benefit of all. Since hobbyists will have much less incentive to run peers, we expect that the peers run by organizations will have much higher uptime (basically 100%) and more consistent resource guarantees, which together ensure that the network as a whole operates efficiently and with lower performance variance than other decentralized systems.

Since requests between Libri peers are "free" of economic costs, we restrict their usage via simple rate limiting. Each peer maintains per-second and per-day rate limit counters for requests from every other peer and for each Libri endpoint. They also maintain limits on the number of distinct peers they may receive requests from per second and per day. Peers have the concept of "know-

[6] We estimate that a modest 8-peer fleet at 1 CPU, 6 GB RAM, & 100 GB storage per peer would cost about $400 per month on Google Cloud Platform.

ing" other peers or the organizations they belong to. If a peer is configured (e.g., via a peer and/or organization ID whitelist) to know certain other peers, it may allow higher rate limits from those peer. The extreme example of this form of authorization is that an organization probably will want to restrict `Get` and `Put` endpoint requests to only its own clients, since each `Put` and `Get` request requires the server to make a number of additional `Find` and `Store` requests.

Occasionally, Librarians peers will fall into a bad or unavailable state, perhaps because they are momentarily undergoing some sort of routine maintenance (e.g., being redeployed with an updated version or having their local storage backed up) or because they are legitimately no longer available. In either case, we wish to proactively avoid sending requests to those peers. Each Librarian maintains a current healthy vs. not-healthy state of every other Librarian in its routing table. This state is updated after every request (both successful and unsuccessful) to another Librarian. Each Librarian also has a healthcheck endpoint that others may call to confirm that it is up and receiving requests. By avoiding sending requets to known-unhealthy peers, Librarians reduce the variance and latency of some of the more involved query patterns, like those required for `Get`s and `Put`s.

4.2 Durability

When clients store documents in Libri, they store them in the network as a whole, rather than with specific peers, as is the case with other decentralized storage networks[2, 4]. The Libri network is thus responsible for ensuring that, once stored, documents are never lost due to peer drop-outs or network connectivity issues. Librarian peers thus are responsible for maintaining their own locale of the hash ring, ensuring that documents they're storing are sufficiently replicated and storing additional copies in other peers if they become under-replicated.

Let's say that organizations A, B, C, & D are each running a fleet of 8 peers, but then organization D decides to stop running these peers. A, B, & C are responsible for storing the additional copies of D's documents. These peers must first detect that documents are under-replicated and then send additional `Store` requests to each other to bring the replication level back to normal.

In addition to serving synchronous requests, each Librarian also has an asynchronous process that loops through the documents in its own internal storage. For each document, it sends out `Verify` requests, which behaves almost exactly like the `Find` requests except that if the peer has the document, instead of returning the value, it returns a MAC of the value using the HMAC key given in the request, thereby proving that they do in fact have the document of interest. Most verification operations will conclude with the other peers storing the replicas successfully proving they they do indeed have the value stored.

If one of the peers that once stored the value no longer has it or is unavailable, the verifier will detect that the document is under-replicated and will issue a

series of `Store` requests (the same as what would occur during a `Put`) in order to re-store the document on additional peers. Because each peer stores documents with keys close to its ID on the hash ring, the requests it will make to verify and possibly re-replicate documents will be within its local neighborhood on the ring and thus will be fast and efficient.

Librarians currently wait 1 second between verifications, meaning that if a Librarian is storing n documents, it is verifying the replication of each document on a period of roughly n seconds. Assuming a replication factor of k, the network as a whole is thus verifiying each document on average k/n seconds. It is hard to know exactly what the "right" verification period should be. We expect to monitor how quickly the network is able to "heal" itself after losing peers and update the replication period accordingly.

4.3 Protecting against malicious actors

As with any decentralized system where one does not necessarily have identity and/or reputation information for every peer, Libri is designed to be resilient to many different types of malicious actors. Below we discuss some of the most common forms of attack and Libri's defense against them.

4.3.1 DDOS

While decentralized systems are intended to be more resilient to distributed denial of service (DDOS) attacks than traditional, centralized services, many—Libri included—are probably small enough that a well-equipped group could reasonably DDOS all nodes. In the event of an attack, each organization running Librarian peers could choose to change their firewall rules to block all external traffic, partitioning their fleet from the rest of the (overwhelmed) network, and operate in degraded read-only mode. Depending on the overall size of the network and the number of peers in their fleet, organizations might still have access to a non-trivial subset of the documents.

We also expect that organizations running nodes to usually maintain some local copy of the subset of data in Libri they care about, since interacting with that local copy will always be faster and less variable than the Libri network. In the face of a prolonged, Libri-wide DDOS attack, organizations may decide to fall back to degraded read-only mode against their local caches and pause writes to Libri until the attack has subsided.

Of course, adding nodes and partner organizations to increase the size and diversity of the network is the best defense against DDOS.

4.3.2 Spam

A spam attack might take the form of one or more clients issuing many `Get` or `Put` requests, potentially trying to download or upload many terabytes of data in order to overwhelm the network. We expect that unknown vs. known peer

and organization rate limits will reduce what otherwise could be a massive flood of requests to a much smaller fraction of the total. For example, a Librarian might only accept 1 `Find` request per second from all unknown Librarians but 50 requests per second from all of its known peers and organizations. It might accept zero `Store` requests from unknown peers and 25 requests per second from its known peers. While such rate limiting doesn't eliminate the spam problem entirely, it can reduce its impact to be just a small fraction of the overall requests.

4.3.3 Sybil

A Sybil attack occurs when one actor creates many peers in order to disrupt the network's regular operation, perhaps by ignoring or disobeying requests (e.g., not storing data when it says that it has). Like DDOS, the feasibility of this attack is inversely proportional to the size of the network. Furthermore, since peers maintaining whitelists of "known" organizations will likely have much strictly rate limits on unknown peers, new peers may not receive many requests or have the opportunity to send many bad requests.

4.3.4 Honest Gepetto

In the honest Gepetto [6] attack, an organization might run legitimate, well-behaved nodes for some time before pulling their peers off the network all at once. Since each document is replicated a number of times, the probability that their nodes would contain all copies of some document is very low, effectively eliminating the risk of full data loss. But the documents these peers did store would now be under-replicated. The remaining peers in the neighborhood of each pulled peer would manage re-replicating documents up to their sufficient replication level.

5 Implementation

Libri is implemented to be as simple to develop and maintain as possible. We thus use off-the-shelf tools when available and strive to be specific and opinionated rather than overly flexible and generic. Below we describe some implementation details.

5.1 Librarian peers

Librarian peers are intended to be run from Docker containers in a cloud provider, like Google or Amazon. These containers are orchestrated via Terraform and Kubernetes, which also manage the persistent SSDs attached to each container for storage, a Prometheus server for monitoring and alerting, and Grafana server for dashboards. While it certainly is possible to run a Librarian peer on a laptop, we orient towards cloud deployment and infrastructure to standardize configuration and make use of the superior reliability, performance, and features offered there.

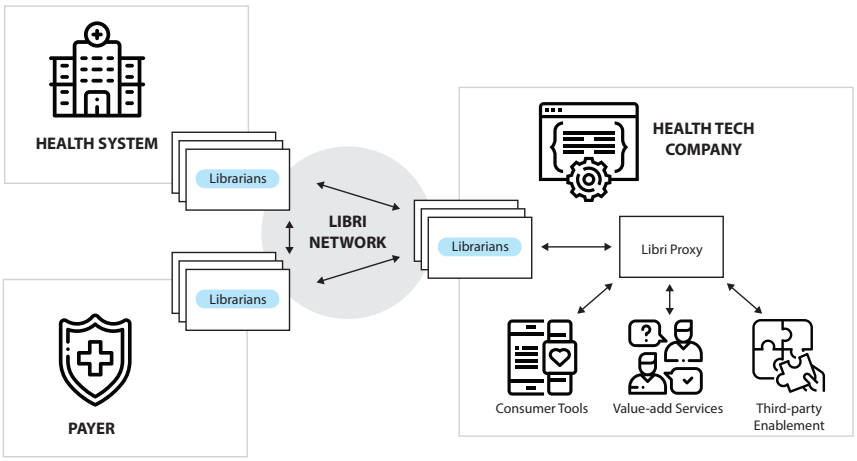

Figure 2: A hypothetical set of Libri users. Larger organizations like health sytems, payers, and health tech companies run their own Librarian nodes. Consumer-facing tools, analytics, and other third party applications use a Libri proxy service maintained by a tech company.

Each Librarian exposes an RPC service over http. The service interface is defined in GRPC, which uses Protobuf for message serialization. GRPC has been battle-tested at Google for over the last decade and has server and client libraries in most common languages. It also has nice features like streaming endpoints, which we use when gossiping publication events between peers. We expect to only develop and maintain a single server implementation in Golang.

Librarians use RocksDB for local storage. RocksDB is an embedded key-value store maintained by Facebook and is optimized for fast writes on SSDs. Each Librarian's RocksDB directory is written to a network-attached SSD volume, which is incrementally backed up to durable cloud storage (e.g., S3).

5.2 Hypothetical organization's setup

An organization runs peers to get read/write access to the DHT as well as the stream of all publication events. A modest integration might look like the following.

The organization runs 8 peers that bootstrap from long-lived peers and introduce themselves to the rest of the Libri network. They may also have an internal service that uses the Author client library to proxy Put and Get requests to Libri via their 8 Librarian peers. This service may also send publication notifications on to an internal message bus (like Kafka) or filter them down to only those involving that organization (via their set of public keys). If they see that someone just shared a document with one of their public keys, they could then use Author client library download, decrypt, decompress, and

concatenate the relevant Pages before storing in their own internal data system (which presumably uses its own encryption at rest and in transit).

Smaller organizations and almost all consumers will not want to run their own peers. We expect an ecosystem of 3rd-party companies to build consumer-facing apps and APIs that will proxy access to the data in Libri much like companies like Coinbase proxy a consumer's access to the underlying Bitcoin network.

Figure 2 shows a schematic of such a setup.

6 Experiments

Decentralized storage systems like IPFS, Sia, and Storj have existed for at least a few years, but we have found very few empirical examinations of their performance. We believe that part of the evaluation for storage systems like these should include how efficiently they are able to handle the routine operations required of them. Below we describe some preliminary experiments on ephemeral Libri clusters. All the configuration and results for the experiments in their paper are publicly available[7].

6.1 Performance across cluster size and load

In this first set of initial experiments, we examined how a Libri cluster performs as the load on it and cluster size increase. In theory `Get` and `Put` requests in Kademlia-based architecture should scale as $O(log(n))$ for a cluster with n peers, but we wanted to examine the extent to which these latencies scale in practice as n increases. We also want to understand how well each of these clusters handles increasing client request load. We use the `Get` and `Put` endpoint latencies as our primary metrics of interest during these experiments, though other metrics like data throughput and cluster queries per second (QPS) are also relevant. Centralized key-value stores like DynamoDB, BigTable, Cassandra, and Riak boast request latencies in the single-digit millisecond ranges at thousands of queries per second. While decentralized storage systems will never be able to match that performance, we believe that it's valuable to strive toward roughly the same orders of magnitude if decentralized storage is going to be used in a production setting.

6.1.1 Methods

We simulated hypothetical user load with random documents in order to examine each cluster's performance. While the real distribution in healthcare document sizes is hard to know and will probably be very wide, we assume here that we are working with PDFs (unfortunately, still one of the most common forms of data exchange between healthcare organizations) on the order of a few hundred KBs. Documents were generated by first sampling a byte length from

[7] https://github.com/drausin/libri-experiments/tree/develop/experiments/exp03

user load (UPD)	`Put` + `Get` rate (QPS)	`Put` throughput (Mbps)
64k	5.8	4.3
256k	23.7	18.9
1024k	92.2	73.2

Table 1: Query rate and throughput for each uploads per day (UPD) load. Results are shown for the 64-peer cluster, but those for other cluster sizes were similar.

a gamma distribution with shape 1.5 and scale 170 (implying a mean of 256 KB and 95% interval of [18, 794] KB and then randomly generating document content of the sampled length.

In an attempt to root cluster load in our expected real-world use case of uploading, sharing, and downloading health-related documents, we define our load by uploads per day (UPD). For this experiment, we assume that each "upload" involves four distinct `Put`s: the `Entry`, the self-shared `Envelope`, and two additional `Envelope`s representing sharing the document with two other parties. We also simulate each of the share parties downloading both the `Envelope` with their public key and original `Entry`, resulting in four total `Get` requests for each upload. Each "upload" thus involves, directly and indirectly, 8 total requests to the Libri network.

We ran 12 trials, each with an ephemeral Libri cluster deployed using Kubernetes on infrastructure provisioned in Google Cloud Platform via Terraform. Each Librarian received 1 CPU, 6 GB RAM, and 10 GB network-attached SSD storage. Request latencies were captured via Prometheus monitoring and visualized in Grafana dashboards. We tested the cluster sizes (i.e., number of Librarian peers) 8, 16, 32, & 64 and 64K, 256K, and 1024K UPD. Each trial ran for an hour, since the primary goal of this experiment was to measure short-term request latencies rather than long term cluster behavior. Table 1 gives the queries per second (QPS) and throughput (Mbps) produced by each user load level.

Request latencies were measured via the 50^{th}- and 95^{th}-percentile values of the latency distributions (a.k.a., p50 and p95, respectively) for the `Put` and `Get` endpoints. These quantiles are estimated by Prometheus via its histogram counts of the request latencies[8].

6.1.2 Results and Discussion

Figure 3 shows the `Put` and `Get` latency estimates aggregated over all Librarians as the cluster size increases from 8 to 64 peers. In each of the four charts, the 1024K UPD load almost always has the best latency performance. While this result may initially seem counterintuitive, it is almost certainly an artifact of

[8] https://prometheus.io/docs/practices/histograms/

?

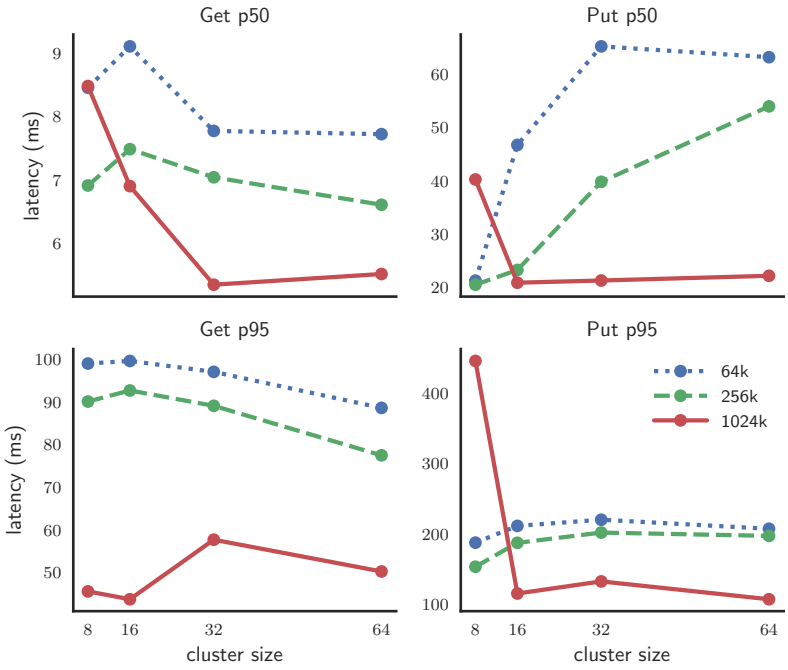

Figure 3: `Get` and `Put` request median (p50) and 95$^{\text{th}}$ percentile (p95) latencies across different cluster sizes and user loads. Each colored line denotes a different user load, in uploads per day.

the way Prometheus calculates quantiles from histograms, which tend to skew a little more conservative when the counts in each histogram bin are higher.

We first note the order of magnitude of the latencies. Median `Put` latencies are in the tens of milliseconds, and p95s are generally between 100-250ms. Median `Get` latencies are in the single-digit milliseconds, and p95s are generally in the mid-to-high tens of milliseconds. The 8-peer cluster receiving 1024 UPD is a little overwhelmed, but increasing the cluster size to 16 and then 32 peers seems to spread the load out effectively. In all but the `Put` p50, increasing the cluster size has little effect on the latencies, showing that the increased communications required with a larger cluster have little effect on the overall latency. The increase in `Put` p50 as the cluster size grows makes sense because each `Put` has to find the k Librarians closest to a document key to send `Store` requests to. As the cluster size n increases, this search time will grow proportional to $\log(n)$, and indeed the shape of the 64K and 256K UPD tends does seem roughly logarithmic.

Figure 4 shows the query rate across all endpoints and storage (`Put`) throughput of both the cluster overall and the average per peer as the UPD load and cluster size vary. The total cluster query rate shows a fairly consistent (with the exception of 1024 UPD on the 8-peer cluster) increasing linear trend of roughly 4 QPS/peer, meaning that each additional peer adds roughly 4 QPS to the total cluster QPS.

The total average peer query rate predictably drops as the cluster size increases for each UPD load. Since we know that 1024 UPD on an 8-peer cluster had anomalously poor latency results while the 16-peer clusters served that same load with good performance, we posit a roughly 50 QPS limit per peer (between the 72 QPS on the 8-peer cluster and 41 QPS on the 16-peer cluster) in order to maintain good latency performance. The cluster-wide throughputs are quite consistent across cluster size with the slightly lower throughput at 1024 UPD on the 8-peer cluster. At 1024 UPD and cluster sizes 16 and larger, the cluster throughput is roughly 72 Mbps. As above with QPS, we might similarly posit a roughly 5 Mbps throughput limit per peer in order to maintain good cluster performance.

The latency, request rate, and throughput results from this experiment show how a modest Libri cluster can achieve quite good performance: on average, `Get`s in the single-digit milliseconds and `Put`s in the tens of milliseconds when serving 80 `Get + Put` QPS with a 72 Mbps throughput. These experiments also show that while increasing the cluster size increases the number of requests within the cluster, its effects on `Put` and `Get` latencies are small, its effects on overall QPS scale linearly with the cluster size, and those on and throughput are undetectable. These results give us initial confidence that a Libri cluster will be able to scale well with the first wave of users and load associated in the first few years of its existence.

We emphasize that these experiments are preliminary, and we expect to bol-

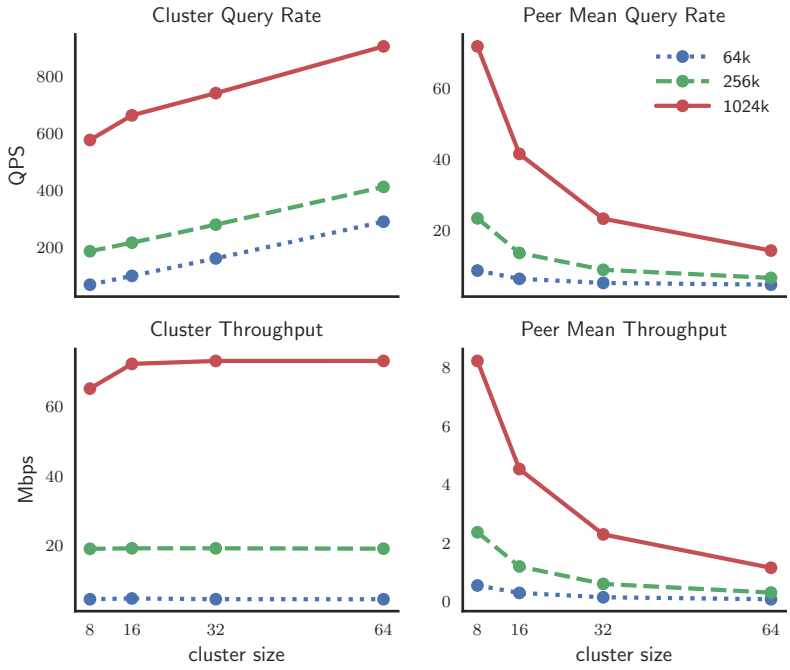

Figure 4: Query rate (in queries per second, or QPS) and `Put` throughput (Megabits per second, or Mbps) across entire cluster and for an average peer. Each colored line denotes a different user load, in uploads per day. Standard error bars on peer mean charts are omitted because they are too small to meaningfully resolve.

ster them with additional tests run on clusters distributed across cloud provider availability zones and regions. We also plan to deploy a continually-running "wild" test network (a.k.a., testnet) for us to run periodic load tests on. While many decentralized systems boast many thousands of peers, we do not think Libri should scale to that many when each Libri peer can potentially be quite large (in the resources available to it) when hosted in a cloud environment. We expect a vibrant Libri network will contain on the order of hundreds of nodes. So while these experiments on clusters up to 64 peers involve fewer peers than the size we expect in perhaps 5 years from now, we expect the size of the network to grow relatively slowly over that period of time.

7 Future Work

Libri is and will always be a fully open source network. Openness engenders trust, and trust is critical when managing something as important and sensitive as long-term health records. We hope to build a vibrant open source community, where prioritization, important features, and bugfixes are contributed by as many participants in the network as possible.

Core development will be led by Elixir Health, which will offer optional paid services on top of Libri, including web and mobile app, identity and access management, and a proxy API to Libri for other 3rd party organizations and apps not yet interested in running their own nodes.

Libri and Elixir Health welcome all members of our healthcare ecosystem: doctors and healthcare systems, public and private insurance organizations, life sciences and device companies, public health and medical research groups, and consumer health organizations like gyms, fitness, and diet trackers. Each organization has a role to play in and much to benefit from a shared, accessible repository of all health data.

The next large technical hurdle is ramping up integrations with healthcare organizations. Initially, some records will be unstructured (or semi-structured) (PDFs, images) and some will be structured (HL7, FHIR, CCD). Easily storing, accessing, and sharing these documents—regardless of the form—dramatically improves on the piecemeal system we have now. Once this unstandardized, heterogenous data flows easily, our collective incentives for better standards, compatibility, and documentation increase. We expect to spend a great deal of collaborative energy on this front. The schema and data dictionary specification given in `EntryMetadata` (see Appendix B) is intended to start us off on that direction.

8 Conclusion

The field of decentralized storage is still in its infancy, with few networks supporting true production loads. The network designs that do exist have few performance studies to give confidence that they can in fact support the production loads they aim to. Libri is intended to be simple and opinionated, which frees it from many of the challenges (e.g., consensus protocols, incentives) associated with other decentralized systems. Documents are encrypted and shared between parties through a very explicit API, and these documents are kept replicated by the network as a whole. Preliminary experimental results indicate that Libri networks exhibit very good request latency, rate, and throughput properties, giving us confidence that we can immediately begin testing a persistent cluster with production-level load.

Acknowledgements

We are grateful in particular to Jamie Campbell for the diagrams, extensive draft edits, and discussion. Thanks also to Diran Li, John Urbanik, and Punya Biswal for helpful discussions and suggestions around Libri design and draft reading.

Appendix A Document structure

Below we describe the structure of each of the three Libri document types: `Page`, `Entry`, and `Envelope`. We use Protobuf types and add a length to `bytes` arrays when it is fixed. Each document key (e.g., `entry_key`) is the SHA-256 hash of the document serialized to binary.

`Page` structure

name	type	description
`author_public_key`	`bytes[32]`	public key of author/sender
`index`	`uint32`	index of this `Page` in the `Entry`
`ciphertext`	`bytes`	encrypted contents of `Page`, up to 2 MB
`ciphertext_mac`	`bytes[32]`	ciphertext HMAC

`Entry` structure

name	type	description
`author_public_key`	`bytes[32]`	public key of author/sender
`page`	`Page`	for single-`Page` content
`page_keys`	`repeated bytes[32]`	for multi-`Page` content
`created_time`	`uint32`	client epoch time when `Entry` created
`metadata_ciphertext`	`bytes`	ciphertext of serialized `EntryMetadata`
`metadata_ciphertext_mac`	`bytes`	`EntryMetadata` ciphertext MAC

`Envelope` structure

name	type	description
`entry_key`	`bytes[32]`	document key of `Entry` whose EEK is encrypted
`author_public_key`	`bytes[32]`	public key of author/sender
`reader_public_key`	`bytes[32]`	public key of reader/receiver
`eek_ciphertext`	`bytes[124]`	encrypted EEK (including 16-byte encryption info)
`eek_ciphertext_mac`	`bytes[32]`	EEK ciphertext HMAC

Appendix B Entry metadata

`EntryMetadata` contains attributes of the `Entry` that inform decompression, decryption, and concatenation. It also optionally contains arbitrary domain-specific properties and the schema/data dictionary versions that it conforms to.

`EntryMetadata` structure

name	type	description
`media_type`	`string`	media/MIME type of the data
`compression_codec`	`CompressionCodec`	identifies compression used
`ciphertext_size`	`uint64`	size of entire ciphertext (across all `Page`s)
`ciphertext_mac`	`bytes[32]`	MAC of entire ciphertext
`uncompressed_size`	`uint64`	size of entire uncompressed content
`uncompressed_mac`	`bytes[32]`	MAC of entire uncompressed content
`properties`	`map<string,string>`	domain-specific metadata
`filepath`	`string`	relative filepath of the file to write contents to
`schema`	`SchemaArtifact`	content schema
`data_dictionary`	`SchemaArtifact`	content data dictionary

`SchemaArtifact` denotes the schema artifact associated with the serialized plaintext of a particular `Entry`. Artifacts can mainly be two separate types:

Schema can be of any type (e.g., Protobuf, Avro, JSON, XML, XSD, etc) that minimally describe the format of the data and optionally/preferably also include the data dictionary (i.e., the semantic meaning of the schema components) as well.

Data dictionary is especially important when the schema is broad/loose (as is the case in some standard data formats like HL7). This additional documentation adds clarity about the semantic meaning/use of each field. For example, a schema may contain two similar fields, A1 and A2, and one entry data producer may store a value in A1, whereas another producer may store the same semantic value in A2. The schema for both messages is the same, but the schema interpretation is different.

New schemas should obviously be as well-defined and unambiguous as possible, but many legacy data formats require additional interpretation. Clients can choose to do what they want with the schema and data dictionary, but commonly they will have combinations of these that they know how to handle explicitly.

`SchemaArtifact` structure

name	type	description
`group`	`string`	group owning the schema (commonly a Github user)
`project`	`string`	project in which schema resides (commonly a Github project)
`path`	`string`	path to schema file within project
`name`	`string`	(optional) name of the schema with the file
`version`	`string`	semantic version of the schema (e.g., "0.1.0")

References

[1] Andrew Lowenstern and Arvid Norberg. *BEP 5: DHT Protocol*. http://www.bittorrent.org/beps/bep_0003.html. 2017.

[2] Jahn Arne Johnson, Lars Erik Karlsen, and Sebjørn Sæther Birkeland. *Peer-to-peer networking with BitTorrent*. http://web.cs.ucla.edu/classes/cs217/05BitTorrent.pdf. 2005.

[3] P. Maymounkov and D. Mazières. *Kademlia: A Peer-to-peer Information System. Based on the XOR Metric*. https://pdos.csail.mit.edu/~petar/papers/maymounkov-kademlia-lncs.pdf. 2002.

[4] Juan Benet. *IPFS - Content Addressed, Versioned, P2P File System (DRAFT 3)*. https://ipfs.io/ipfs/QmR7GSQM93Cx5eAg6a6yRzNde1FQv7uL6X1o4k7zrJa3LX/ipfs.draft3.pdf. 2014.

[5] David Vorick and Luke Champine. *Sia: Simple Decentralized Storage*. https://sia.tech/sia.pdf. 2014.

[6] Shawn Wilkinson et al. *Storj, A Peer-to-Peer Cloud Storage Network*. https://storj.io/storj.pdf. 2016.

[7] Trent McConaghy et al. *BigchainDB: A Scaleable Blockchian Database*. https://github.com/bigchaindb/whitepaper. 2016.

[8] *BigchainDB 2.0: The Blockchain Database*. https://www.bigchaindb.com/whitepaper/bigchaindb-whitepaper.pdf. 2018.

[9] Viktor Trón et al. *swap, swear, and swindle: incentive system for swarm*. http://swarm-gateways.net/bzz:/theswarm.eth/ethersphere/orange-papers/1/sw%5E3.pdf. 2016.

[10] Viktor Trón, Aron Fischer, and Daniel Varga. *smash-proof: auditable storage for swarm secured by masked audit secret hash*. http://swarm-gateways.net/bzz:/theswarm.eth/ethersphere/orange-papers/2/smash.pdf. 2016.

[11] Juan Benet and Nicola Greco. *Filecoin: A Decentralized Storage Network*. https://filecoin.io/filecoin.pdf. 2017.

[12] W. Bryan Smith. *DokChain: Intelligent Automation in Healthcare Transaction Processing*. https://pokitdok.com/wp-content/themes/pokitdok2017/dokchain/static/data/DokChainWhitepaper20170926Draft.pdf. 2017.

[13] A. Park et al. *The Blockchain for Personalized Medicine*. https://docsend.com/view/4xdqkp6. 2018.

[14] *Embleema Blockchain Network: Decentralized Patient-Centric Healthcare*. http://whitepaper.embleema.com. 2018.

[15] *Medicalchain*. https://medicalchain.com/Medicalchain-Whitepaper-EN.pdf. 2018.

[16] *MediBloc: Blockchain-based Healthcare Information Ecosystem*. https://medibloc.org/whitepaper/medibloc_whitepaper_en.pdf. 2017.

[17] Ariel Ekblaw et al. *A Case Story for Blockchain in Healthcare: "MedRec" prototype for electronic health records and medical research data*. https://www.healthit.gov/sites/default/files/5-56-onc_blockchainchallenge_mitwhitepaper.pdf. 2016.

[18] Chrissa McFarlane et al. *Patientory: A Healthcare Peer-to-Peer EMR Storage Network v1.1*. https://patientory.com/patientory_whitepaper.pdf. 2017.

[19] I. Baumgart and S. Mies. *S/Kademlia: A Practicable Approach Towards Secure Key-Based Routing*. http://www.tm.uka.de/doc/SKademlia_2007.pdf. 2007.

[20] Collin Percival. *Stronger Key Derivation Via Sequential Memory-Hard Functions*. https://www.tarsnap.com/scrypt/scrypt.pdf. 2009.