A protocol & event-sourced database for decentralized user-siloed data

Draft 1.7 (Mar 2020) – Textile Threads

Sander Pick*, Carson Farmer*†, Aaron Sutula*, Irakli Gozalishvili†, Ignacio Hagopian*, Andrew Hill*
*Textile.io †Mozilla.org ‡carson@textile.io

***Abstract*—As the Internet expands, the division between a person's digital and physical existence continues to blur. An emerging issue of concern is that the digital part of a person's life is being stored away from the person's control by companies behind apps and services on the Internet. An alternative architecture for data on the web, called user-siloed data, aims to reverse the flow of data and its derived value so that users, not apps capture it. In this paper, we investigate the data formats, access control, and transfer protocols necessary to build a system for user ownership of data. The proposed system aims to help power a new generation of web technologies. Our solution combines a novel use of event sourcing, Interplanetary Linked Data (IPLD), and access control to provide a distributed, scalable, and flexible database solution for decentralized applications.**

I

I. ntroduction

Compared to their predecessors, modern cloud-based apps and services provide an extremely high level of convenience. Users can completely forget about data management, and enjoy seamless access to their apps across multiple devices. This convenience is now expected, but has come at the cost of additional consequences for users. One such consequence is that many of the same development patterns that bring convenience (e.g. single sign-on, minimal encryption, centralized servers and databases) also enable, or even require data hoarding by apps. While collecting large amounts of users' data can create value for the companies building apps (e.g. app telemetry, predictive tools, or even new revenue streams), that value flows mostly in one direction: apps collect, process and benefit from a user's private data, but users rarely have access to the original data or the new insights that come from it. Additionally, the data is generally not accessible to other apps and services. This is *app-siloed* data.

While the fact that companies collect user data may not itself be a significant problem, problems may arise if over time a company's incentives shift from *providing* value to users, to *extracting* value from users [1]. When this incentive shift happens, companies that have been creating data-silos may treat that data as new source of value or revenue for the company. It may be possible to stop this trend through extreme privacy measures, government intervention or legislation, or by boycotting companies that collect any form of data. Ideally, there is an alternative approach that allows individuals to capture the value of their data and still allow developers to build new interfaces and experience on top of this data. This approach is called *user-siloed* data and it fundamentally separates apps from their users' data.

One of the most exciting aspects of user-siloed data is the ability to build data-driven apps and services while users remain in control of their own data. Other projects have identified app-siloed data as a problem [2], [3], and some have identified user-siloed data as a solution [4]. However, none so far have addressed the problem of *how data should be collected to make it extensible*, nor have they provided a sufficiently interoperable protocol for scalable storage, transmission, and use of user-siloed application data.

In this paper, we study existing technologies that could be used to build a network for user-siloed data. We outline six challenge areas: flexible data formats, efficient synchronization, conflict resolution, access-control, scalable storage, and network communication. Based on our findings, we propose a novel architecture for event sourcing (ES) with Interplanetary Linked Data (IPLD) — which we call *Threads* – that is designed to store, share, and host user-siloed data-sets at scale. Our proposed design leverages new and existing protocols to solve major challenges with building a secure and distributed network for user data while at the same time providing the flexibility and scalability required by today's apps.

II. Background

We now describe some of the technologies and concepts motivating the design of our novel decentralized database system. We highlight some of the advantages and lessons learned from event sourcing (ES) and discuss drawbacks to using these approaches in decentralized systems. We provide an overview of some important technologies related to the Interplanetary File System (IPFS) that make it possible to rethink ES in a decentralized network. Finally, we cover challenges to security and access control on an open and decentralized network, and discuss possible solutions based on concepts from more traditional database management (DBMS) systems.

A. Data Synchronization

To model realistic systems, apps often need to map data between domain models and database tables. Often the same data model is used to both query and update a database. To solve *synchronization* of write and read storage, it is helpful to handle updates on just the database (write side) and then provide separate query and read interfaces.

One powerful approach to synchronization is to use a set of append-only logs to model the state of an object simply by applying its change sequence in the correct order. This concept can be expressed succinctly by the state machine approach [5]: if two identical, deterministic processes begin in the same state and get the same inputs in the same order, they will produce the same output and end in the same state. This is a powerful concept baked into a simple structure, and is at the heart of many distributed database systems [6].

- Logs or Append-only log A log is a registry of database transactions that is read sequentially (normally ordered by time) from beginning to end. In distributed systems, logs are often treated as append-only, where changes or updates can only be added to the set and never removed.

1) CQRS, Event Sourcing, and Logs: For most apps, it is critical to have reliable mechanisms for publishing updates and events (i.e. to support event-driven architectures), scalability (i.e. optimized write and read operations), forward-compatible application updates (e.g. code changes, retroactive events), auditing systems, etc. To support such requirements, developers have begun to utilize ES and command query responsibility segregation (CQRS) patterns [7], relying on append-only logs to support immutable state histories. Indeed, a number of commercial and open source software projects have emerged in recent years that facilitate ES and CQRS-based designs, including Event Store [8], Apache Kafka [9] and Samza [10], among others [11].

- CQRS Command query responsibility segregation or CQRS is a design pattern whereby reads and writes are separated into different models, using commands to write data, and queries to read data [12].
- ES Event sourcing or ES is a design pattern for persisting the state of an application as an append-only log of state-changing events.

A key principal of ES and append-only logs is that all changes to application state are stored as a sequence of events. Because any given state is simply the result of a series of atomic updates, the log can be used to reconstruct past states or process retroactive updates [13]. The same principal means a log can be viewed as a mechanism to support an infinite number of valid state interpretations (see sec. II-A2). In other words, with minimal conformity, a single log can model multiple application states [14].

2) Views & Projections: In CQRS and ES, the separation of write operations from read operations is a powerful concept. It allows developers to define views into the underlying data that are best suited for a given use-case the are addressing. Multiple (potentially very different) views can be built from the same underlying event log.

- View A (typically highly de-normalized) read-only model of event data. Views are tailored to the requirements of the application, which helps to maximize display and query performance. Views that are backed by a database or filesystem-optimized access are referred to as materialized views.
- Projection An event handler and corresponding reducer/fold function used to build and maintain a view from a set of (filtered) events. In general, a projection's reducer should be a "pure" function.

Views themselves are enabled by projections[1], which can be thought of as transformations that are applied to each event in a stream. They update the data backing the views, be this in-memory or persisted to a database. In a distributed setting, it may be necessary for projections to define and operate as eventually consistent data structures, to ensure all

[1]Terminology in this section may differ from some other examples of ES and CQRS patterns, but reflects the underlying architecture and designs that we will elaborate on in sec. III

peers operating on the same stream of events have a consistent representation of the data.

3) Eventual Consistency: The CAP theorem [15], [16] states that a distributed database can guarantee only two of the following three promises at the same time: consistency (i.e. that every read receives the most recent write or an error), availability (i.e. that every request receives a [possibly out-of-date] non-error response), and partition tolerance (i.e. that the system continues to operate despite an arbitrary number of messages being dropped [or delayed] by the network). As such, many distributed systems are now designed to provide availability and partition tolerance by trading consistency for *eventual* consistency. Eventual consistency allows state replicas to diverge temporarily, but eventually arrive back to the same state. As an active area of research, designing real systems with provable eventual consistency guarantees remains challenging [17], [18].

CRDTA conflict-free replicated data type (CRDT) assures eventual consistency through optimistic replication (i.e. all new updates are allowed) and eventual merging. CRDTs rely on data structures that are mathematically guaranteed to resolve concurrent updates the same way regardless of the order in which those events were received.

How a system provides eventual consistency is often decided based on the intended use of the system. Two well-documented categories of solutions include logs (def. 1) and CRDT (def. 2).

A common minimum requirement for log synchronization across multiple peers is that the essential order of events is respected and/or can be determined [19], [20]. For these cases, logical clocks are a useful tool for eventual consistency and total ordering [21]. However, some scenarios (e.g. temporarily missing events or ambiguous order) can force a replica into a state that cannot be later resolved without costly recalculation. In specific cases, CRDTs can provide an alternative to log-based consensus (see sec. II-A5).

4) Logical Clocks: In a distributed system with multiple peers creating events, each with an independent clock, local timestamps can't be used to determine "global" event causality. Machine clocks are never perfectly synchronized [22], meaning that one peer's concept of "now" is not necessarily the same as another. Machine speed, network speed, and other factors compound the issue. For this reason, simple wall-clock time does not provide a sufficient notion of order in a distributed system. Alternatives to wall-clock time exist to help achieve eventual consistency. Examples include various logical clocks (Lamport [22] Schwartz [19], Bloom [23], and Hybrid variants [21], etc.), which use "counter"-based time-stamps to provide partial ordering.

Cryptographically linked events can also represent a clock (see sec. II-B3). One such example is called the Merkle-clock [24], which relies on properties of a Merkle-DAG (directed acyclic graph) to provide strict partial ordering between events. Like any logical clock, this approach does have its limitations [24, Sec. 4.3]:

> Merkle-Clocks represent a strict partial order of events. Not all events in the system can be compared and ordered. For example, when having multiple heads, the Merkle-Clock cannot say which of the events happened before.

5) Conflict-Free Replicated Data Types: CRDTs (def. 2) are one way to achieve strong eventual consistency, where once all replicas have received the same events, they will arrive at the same final state, *regardless of ordering*[2]. A review of the types of possible CRDTs is beyond the scope of this paper, however, it is important to note their role in eventually consistent systems and how they relate to clock-based event ordering. See for example [24], [25] for informative reviews of these types of data structures.

Whether a system or app uses a CRDT or a clock-based sequence of events is entirely dependent on the use-case and final data-model. While CRDTs may seem superior (and are currently a popular choice among decentralized systems), it is not possible to model every system as a CRDT. Additionally, the simplicity of clock-based sequencing often makes it easier to leverage in distributed systems where data conflicts will only rarely arise. Lastly, logs and CRDTs are not mutually exclusive and can be used together or as different stages of a larger system.

B. Content-based Addressing

Internet application architecture today is often designed as a system of clients (users) communicating to endpoints (hosts or servers). Communication between clients and endpoints usually happens via the TCP/IP protocol stack and depends on *location-based addressing*. Location-based addressing, where the client makes a request that is routed to a specific endpoint based on prior knowledge (e.g. the domain name or IP address), works relatively well for many use-cases. However, there are many reasons why addressing content by location is problematic, such as duplication

[2]Though non-commutative CRDTs may require a specific ordering of events in certain cases [24]

of storage, inefficient use of bandwidth, invalid/dead links (link rot), centralized control, and authentication issues.

An alternative to location addressing, called *content-based addressing*, may provide a solution to many of the problems associated with location-based addressing. Content-based addressing is where the content itself is used to create an address which is then used to retrieve said content from the network [26].

1) IPFS & Content-based Addressing: There are a number of systems that utilize content-based addressing to access content and information (e.g. Git, IPFS, Perkeep, Tahoe-LAFS, etc), and there is an active body of literature covering its design and implementation [27]–[29]. The Interplanetary File System (IPFS) — which is a set of protocols to create a content-addressed, peer-to-peer (P2P) filesystem — is one such system [27]. In IPFS, the address for any piece of content is determined based on a cryptographic hash of the content itself. In practice, an IPFS Content IDentifier (CID) is a multihash, which is a self-describing "protocol for differentiating outputs from various well-established cryptographic hash functions, addressing size [and] encoding considerations" [30]. That addressing system confers several benefits to the network, including tamper resistance (i.e. a given piece of content has not been modified en route if its hash matches what we were expecting), de-duplication (i.e. the same content from different peers will produce the same hash address), and data-corruption (again if a hash matches what we were expecting, we know it is complete). Additionally, IPFS content-based addresses are immutable and universally unique.

While content-based addressing doesn't dictate to a peer *how* to get a piece of content, IPFS (via libp2p[3]) does provide a system for moving content across the network. On the IPFS network, a client who wants specific content requests the CID from the network of IPFS hosts. The client's request is routed to the first host capable of fulfilling the request (i.e. the first host that is actively storing the content behind the given CID). The IPFS network can be seen as a distributed file system, with many of the benefits that come with this type of system design.

2) IPLD: As discussed previously, IPFS uses the cryptographic hash of a given piece of content to define its content-based address (see [27] for details on this process). However, in order to provide standards for accessing content-addressable data (on the web or elsewhere), it is necessary to define a common format or specification. In IPFS and other systems e.g. [31],

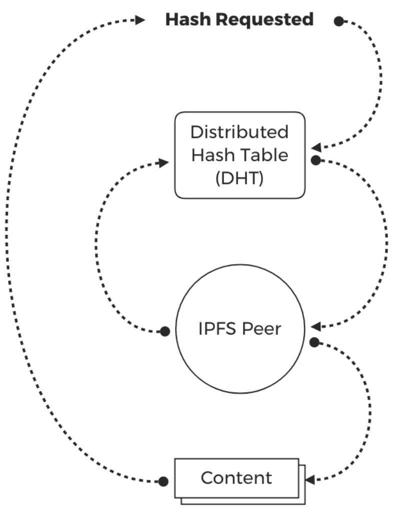

Figure 1: The cryptographic hash of content is used to make a request to the network of IPFS peers. Using the built-in routing mechanisms and a distributed hash table (DHT), peers hosting the requested content are identified and content is returned.

this common data format is called Interplanetary Linked Data (IPLD)[4]. As the name suggests, IPLD is based on principals of linked data [32], [33] with the added capabilities of a content-based addressing storage network.

IPLD is used to represent linked data that is spread across different "hosts," such that everything (e.g. entities, predicates, data sources) [34] uses content-based addresses as unique identifiers. To form its structure, IPLD implements a Merkle DAG, or directed acyclic graph[5]. This allows all hash-linked data structures to be treated using a unified data model, analogous to linked data in the Semantic Web sense [35]. In practice, IPLD is represented as objects, each with `Data` and (possibly multiple) `Link` fields, where `Data` can be a small blob of unstructured, arbitrary binary data, and a `Link` simply links to other IPLD objects.

3) Merkle-Clocks: A Merkle-Clock is a Merkle-DAG that represents a sequence of events, or a log [24]. When implemented on IPFS (or an equivalent network where content can be cryptographically addressed), Merkle-Clocks provide a number of benefits for data synchronization between replicas [24, Sec. 4.3]:

1. Sharing the Merkle-Clock can be done using only the *head*[6] CID. The whole Clock is unam-

[3]https://libp2p.io/

[4]https://ipld.io/

[5]Other examples of DAGs include the Bitcoin blockchain or a Git version history.

[6]In this context, "head" refers to the most recent event/update, and should not be confused with the "root" CID.

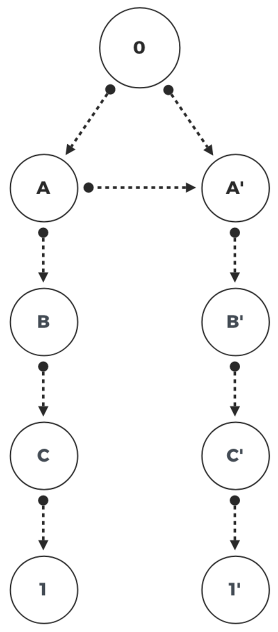

Figure 2: Divergent heads in a multi-writer Merkle-DAG

biguously identified by the CID of its head, and its full structure can be traversed as needed.

2. The immutable nature of a Merkle-DAG allows every replica to perform quick comparisons, and fetch only those nodes (leaves) that it is missing.
3. Merkle-DAG nodes are self-verified and immune to corruption and tampering. They can be fetched from any source willing to provide them, trusted or not.
4. Identical nodes are de-duplicated by design: there can only be one unique representation for every event.

However, since Merkle-clocks are logical clocks (see sec. II-A4), they cannot be used to order divergent heads representing concurrent events alone. For example, in fig. 2, two replicas (left and right columns) are attempting to write (top to bottom) events to the same Merkle-clock. After the first replica writes event A, the second writes event A' and properly links to A. At that point, perhaps the two replicas stop receiving events from one another. To a third replica (not pictured) that does continue to receive events, there would now be two independent heads, 1 and 1'. For the third replica, resolving these two logs of events may be costly (many updates happened since the last common node) or impossible (parts of the chain may not be available on the network).

In order to reduce the likelihood of divergent heads, all replicas should be perfectly connected and be able to fetch all events and linkages in the Merkle-clock. On real-world networks with many replicas that are often offline (mobile and Internet of Things (IoT) devices, laptops, etc.), these conditions are rarely met, making the use of a single Merkle-clock (or any logical clock) to synchronize replicas problematic.

C. Networking

So far we have primarily discussed the mechanics of creating or linking content in a series of updates. Now we will overview some common networking tools for connecting distributed peers who aim to maintain replicas of a shared state. This could be any decentralized network of interacting entities (e.g. cloud servers, IoT devices, botnets, sensor networks, mobile apps, etc) collectively updating a shared state. IPFS contains a collection of protocols and systems to help address the networking needs of different use-cases and devices — be it a phone, desktop computer, browser, or Internet-enabled appliance.

1) Libp2p: The libp2p project provides a robust protocol communication stack. IPFS and a growing list of other projects (e.g. Polkadot, Ethereum 2.0, Substrate, FileCoin, OpenBazzar, Keep, etc) are building on top of libp2p. Libp2p solves a number of challenges that are distinct to P2P networks. A comprehensive coverage of networking issues in P2P systems is out of the scope for this paper, however, some core challenges that libp2p helps to address include network address translator [36] (NAT) traversal, peer discovery and handshake protocols, and even encryption and transport security — libp2p supports both un-encrypted (e.g. TCP, UDP) and encrypted (e.g. TLS, Noise) protocols — among others. Libp2p uses the concept of a *multiaddress* to address peers on a network, which essentially models network addresses as arbitrary encapsulations of protocols [37]. In addition to “transport layer” modules, libp2p provides several tools for sharing and/or disseminating data over a P2P network.

2) Pubsub: One of the most commonly used P2P distribution layers built on libp2p is its Pubsub (or publish-subscribe) system. Pubsub is a standard messaging pattern where the publishers don't know who, if anyone, will subscribe to a given topic. *Publishers* send messages on a given topic or category, and *subscribers* receive only messages on a give topic to which they are subscribed. Libp2p's Pubsub module can be configured to utilize a *floodsub* protocol — which floods the network with messages, and peers are required to ignore messages in which they are not interested — or *gossipsub* — which is a proximity-aware epidemic Pubsub system, where peers communicate with proximal peers, and messages can be routed

Table I: Example Access Control List.

	Create	Delete	Edit	Read
Jane	-	-	-	✓
John	✓	✓	✓	✓
Mary	-	-	✓	✓

1) Agent-centric Security: Agent-centric security refers to the maintenance of data integrity without leveraging a central or blockchain-based consensus. The general approach is to let the reader enforce permissions and perform validations, not the writer or some central authority. Agent-centric security is possible if the reader can reference local-only, tamper-free code or if the local system state can be used to determine whether a given operation (e.g. delete operation) is permitted. Many decentralized networks, such as Secure Scuttlebutt [38] and Holochain [39], make use of agent-centric security. Each of these systems leverage cryptographic signatures to validate agent identities and messages.

2) Access control: All file-systems and databases have some notion of "access control." Many make use of an access-control list (ACL), which is a list of permissions attached to an object or group of objects [40]. An ACL determines which users or processes can access an object and whether a particular user or process with access can modify or delete an object (see tbl. I).

Using ACLs in systems where identity is derived from various configurations of PKI has been around for some time [41]. Still, many existing database and communication protocols built on IPFS to date lack support for an ACL or only have primitive ACL support. Where ACLs are missing, many systems use cryptographic primitives like signature schemes or enable encryption without any role-based configuration. Even more, many systems deploy an all-or-none security model, where those with access to a database have complete access, including write capabilities. Ideally, ACLs are mutable over time, and permission to modify an ACL should also be recorded in an ACL.

Event-driven systems (e.g. event sourcing) often make use of ACLs with some distinct properties. The ACL of an ES-based system is usually a list of access rules built from a series of events. For example, the two events, "grant Bob write access" and "revoke read access from Alice" would together result in a final ACL state where, Bob has read and write access, but Alice does not.

III. The Threads Protocol

We propose *Threads*, a protocol and decentralized database that runs on IPFS meant to help decouple apps from user-data. Inspired by event sourcing and object-based database abstractions, Threads is a protocol for creating and synchronizing state across collaborating peers on a network. Threads offer a multi-layered encryption and data access architecture that enables datasets with independent roles for writing, reading, and following changes. By extending on the multiaddress addressing scheme, Threads differs from previous solutions by allowing *pull*-based replica synchronization in addition to *push*-based synchronization that is common in distributed protocols. The flexible event-based structure enables client applications to derive advanced applications states, including queryable materialized views, and custom CRDTs.

A *Thread* is topic-based collection of single-writer logs. Taken together, these logs represent the current "state" of an object or dataset. The basic units of a Thread — Logs and Records — provide a framework for developers to create, store, and transmit data in a P2P distributed network. By structuring the underlying architecture in specific ways, this framework can be deployed to solve many of the problems discussed above.

Threads vs. Thread In short, *Threads* (plural) encompasses the whole system, including log orchestration and data stores built on those logs, whereas a *Thread* (sigular) is one or more logs grouped together by a topic or ID.

A. Record Logs

In multi-writer systems, conflicts arise as disparate peers end up producing disconnected state changes, or changes that end up out of sync. In order to proceed, there must be some way to deal with these conflicts. In some cases (e.g, `ipfs-log` [42]), a Merkle-clock can be used to induce ordering. Like all solutions based on logical clocks, this approach cannot achieve a total order of updates without implementing a data-layer conflict resolution strategy [24]:

> A total order can be useful . . . and could be obtained, for example, by considering concurrent events to be equal. Similarly, a strict total order could be built by sorting concurrent events by the CID or their nodes or by any other arbitrary user-defined strategy based on additional information attached to the clock nodes (data-layer conflict resolution).

Solutions such as `ipfs-log` include a built-in CRDT to manage conflicts not resolved by its logical clock. This approach works for many cases, but the use of a deterministic resolution strategy can be insufficient in cases with complicated data structures or complicated network topologies. A git merge highlights one such example, in which a predetermined merge strategy could be used, but is not often the best choice in practice. Furthermore, a multi-writer log using a linked data format (e.g. a Merkle-DAG) in imperfect networking or storage environments can lead to states where it is prohibitively difficult (e.g. due to networking, storage, and/or computational costs) to regain consistency.

A promising approach to dealing with this is to leverage the benefits of both a Merkle-clock for events from any one peer, and a less constrained ordering mechanism to combine events from all peers. In this case, developers can more freely institute their own CRDTs or domain-specific conflict resolution strategies. Additionally, it naturally supports use-cases where all peers contributing to a dataset may not be interested in replicating the events of all other peers (e.g. in a Pubsub-based system).

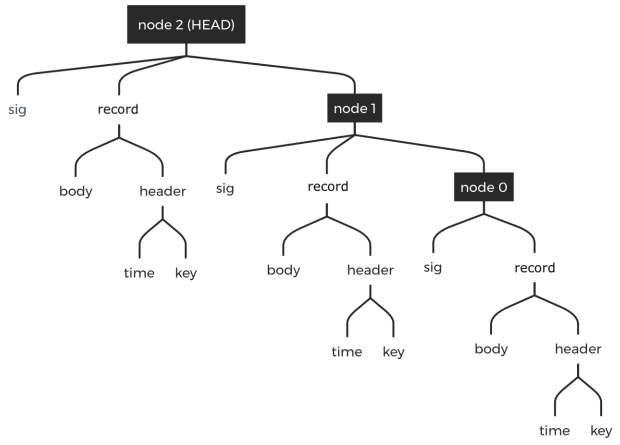

Figure 3: A single-writer Merkle-clock Log.

1) Single-writer Logs: Our solution to dealing with log conflicts (i.e. divergent Merkle-clocks) is to institute a *single-writer rule*: A Log can only be updated by a single replica or *identity*. A Log is then a *single-writer Merkle-clock* that can be totally ordered (fig. 3), and *separate* Logs can be composed into advanced structures, including CRDTs [25].

This concept is similar to ideas from within the Dat[7] community (among others). In particular, the use of multiple single-writer logs is akin to the Dat Multi-Writer proposal [43] (see also Hyperdb [44]), as well as concepts from Hypermerge[8] and Hypercore [45]. In the case of Hyperdb and Hypermerge, for example, the approach is to create a multi-writer system by combining *multiple* single-writer systems and some external or "out-of-band" messaging system to coordinate updates. In both cases, writers are able to resolve (or identify) conflicts using CRDTs (operation-based in Hyperdb and state-based in Hypermerge).

Threads provides a slightly different take on multi-writer systems [25], such that conflict resolution is *deferred* to a point at which a decision is actually required. This means that imperfect information may be supplemented along the way without causing conflicts in the mean time. It also means that apps can choose conflict resolution strategies specific to the task at hand. For example, if using a downstream CRDT, ordering is irrelevant and can be ignored completely. Alternatively, an additional clock may be required to ensure consistent ordering, such as a vector or Bloom clock (see sec. II-A4 and [43]). Finally, even manual merge-type strategies are possible if this is the desired conflict resolution strategy.

Writer	The single Peer capable of writing to a Log.
Reader	Any Peer capable of reading a Log. Practically speaking, this means any Peer with the Log's Read Key (sec. III-A3).

Together with a cryptographic signature, a Record is written to a log with an additional Payload (see fig. 3) enabling Log verification by Readers (sec. III-A3). At a minimum, a Record must link to its most immediate ancestor. However, links to older ancestors are often included as well to improve concurrency during traversal and verification [24], [46].

As shown in sec. VII-A, a Records's actual content (or body), is contained in a separate Block. This allows Records to carry any arbitrary Block structure, from complex directories to raw bytes.

2) Multi-addressed Logs: Much like IPFS peers, Logs are identified on the network with addresses, or more specifically, with multiaddresses [37]. Here we introduce *log* as a new protocol tag to be used when composing Log multiaddresses. To reach a Log via its multiaddress, it must be encapsulated in an IPFS peer multiaddress.

```
// Log ID encapsulated in peer multiaddress.
/ip4/127.0.0.1/tcp/4006/p2p/12D3KooEHd...
...PbbdRME/log/12D3KooasaDGS...iHKfiMf
```

Unlike peer multiaddresses, Log addresses are not stored in the global IPFS DHT [27]. Instead, they

[7]https://dat.foundation

[8]https://github.com/automerge/hypermerge

are *exchanged* with a push and pull mechanism (see sec. III-B4). This is in contrast to mutable data via IPNS for example, which requires querying the network (DHT) for updates. Updates are requested directly from the (presumably trusted) peers that produced them, resulting in a hybrid of content-addressed Records arranged over a data-feed[9] like topology. Log addresses are recorded in an address book (AddrBook), similar to an IPFS peer address book (see sec. III-A3). Addresses can also expire by specifying a time-to-live (TTL) value when adding or updating them in the address book, which allows for unresponsive addresses to eventually be removed.

Log addresses can also change over time, and these changes are again advertised to peers via the push and pull mechanism (see sec. III-B4). The receiving peers can then update their local AddrBook to reflect the new address(es) of the Log.

Modern, real-world networks consist of many mobile or otherwise sparsely connected computers (peers). Therefore, datasets distributed across such networks can be thought of as highly partitioned. To ensure updates are available between mostly offline or otherwise disconnected peers, Logs are designed with a built-in replication mechanism via *Replicas*. Replicas (or Replicators) are represented as additional addresses, meaning that a Log address book may contain *multiple* multiaddresses for a single Log.

Replica Log Writers can designate other IPFS Peers to "replicate" a Log, potentially republishing Records. A Replica is capable of receiving Log updates and traversing linkages via the Service Key (sec. III-A3), but is not able to read the Log's contents.

In practice, Writers are solely responsible for announcing their Log's addresses. This ensures a conflict-free address list without additional complexity. Some Services may be in the business of replicating Logs (sec. VI-A3), in which case Writers will announce the additional Log address to Readers. This allows them to *pull* (or subscribe to push-based) Records from the Service's Log address when the Writer is offline or unreachable (fig. 5).

3) Keys & Encryption: I

Logs are designed to be shared, composed, and layered into datasets (fig. 4). As such, they are encrypted by default in a manner that enables access control (sec. V-D2) and the Service mechanism discussed in the previous section. Much like the Log AddrBook, Log *keys* are stored in a KeyBook.

Identity Key Every Log requires an asymmetric key-pair that determines ownership and identity. The private key is used to sign each Record added to the Log, so down-stream processes can verify the Log's authenticity. Like IPFS Peers, a hash of the public key of the Log is used as an identifier (Log ID).

The body, or content of an Record, is encrypted by a *Content Key*. Content Keys are generated for each piece of content and never reused. The Content Key is distributed directly in the header of the Record.

Content Key The Content Key is a variable-format key used to encrypt the body (content) of an Record. This key can be symmetric, asymmetric, or possibly non-existent in cases where encryption is not needed.

One of two common encryption choices will typically be used for the Content Key of a Record:

1. When broadcasting Records to many possible recipients, a single-use symmetric key is generated per unique content body.
2. When sending Records to specific recipients, the recipient's public key can be used to restrict access from all others[10].

If a single-use symmetric key is used for the Content Key, it is necessary to distribute each new key to users by including it in the header of the Record. Therefore, the Record itself is further encrypted using a *Read* key. The Read Key is not distributed within the Log itself but via a separate (secure) channel to all Peers who require access to the content of the Log. Read Keys are scoped to a given *Thread*, so that the same key is used to encrypt all Records associated with a given Thread. This enables Readers to read Logs from new Writers.

Read Key The Read Key is a symmetric key created by the Thread initializer (creator) and used to encrypt the Content Key in each Record by all Thread participants.

Finally, the encrypted Record, its signature, and the IPLD linkage(s) from a Record to its antecedents are encrypted together using a *Service* Key. Service Keys allow Logs to be *transferred* to Peers on the network who do not have access to any content within the Record. Service providers can only see signatures and

[9]This is similar to the append-only message feeds used in Secure Scuttlebutt's global gossip network [38]

[10]Much like private messages in Secure Scuttlebutt.

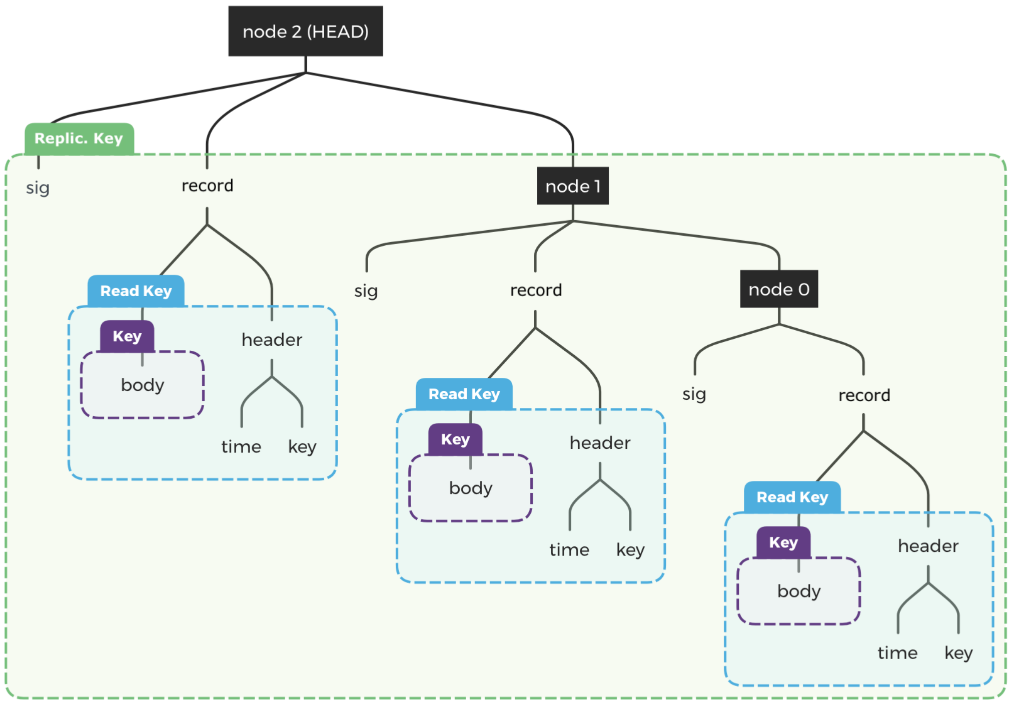

Figure 4: The three layers of Log encryption.

linkage(s) between Records. Again, Service Keys are scoped to the Thread, allowing all Thread participants to use the same key for Service access.

Service Key The Service Key is a symmetric key created by the Thread initializer (creator) and used to encrypt the entire Record before adding its payload to the Log.

B. *Threads*

A Thread is essentially an identifier that links a collection of Logs to a given topic. Threads are an event-sourced, distributed database, and can be used to maintain a single, collaboratively edited, replicated, or hosted dataset across multiple peers. Threads provide the mechanism to combine multiple Logs from individual Writers into singular shared states through the use of either cross-Log sequencing (e.g. using a Bloom clock, Merkle-clock, or hybrid logical clock [21]) or a CRDT (sec. II-A5).

1) Identity: A unique Thread Identity (TID) is used to group together Logs which compose a single dataset and as a topic identifier within Pubsub-based synchronization. The components of a TID are given in eq. 1.

$$\text{TID} = \underbrace{\texttt{0x62}}_{\text{Multibase}}\ \overbrace{\texttt{0x01}}^{\text{Version}}\ \underbrace{\texttt{0x55}}_{\text{Variant}}\ \overbrace{\texttt{0x539bc...a4546e}}^{\text{Random Component}} \tag{1}$$

TIDs share some similarities with UUIDs [47] (version and variant) and IPFS-based CIDs, and are multibase encoded[11] for maximum forward-compatibility. Base-32 encoding is used by default, but any multibase-supported string encoding may be used.

Multibase Prefix The encoding type used by the multibase encoder. 1 byte.

Version ID format version. 8 bytes max. This allows future version to be backwards-compatible.

Variant Used to specify thread-level expectations, like access-control. 8 bytes max. See sec. III-B2 for more about variants.

Random Component A random set of bytes of a user-specified length. 16 bytes or more.

2) Variants: Certain TID *variants* may be more appropriate than others in specific contexts. For example, Textile provides an *access-controlled* Thread variant, which supports various collaborative structures — e.g. social media feeds, shared documents, blogs, photo albums, etc.

[11] https://github.com/multiformats/multibase

Raw — This variant declares that consumers are not expected to make additional assumptions. This is the default variant (see (a) below).

Access-Controlled — This variant declares that consumers should assume that the random component of the TID is the CID of an immutable access control list. The ACL represents a permissions rule set that must be applied when reading data (sec. V-D2 and (b) below). Additionally, each ACL carries a reference to its previous version (if one exists), forming a *chain* of versions. Writers may choose to alter the ACL, which results in a fork.

```
// (a) Raw identity. V1, 128 bit
"bafkxd5bjgi6k4zivuoyxo4ua4mzyy"

// (b) ACL enabled identity. V1, 256 bit.
"bafyoiobghzefwl...gmdmgeobw2mimktr5jivsavya"
```

3) Multi-addressed Threads: Much like Logs, entire Threads are identified on the network with multiaddresses [37]. Here we introduce *thread* as an additional new protocol tag to be used when composing Thread multiaddresses. To reach a Thread via its multiaddress, it must also be encapsulated in an IPFS peer multiaddress.

```
// Thread ID encapsulated in peer multiaddress
/ip4/127.0.0.1/tcp/4006/p2p/12D3Koo...
...PbbdRME/thread/bafkxd5bjgi6k4zivuoyxo4ua4mzyy
```

Like Log multiaddresses, Thread addresses are not stored in the global IPFS DHT [27]. In practice, Threads are directly addressed only when creating links for viewing via a gateway or sharing. A Thread address serves Logs, which can be inflated via the push and pull mechanism outlined above.

4) Log Synchronization: Log Writers, Readers, and Service providers synchronize the state of their Logs by pushing and pulling Log keys and Records. Inspired by Git[12], a reference to the latest Record in a Log is referred to as the *Head*. When a new Record is received, Readers and Service providers simply advance their Head reference for the given Log. This is similar to how a system such as OrbitDB [42] works, except we are tracking *multiple* Heads (one per Log), rather than a single Head.

Thread Peers are able to inform each other of the latest Records and Log keys via a push and pull mechanism. The following represents the P2P network API for Thread orchestration:

```
service Threads {
  // GetLogs from a peer.
  rpc GetLogs(GetLogsReq) returns (GetLogsRep)
  // PushLog to a peer.
  rpc PushLog(PushLogReq) returns (PushLogRep)
  // GetRecords from a peer.
  rpc GetRecs(GetRecsReq) returns (GetRecsRep)
  // PushRecord to a peer.
  rpc PushRec(PushRecReq) returns (PushRecRep)
}
```

[12]https://git-scm.com/

a) Push: Pushing Records is performed in multiple phases because, invariably, some Thread participants will be offline or unresponsive:

1. New Records are pushed[13] directly to the Thread's other Log Writers.
2. New Records are pushed directly to the target Log's Service provider(s), who may not maintain their own Log.
3. New Records are published over gossip-based Pubsub using TID as a topic, which provides potentially unknown Readers or Services with an opportunity to consume Records in real-time.

Step 2 above allows for *additional* push mechanisms, as Services with public IP addresses can become relays:

1. New Records may be pushed directly to web-based participants over a WebSocket.
2. New Records may be pushed to the Thread's other Log Writers via federated notification services like Apple Push Notification Service (APNS), Google Cloud Messaging (GCM), Firebase Cloud Messaging (FCM), and/or Windows Notification Service (WNS).
3. New Records may trigger web-hooks, which could enable many complex (e.g. IFTTT[14]) workflows.

b) Pull: Pulling Records helps to maximize connectivity between peers who are often offline or unreachable.

1. Records can be pulled from Services via HTTP, RSS, Atom, etc.
    1. In conjunction with push over WebSockets (seen in Step 2 of the additional push mechanisms above), this method provides web-based Readers and Services with a reliable mechanism for receiving Log Records (fig. 5).

5) Log Replication: The notion of the Service Key (sec. III-A3) makes duplicating all Log Records trivial. This allows any peer on the network to be granted the responsibility of replicating data from another Peer

[13]Here push means "send to multiaddress(es)," which may designate different protocols, e.g. P2P, HTTP, Bluetooth, etc.
[14]https://ifttt.com

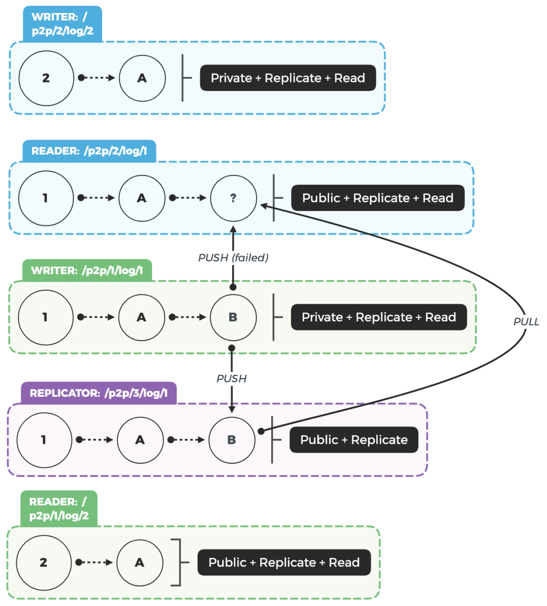

Figure 5: A pull-based request from a Service.

without having read access to the raw Log Records. This type of Log replication can act as a data backup mechanism. It can also be used to build services that react to Log Records, potentially pushing data to disparate, non-Textile systems, especially if the replication service *is* granted read access to the Log Records (sec. III-B4).

IV. Threads Internals

Previous sections have discussed the high-level concepts of the Threads protocol. However, we have not yet discussed the internals of managing Log Records. In this section, we provide a description of the Threads reference implementation. The Threads *Store* outlined here takes advantage of ideas from several existing CQRS and ES systems (e.g. [48]), as well as concepts and designs from Flux [49], Redux[15] [50] and domain driven design [51] (DDD)[16]. Following this discussion of the Threads Store, in sec. V we outline how it can be used to build intuitive developer-facing application programing interfaces (APIs) to make adopting and using Threads "the right choice" for a wide range of developers.

A. Overview

Application state management tools such as Redux provide opinionated frameworks for making state mutations *predictable* by imposing certain restrictions on how and when updates can happen. The architecture of such an application shares several core features with CQRS and ES systems (see [49] and/or [52]). In both cases, the focus is on unidirectional data flows that build downstream views from atomic updates in the form of events (or actions).

I

We adopt a similar flow in Threads (see fig. 6). Like any CQRS/ES-based system (see also sec. II-A1), Threads are built on *Events* ([13]). Events are used to produce *predictable* updates to downstream state. Similarly to a DDD-based pattern, to add an Event to Threads, we use *Collections*, which are used to create and send *Actions* via an *Event Codec* (projection in CQRS terminology), to a *Dispatcher*. The Dispatcher then notifies downstream *Reducers* of new Events, which are then used to mutate the Store's state; all within a single *Transaction*. This unidirectional, transaction-based system provides a flexible framework for building complex event-driven application logic.

Fig. 6 depicts the various components that make up a Store. New Events travel through the system once a Transaction is committed by a local Actor (via the User API). Conversely, new Events caused by updates in an *external* peer's Log (associated with the given Thread), go through essentially the same process, but in reverse. The various core components are discussed in detail in the following sections.

[15]Redux builds on concepts from CQRS and ES itself, and is arguably an implementation of the Flux application architecture.

[16]https://dddcommunity.org

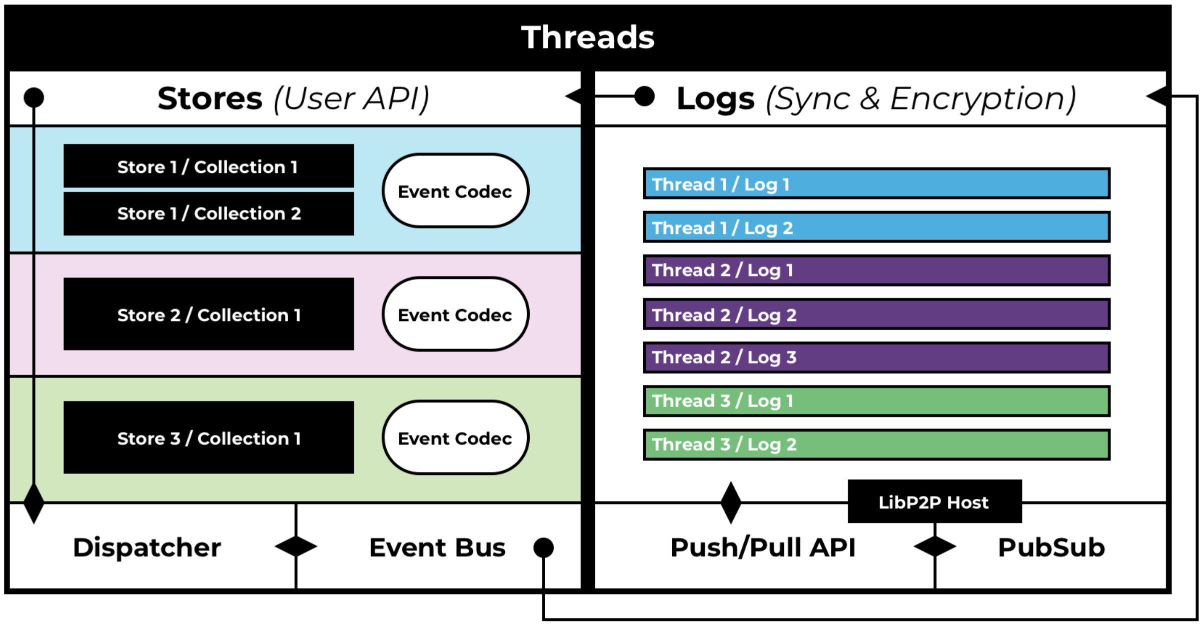

Figure 6: Components diagram for internal Store implementation.

1) Collections: As in many CQRS-based systems, Events are dispatched on the write side, and are *reacted to* on the read side. The primary interface to the write side is exposed via a Store's Collection(s). As shown in fig. 6, Collections are a core component of the Threads Store. They are used to send *Actions* from a local application to the rest of the internal Store. If built around a specific *domain*, Collections provide bounded context that can be roughly compared to an aggregate root in DDD [51].

Collections Collections are part of a Store's public-api. Their main responsibilities are to store instances of user-defined Schemas, and operate on Instances defined by said Schema.

Actions Every update to local and shared (i.e. across peers) state happens via Actions. Actions are used to describe *changes to an application state* (e.g. a photo was added, an item was added to a shopping cart, etc). Collections are used to create Actions.

Currently, Collections are defined using a JSON Schema [53] that *describes the shape* of the underlying Instance that it represents. This is also quite similar to a document in a document-based database context. For example, a Collection might define a `Person` Instance, with a `first` and `last` name, `age`, etc. Collections also provide the public API (bounded context) for creating, deleting, updating, and querying these Instances. Lastly, they also provide read/write *Transactions* which have *serializable isolation* within the entire Store scope. In other words, Transactions can be assumed to be the only running operation on the entire Store. It is via Transactions that Collections introduce Actions into the system.

Transactions Actions describing updates to the system happen within Transactions [54] in order to ensure consistency of the local Store. It is only after a Transaction has been committed that its Actions are sent to the Event Codec in order to be translated into Events.

2) Event Codec: Before Actions from the client-side are introduced into the system, they are encoded as Events using the Event Codec. Here, the core function is to transform (i.e. encode/decode) and apply Transaction Actions. An Event Codec is therefore an internal abstraction layer used to:

1. Transform Actions made in a write Transaction into an (array of) Events that will be dispatched via the Dispatcher to be "reduced."
2. Encode Actions made in a Transaction in an IPLD Node, which will serve as a building block for an event Record entry in the local Peer's Datastore.
3. Decode external IPLD Nodes into Events, which can then be dispatched locally.

For example, if within a Collection's (write) Transaction, a new Instance is created and another one is updated, these two Actions will be sent to the Event Codec to transform them into Events. These Events have a Payload of bytes with the encoded transformation(s). Currently, the only implementation of Event Codec is a *JSON Patcher*, which transforms Actions into JSON-merge/patch objects [55], [56]. The Event Codec is essentially a "projection" in ES + CQRS terminology.

Instance An Instance is made up of a series of ordered Events referring to a specific object. An Instance might have a unique `UUID` which can be referenced across Event updates.

Once these Events have been aggregated by the Event Codec into a single IPLD Node, this information is used by the Thread Service to actually persist the Event under the Thread associated with the given Store (i.e. it is written to an underlying Datastore). Likewise, the Event Codec can also do the inverse transformation: given an IPLD Node, it transforms its Payload into Actions that will be reduced in the Store.

To summarize, while a Transaction is running and making changes, the individual Actions are accumulated. Once the Transaction is committed, two things happen:

1. Each Action is transformed to an Event
2. The list of Actions is transformed to a single Payload

In the above, the list of Events are sent to the Dispatcher (for storage, and broadcasting to downstream Reducers), and the Event Payload is sent to the Thread Service, where it is added to the local Peer's Log.

The Event Codec abstraction provides a useful extensibility mechanism for the Store. For instance, eventually consistent, CRDT-based structures are particularly useful for managing views of a document in a multi-peer collaborative editing environment (like Google Docs or similar). To support this type of offline-first use-case (where peers may be making concurrent edits on a shared JSON document), one could implement an Event Codec that supports a JSON CRDT datatype [57] (or even a hybrid JSON Patch [56] with logical clocks). Here updates to a JSON document would be modeled as CRDT operations (ops) *or* deltas to provide CRDT-like behavior to downstream Reducers. Libraries such as Automerge[17] provide useful examples of reducer functions that make working with JSON CRDTs relatively straightforward.

[17]https://github.com/automerge/automerge

3) Dispatcher: In order to persist and dispatch Events to downstream consumers, a *Dispatcher* is used. Every Event generated in the Store is sent to a Dispatcher when write Transactions are committed. The Dispatcher is then responsible for broadcasting these events to all registered *Reducers*. For example, if a particular Instance is updated via an Action, this will be encoded as an Event by the Event Codec (as mentioned previously). These Events will then be dispatched via the Dispatcher, which will:

1. Store the new Event in persistent storage (*Datastore*). If the Transaction made *multiple* changes, this is done *transactionally*.
2. Broadcast all *new* Events to all registered Reducers. Reducers will then apply the changes encoded by the Event Codec as they "see fit."

This design implies that Store state changes *can only happen when the Dispatcher broadcasts new Events.* A Reducer can't distinguish between Events generated locally or externally. External events are the results of the Thread Service sending new Events to the Dispatcher, which means that new Events where detected in another peer's Log from the same Thread. In the reverse case, local Events are sent to the network via an *Event Bus*, to be handled by the *Log Service.*

Dispatcher All Events must go through the singleton Dispatcher, whether these initiated as local or remote Events. The Dispatcher is responsible for informing registered Reducers of new Events.

Reducer A Reducer is a party which is interested in knowing about Store Events. Currently, the only Reducer is the Store itself.

Datastore A Datastore is the underlying persistence layer for Collections/Instances and a Dispatcher's raw Event information. the Datastore is updated via a Transaction to have transactional guarantees.

Event Bus The Event Bus is responsible for delivering local encoded Events to the Log Service for distribution to external peers (i.e. the network).

4) Store Listener: After an Event (or set of Events) has been dispatched by the Dispatcher, it is necessary to notify various "external" actors that the Store has changed its state. Details of the change might include in which model the change occurred, what Action(s) (`Create`, `Save`, `Delete`, etc) were handled, and which specify Instance (`InstanceIDs`) were modified. These external actors are called *Listeners*, and are useful for clients that want to be notified about changes in the Store.

Store Listener A Listener is a “client” or external actor that would like to subscribe to Store updates based on a set of conditions.

5) Log Service: The Log Service is the primary networking interface for Threads. It is part of the public developer API, so it can be accessed by external components. Its main responsibility is to provide an interface between the Store and the broader network. Its APIs can be used to perform sync and encryption/decryption tasks. It is also responsible for storing transactions in the local Peer’s Log, and when it detects new Records in *another* Peer’s Logs, it will dispatch them via the Dispatcher allowing the local Store to handle the external Event and take appropriate action. The various APIs (Push/Pull) and libp2p functions of this component are discussed in part in sec. III-B4.

Log Service The Log Service is the bidirectional communication interface for Threads.

V. The Store Interface

To make Threads as easy to adopt and use as possible, we have designed a developer facing API on top of the Threads internals that simplifies dealing with events and data, while still maintaining the power and flexibility of CQRS and ES. Developers should not have to learn a whole new set of terms and tools to take advantage of Threads’ capabilities. This simple, public-facing API will be comfortable to application developers looking to leverage distributed systems that leverage user-siloed data, with minimal configuration and maximum interoperability. Inspired by tools such as MondoDB[18], Parse[19], and Realm[20], as well as the idea of bounded context and aggregate roots from DDD, we provide simple abstractions for performing distributed database operations as if they were local.

Indeed, the components of Threads already provide several features that you would expect when operating on a dataset or table within a database: each Thread is defined by a unique ID, and provides facilities for access control and permissions, networking, and more. To illustrate how these underlying components can be combined to produce a simple API with minimal configuration and boilerplate, consider the following example in which we provide working Javascript code for a hypothetical User-management app.

A. Illustrative Example

To create a useful application, developers start with a `Store`. Generally, a Store is accessed via a `Client` instance. This could be a local, or remote Threads provider.

```
const client = new Client(...opts)
```

A default (or empty) `Store` can be created directly from the `Client`. This would create a new Thread under-the-hood, with an empty `Store` to be populated by Actions generated from a `Collection` (next step).

```
const store = await client.newStore()
```

To interact with the `Store`, a developer must first create a new `Collection`. A `Collection` is essentially the public API for the Store (see sec. IV-A1). For example a developer might create a new `Collection` to represent `Person` information. `Collections` are defined by their `Schema`, which is a JSON Schema [53] object the defines and is used to validate `Instances` created within the `Collection`. In practice, there will be many pre-defined `Schemas` that developers can use to make their applications interoperable with others. See sec. V-C for some initial plans in this regard.

```
// Interface corresponds to JSON Schema
interface Person {
  _id: string // Instance have `_id` by default
  firstName: string
  lastName: string
  age: number
}

// JSON Schema defining above interface
const personSchema = {
  $id: 'https://example.com/person.schema.json',
  $schema: 'http://json-schema.org/draft-07/schema#',
  title: 'Person',
  type: 'object',
  required: ['ID'],
  properties: {
    _id: {
      type: 'string',
      description: "Instance ID.",
    },
    firstName: {
      type: 'string',
      description: "First name.",
    },
    lastName: {
      type: 'string',
```

[18]http://www.mongodb.com
[19]https://parseplatform.org
[20]https://realm.io

```
      description: "Last name.",
    },
    age: {
      description: 'Age in years.',
      type: 'integer',
      minimum: 0,
    },
  },
}
```

With a Schema in hand, the developer is able to register it with the Store, and then generate Collections based on said Schema. The Schema is added to the system at the moment a Collection is created or joined by external link.

```
// Register a Schema during Collection creation
await client.newCollection(
  store.id, 'Person', personSchema
)
```

At this point, the developer is looking at an empty Store/Thread. To add data, the developer can create a new instance in a Collection (an Instance), and then modify this Instance and save the changes (via Actions). These operations are all done automatically by the Store, under-the-hood.

```
// Initial Person object
const person: Person = {
  _id: '',
  firstName: 'John',
  lastName: 'Doe',
  age: 21,
}

// Create a Collection Instance
const created = await client.create(
  store.id, 'Person', person
)
console.log(created) // Instance List
// [<uuid>]
```

Specific Instances can be modified (or deleted) via Actions, which are created by the Collection instance (Instance):

```
// Record the _id and update the age
person._id = created[0]
person.age = 26

// Modify/mutate existing Instance
await client.save(
  store.id, 'Person', [person]
)

// Delete existing Instance
await client.delete(
  store.id, 'Person', [person._id]
)

// Check whether Instance exists
const has = await client.has(
  store.id, 'Person', [person._id]
)
console.log(has) // false
```

Like any useful database, the Textile Store interface exposed here also provides mechanisms for search, monitoring (subscriptions), and database transactions. Each of these features interact with the various components of the underlying Event Store. For example, Transactions are a core feature of the Event Store (see sec. IV-A1 and sec. IV-A3), and subscriptions provide a Store Listener (sec. IV-A4) interface. Transactions work as in most database implementations:

```
const txn = client.writeTransaction(
  store.id, 'Person'
)
await transaction.start()

const person = {
  _id: '',
  firstName: 'John',
  lastName: 'Doe',
  age: 30,
}
const created = await transaction.create([person])
const existing = created.pop()
person._id = existing

const has = await transaction.has([existing])
console.log(has) // true
person.age = 99
await transaction.save([person])
await transaction.delete([person._id])
await transaction.end()
```

Similarly, subscriptions can be used to monitor updates, right down to individual Instances:

```
const closer = client.listen(
  store.id, 'Person', person._id, reply => {
  console.log(JSON.stringify(reply.instance))
  // Instance modified: { ..., age: 30 }
  // Instance modified: { ..., age: 40 }
})

person.age = 30
await client.save(
  store.id, 'Person', [person]
)
```

```
person.age = 40
await client.collectionSave(
  store.id, 'Person', [person]
)

// Find or search for a specific Instance
const found = await client.findByID(
  store.id, 'Person', person._id
)
console.log(found.person.age) // 40
```

Lastly, search is performed via queries to the underlying Instances. For example, to search for all `Persons` matching a given `age` range, a developer can implement the following Query on the Store.

```
// Select all Persons...
const query = Query.where('age') // where `age`...
  .ge(60) // is greater than or equal to 60, ...
  .and('age') // and `age`...
  .lt(66) // is less than 66, ...
  // or where `age` is equal to 67
  .or(new Where('age').eq(67))
const res = await client.find(
  store.id, 'Person', query
)
console.log(res.instancesList.length)
```

Queries can be arbitrarily complex, and when possible, will take advantage of indexes and other features of the underlying database implementation. Details of the underlying Store database are not part of the Threads specification, so clients are welcome to optimize Store implementations for specific use-cases and design considerations. For instance, see sec. V-B for some possible optimizations/alternative interfaces to the Threads Store.

B. Alternative Interfaces

While the above Store interface provides a intuitive implementation of Threads, it is not difficult to imagine alternative, high-level APIs in which Threads are exposed via interfaces compatible with *existing* datastores or DBMS. Here we draw inspiration from similar projects (e.g. OrbitDB [42]) which have made it much easier for developers familiar with centralized database systems to make the move to decentralized systems such as Threads. For example, a key-value store built on Threads would “map” key-value operations, such as `put`, `get`, and `del` to internal Actions. The Events derived from said Actions would then be used to mutate the Store like any Instance, effectively encapsulating the entire Store in a database structure that satisfies a key-value store interface. These additional interfaces would also be distributed as Modules (see sec. V-C), making it easy for developers to swap in or substitute existing backend infrastructure.

Similar abstractions will be used to implement additional database types and functions. Tables, feeds, counters, and other simple stores can also be built on Threads. Each database style would be implemented as a standalone software library, allowing application developers to pick and choose the solution most useful to the application at hand. Similarly, more advanced applications could be implemented using a combination of database types, or by examining the source code of these *reference* libraries.

C. Modules

One of Textile’s stated goals is to allow individuals to better capture the value of their data while still enabling developers to build new interfaces and experiences on top of said data. A key piece of this goal is to provide *inter-application* data access to the *same underlying user-siloed data*. In order to meet this goal, it is necessary for developers to be using the same data structure and conventions when building their apps. In conjunction with community developers, Textile will provide a number of *Modules* designed to wrap a given domain (e.g. Photos) into a singular software package to facilitate this. This way, developers need only agree on the given `Schema` in order to provide seamless inter-application experiences. For example, any developer looking to provide a view on top of a user’s Photos (perhaps their phone’s camera roll) may utilize a `Photos` Module. Similarly, a `Person` module might be designed as in the example from the previous section. Developers may also extend a given Module to provide additional functionality.

In building on top of an existing Module, developers ensure other application developers are also able to interact with the data produced by their app. This enables tighter coupling between applications, and it allows for smaller apps that can focus on a very specific user experience (say, filters on Photos). Furthermore, it provides a *logically centralized*, platform-like developer experience, without the actual centralized infrastructure. APIs for Photos, Messages, Chat, Music, Video, Storage, etc are all possible, extensible, and available to all developers. This is a powerful concept, but it is also flexible. For application developers working on very specific or niche apps with less need for inter-application usability, Modules are not needed, and they can instead focus on custom Collections. However, it is likely that developers who build on openly available *standard* Modules will provide a more useful experience for their users, and will benefit from the *network effects* [58] produced by many interoperable apps.

D. Thread Extensions

The Threads protocol provides a distributed framework for building shared, offline first, Stores that are fault tolerant[21], eventually consistent, and scalable. Any internal implementation details of a compliant Threads *client* may use any number of well-established design patterns from the CQRS and ES (and related) literature to *extend* the Threads protocol with additional features and controls. Indeed, by designing our system around Events, a Dispatcher, and generic Stores, we make it easy to extend Threads in many different ways. Some extensions included by *Textile's* Threads implementation are outlined in this section to provide some understanding of the extensibility this design affords.

1) Snapshots and Compaction: Snapshots[22] are simply the current state of a Store at a given point in time. They can be used to rebuild the state of a view Store without having to query and re-play all previous Events. When a Snapshot is available, a Thread Peer can rebuild the state of a given view Store/Model by replaying only Events generated since the latest Snapshot. Snapshots could be written to their own internal Store and stored locally. They can potentially be synced (sec. III-B4) to other Peers as a form of data backup or to optimize state initialization when a new Peer starts participating in a shared Thread (saving disk space, bandwidth, and time). They can similarly be used for initializing a local view Store during recovery.

Compaction is a *local-only operation* (i.e. other Peers do not need to be aware that Compaction was performed) performed on an Store to free up local disk space. As a result, it can speed up re-hydration of a downstream Stores's state by reducing the number of Events that need to be processed. Compaction is useful when only the latest Event of a given type is required.

2) Access Control: One of the most important properties of a shared data model is the ability to apply access control rules. There are three forms of access control possible in Threads, Instance-level ACLs, Collection-level ACLs, and Thread-level ACLs. Thread-level access control lists (ACLs) allow creators to specify who can *replicate, read, write, and delete* Thread data. Similarly, Collection and Instance-level ACLs provide more granular control to Thread-writers on a per-Collection and Instance (see def. 4) basis.

[21]When using an ACID compliant backing store for example.

[22]The literature around snapshots and other CQRS and ES terms is somewhat confusing, we attempt to use the most common definitions here.

Textile's Threads includes ACL management tooling based on a *Role-based access control* [59] pattern, wherein individuals or groups are assigned roles which carry specific permissions. Roles can be added and removed as needed. Textile ACLs can make use of five distinct roles[23]: *No-access, Replicate, Read, Write, and Delete.*

No-access	No access is permitted. This is the default role.
Service	Access to Log Service Keys is permitted. Members of this role are able to verify Events and follow linkages. The Service role is used to designate a "Service" Peer for offline replication and/or backup.
Read	Access to Log Read Keys is permitted in addition to Service Keys. Members of this role are able to read Log Event payloads.
Write	Members of this role are able to author new Events, which also implies access to Log Service and Read Keys. At the Thread-level, this means authoring a Log. At the Model-level, this means authoring a specific Model. At the Instance-level, the Write role means that Events in this Log are able to target a particular Instance.
Delete	Members of this role are able to delete Events, which implies access to Log Service Keys. In practice, this means creating a new "Tombstone" Event which marks an older Event as "deleted." See sec. V-D2b for additional notes on deleting data.

Below is a typical Thread-level ACL:

```
{
  "ID": "bafykrq5i25vd64ghamtgus6lue74k",
  "default": "no-access",
  "peers": {
    "12D..dwaA6Qe": ["write", "delete"],
    "12D..dJT6nXY": ["service"],
    "12D..P2c6ifo": ["read"],
  }
}
```

The `default` key states the default role for all network Peers. The `peers` map is where roles are delegated to specific Peers. Here, `12D..dwaA6Qe` is likely the owner, `12D..dJT6nXY` is a designated Service, and `12D..P2c6ifo` has been given read access.

[23]By default, Threads without access control operate similar to Secure Scuttlebutt (SSB; where every Peer consumes what they want and writes what they want).

a) Note About Threats: The design of threads is suited to two broad use-cases. The first is for datasets and documents where one or a few verified owners have the right to update those documents, and others (possibly many) can read those updates. The second is for closed networks of collaborating peers, so peers are generally assumed to be using a "compliant" Thread implementation. In the latter case, each Thread Peer is expected to enforce its own ACL rules via agent-centric security practices. For example, if a Peer A, who has ACL write access, updates the ACL, all Thread participants who received the said update are expected to enforce said ACL then locally. If Thread updates come out of order, it is up to the receiving Peers to re-process the updates to take into account new ACL updates.

b) Note About Deleting: Deleting data in distributed systems is a complex concept. In practice, it is impossible to ensure all Peers in a system will comply with any given Tombstone Event. Often, data (i.e. Blocks, Events, etc.) are kept locally, including *original and tombstone* Events, to facilitate parsing of the Datastore. This means raw data that have been "deleted" are not immediately purged from a Peer's storage. However, in strict data compliance situations (e.g. the EU's GDPR), a deletion Event *may* additionally generate a Snapshot Event, allowing past data to be purged from the local Event and Block stores. Compliant Peers should then purge "deleted" data. However, the possibility of non-compliant data caching remains. The initial Textile Threads reference implementation will *not* automatically purge deleted data from the local store. This compliance requirement will initially be left up to application developers.

VI. Conclusion

In this paper, we described the challenges and considerations when creating a protocol suitable for large-scale data storage, synchronization, and use in a distributed system. We identified six requirements for enabling *user-siloed* data: flexible data formats, efficient synchronization, conflict resolution, access-control, scalable storage, and network communication. We have introduced a solution to these requirements that extends on IPFS and prior research conducted by ourselves and others, which we term Threads. Threads are a novel data architecture that builds upon a collection of protocols to deliver a scalable and robust storage system for end-user data.

We show that the flexible core structure of single-writer append-only logs can be used to compose higher-order structures such as Threads, Views, and/or CRDTs. In particular, we show that through the design of specific default Collections, we can support important features such as access control lists and common, specialized, or complex data models. The Threads protocol described here is flexible enough to derive numerous specific database types (e.g. key/value stores, document stores, relational stores, etc) and model an unlimited number of applications states. The cryptography used throughout Threads will help shift the data ownership model from apps to users.

A. Future Work

The research and development of Threads has highlighted several additional areas of work that would lead to increased benefits for users and developers. In particular, we have highlighted network services and security enhancements as core future work. In the following two sections, we briefly outline planned future work in these critical areas.

1) Enhanced Log Security: The use of a single Read and Service Key for an entire Log means that, should either of these keys be leaked via malicious (or other possibly accidental) means, there is no way to prevent a Peer with the leaked keys from listening to Events or traversing the Log history. Potential solutions currently being explored by Textile developers include key rotation at specific Event offsets [60], and/or incorporating the Double Ratchet Algorithm [61] for forward secrecy [62].

2) Tighter Coupling with Front End Models: Implementing Threads internals (see sec. IV) using similar patterns to common frontend workflows (e.g. Redux) presents opportunities for tighter coupling between "backend" logic and frontend views. This is a major advantage of tools such as reSolve[24], where "system changes can be reflected immediately [on the frontend], without the need to re-query the backend" [48]. Textile developers will create frameworks to more directly expose the internals of Threads to frontend SDKs (or DBMS, see sec. VII-B), making it possible to sync application state across and between apps and services on the IPFS network.

3) Textile: The Thread & Bot Network: Threads change the relationship between a user, their data, and the services they connect with that data. The nested, or multi-layered, encryption combined with powerful ACL capabilities create new opportunities to build distributed services, or bots, in the network of IPFS Peers. Based on the Service Key now available in Threads, Services such as the Textile Hub can

[24]https://reimagined.github.io/resolve/

be built to help relay, replicate, and store data for database users.

This will be a powerful framework to build Services such as bots that is synchronize data via real-time updates in a *trust-less*, partially trusted, or fully-trusted way. Bots can additionally enhance the IPFS network by providing a framework to build and deploy many new kinds of services available over HTTP or P2P. Services could include simple data archival, caching and republishing, translation, data conversion, and more. Advanced examples could include payment, re-encryption, or bridges to Web 2.0 services to offer decentralized access to Web 2.0.

VII. Appendix

A. Records, Events, and Blocks

```
// Record is most basic component of log.
type Record interface {
  format.Node

  // BlockID returns cid of event block.
  BlockID() cid.Cid

  // GetBlock loads the event block.
  GetBlock(
    context.Context, format.DAGService
  ) (format.Node, error)

  // PrevID returns cid of previous record.
  PrevID() cid.Cid

  // Sig returns record signature.
  Sig() []byte

  // Verify checks for valid record signature.
  Verify(pk ic.PubKey) error
}

// Event is the Block format used by Threads.
type Event interface {
  format.Node

  // HeaderID returns cid of event header.
  HeaderID() cid.Cid

  // GetHeader loads and decrypts event header.
  GetHeader(
    context.Context,
    format.DAGService,
    crypto.DecryptionKey
  ) (EventHeader, error)

  // BodyID returns cid of event body.
  BodyID() cid.Cid

  // GetBody loads and decrypts event body.
  GetBody(
    context.Context,
    format.DAGService,
    crypto.DecryptionKey
  ) (format.Node, error)
}

// EventHeader is format of event's header.
type EventHeader interface {
  format.Node

  // Time returns wall-clock time + sequence id
  // from when event was created.
  // Can be used to derive Hybrid Logical Clock
  Time() (*time.Time, int32, error)

  // Key returns single-use decryption key.
  Key() (crypto.DecryptionKey, error)
}
```

B. Database Management Systems (DBMS)

1. Relational
   1. MariaDB https://mariadb.org/
   2. PostgreSQL https://www.postgresql.org/
   3. SQLite https://www.sqlite.org
2. Key-value stores
   1. Dynamo https://aws.amazon.com/dynamodb/
   2. LevelDB https://github.com/google/leveldb
   3. Redis https://redis.io/
3. Document stores
   1. CouchDB http://couchdb.apache.org/
   2. MongoDB https://www.mongodb.com/
   3. RethinkDB https://rethinkdb.com/

VIII. Acknowledgements

The authors would like to acknowledge the contributions of various community members who contributed to early versions of this whitepaper. The work presented here has been greatly improved by their comments, suggestions, and revisions. In particular, the authors thank Hector Sanjuan, Adin Schmahmann, Brendan O'Brien, Nicklas Gummesson, Cody Hatfield, and Aaron Steele.

References

[1] C. Dixon, "Why Decentralization Matters," *Medium*, Oct. 26, 2018. https://medium.com/s/story/why-decentralization-matters-5e3f79f7638e (accessed Sep. 19, 2019).

[2] T. Berners-Lee and K. O'Hara, "The read–write Linked Data Web," *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 371, no. 1987, p. 20120513, Mar. 2013, doi: 10.1098/rsta.2012.0513.

[3] Y.-A. de Montjoye, S. S. Wang, A. Pentland, D. T. T. Anh, and A. Datta, "On the Trusted Use of Large-Scale Personal Data." *IEEE Data Eng. Bull.*, vol. 35, no. 4, pp. 5–8, 2012.

[4] A. V. Sambra *et al.*, *Solid: A Platform for Decentralized Social Applications Based on Linked Data*. 2016.

[5] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990.

[6] Jay Kreps, "The Log: What every software engineer should know about real-time data's unifying abstraction," Dec. 16, 2013. https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying (accessed Sep. 19, 2019).

[7] D. Betts, J. Dominguez, G. Melnik, F. Simonazzi, and M. Subramanian, *Exploring CQRS and Event Sourcing: A Journey into High Scalability, Availability, and Maintainability with Windows Azure*, 1st ed. Microsoft patterns & practices, 2013.

[8] "Event Store." https://eventstore.org/ (accessed Sep. 19, 2019).

[9] "Apache Kafka." https://kafka.apache.org/ (accessed Sep. 19, 2019).

[10] "Apache Samza." http://samza.apache.org/ (accessed Sep. 19, 2019).

[11] M. Kleppmann, *Designing data-intensive applications: The big ideas behind reliable, scalable, and maintainable systems.* " O'Reilly Media, Inc.", 2017.

[12] Martin Fowler, "CQRS," *martinfowler.com*, Jul. 14, 2011. https://martinfowler.com/bliki/CQRS.html (accessed Sep. 20, 2019).

[13] M. Fowler, "Event Sourcing," *martinfowler.com*. https://martinfowler.com/eaaDev/EventSourcing.html (accessed Sep. 19, 2019).

[14] Microsoft Corporation, "Azure Application Architecture Guide," *Microsoft Docs*. https://docs.microsoft.com/en-us/azure/architecture/guide/ (accessed Sep. 19, 2019).

[15] E. Brewer, "Towards Robust Distributed Systems," presented at the Invited Talk, 2000, [Online]. Available: http://pld.cs.luc.edu/courses/353/spr11/notes/brewer_keynote.pdf.

[16] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *Acm Sigact News*, vol. 33, no. 2, pp. 51–59, 2002.

[17] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "A comprehensive study of Convergent and Commutative Replicated Data Types," report, Jan. 2011. Accessed: Sep. 19, 2019. [Online]. Available: https://hal.inria.fr/inria-00555588.

[18] P. S. Almeida, A. Shoker, and C. Baquero, "Delta state replicated data types," *Journal of Parallel and Distributed Computing*, vol. 111, pp. 162–173, Jan. 2018, doi: 10.1016/j.jpdc.2017.08.003.

[19] R. Schwarz and F. Mattern, "Detecting causal relationships in distributed computations: In search of the holy grail," *Distrib Comput*, vol. 7, no. 3, pp. 149–174, Mar. 1994, doi: 10.1007/BF02277859.

[20] S. Katz and D. Peled, "Interleaving set temporal logic," *Theoretical Computer Science*, vol. 75, no. 3, pp. 263–287, 1990.

[21] S. S. Kulkarni, M. Demirbas, D. Madappa, B. Avva, and M. Leone, "Logical Physical Clocks," in *Principles of Distributed Systems*, vol. 8878, M. K. Aguilera, L. Querzoni, and M. Shapiro, Eds. Cham: Springer International Publishing, 2014, pp. 17–32.

[22] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978, doi: 10.1145/359545.359563.

[23] L. Ramabaja, “The Bloom Clock,” May 2019, Accessed: Sep. 19, 2019. [Online]. Available: http://arxiv.org/abs/1905.13064.

[24] H. Sanjuan, S. Poyhtari, and P. Teixeira, “Merkle-CRDTs,” May 2019.

[25] V. Enes, P. S. Almeida, and C. Baquero, “The Single-Writer Principle in CRDT Composition,” in *Proceedings of the Programming Models and Languages for Distributed Computing on - PMLDC ’17*, 2017, pp. 1–3, doi: 10.1145/3166089.3168733.

[26] R. Mört, *Content Based Addressing : The case for multiple Internet service providers*. 2012.

[27] J. Benet, “IPFS: Content addressed, versioned, P2p file system,” *arXiv preprint arXiv:1407.3561*, vol. (Draft 3), 2014.

[28] M. Selimi and F. Freitag, “Tahoe-LAFS Distributed Storage Service in Community Network Clouds,” in *2014 IEEE Fourth International Conference on Big Data and Cloud Computing*, Dec. 2014, pp. 17–24, doi: 10.1109/BD-Cloud.2014.24.

[29] S. C. Rhea, R. Cox, and A. Pesterev, “Fast, Inexpensive Content-Addressed Storage in Foundation.” in *USENIX Annual Technical Conference*, 2008, pp. 143–156.

[30] Protocol Labs, “Multihash,” *Multiformats*. https://multiformats.io/ (accessed Sep. 27, 2019).

[31] Protocol Labs, “Filecoin: A Decentralized Storage Network,” Jul. 19, 2017.

[32] T. Berners-Lee, “Linked Data,” *Design Issues*, Jun. 18, 2009. https://www.w3.org/DesignIssues/LinkedData.html (accessed Sep. 20, 2019).

[33] C. Bizer, T. Heath, and T. Berners-Lee, “Linked data: The story so far,” in *Semantic services, interoperability and web applications: Emerging concepts*, IGI Global, 2011, pp. 205–227.

[34] T. Heath and C. Bizer, “Linked Data: Evolving the Web into a Global Data Space,” *Synthesis Lectures on the Semantic Web: Theory and Technology*, vol. 1, no. 1, pp. 1–136, Feb. 2011, doi: 10.2200/S00334ED1V01Y201102WBE001.

[35] Brendan O’Brien and Michael Hucka, “Deterministic Querying for the Distributed Web,” Nov. 2017.

[36] P. Srisuresh and M. Holdrege, “IP Network Address Translator (NAT) Terminology and Considerations.” https://tools.ietf.org/html/rfc2663 (accessed Sep. 20, 2019).

[37] Protocol Labs, “Multiaddr,” *Multiformats*. https://multiformats.io/ (accessed Sep. 20, 2019).

[38] Secure Scuttlebutt, “Scuttlebutt Protocol Guide.” https://ssbc.github.io/scuttlebutt-protocol-guide/ (accessed Sep. 11, 2019).

[39] Eric Harris-Braun, Nicolas Luck, and Arthur Brock, “Holochain: Scalable agent-centric distributed computing,” Ceptr LLC, Feb. 15, 2018.

[40] R. W. Shirey, “Internet Security Glossary, Version 2,” *Network Working Group*, Aug. 2007. https://tools.ietf.org/html/rfc4949 (accessed Sep. 20, 2019).

[41] A. Herzberg, Y. Mass, J. Mihaeli, D. Naor, and Y. Ravid, “Access control meets public key infrastructure, or: Assigning roles to strangers,” in *Proceeding 2000 IEEE Symposium on Security and Privacy. S P 2000*, May 2000, pp. 2–14, doi: 10.1109/SECPRI.2000.848442.

[42] Mark Robert Henderson, Samuli Pöyhtäri, Vesa-Ville Piiroinen, Juuso Räsänen, Shams Methnani, and Richard Littauer, “The OrbitDB Field Manual,” Haja Networks Oy, Sep. 26, 2019.

[43] B. Newbold, S. Whitmore, and M. Buus, “DEP-0008: Multi-Writer.” 0008-multiwriter; Dat Protocol, Jul. 2018, [Online]. Available: https://www.datprotocol.com/deps/0008-multiwriter/.

[44] B. Newbold, S. Whitmore, and M. Buus, “DEP-0004: Hyperdb.” 0008-hyperdb; Dat Protocol, May 2018, [Online]. Available: https://www.datprotocol.com/deps/0008-hyperdb/.

[45] P. Frazee and M. Buus, “DEP-0002: Hypercore.” 0002-hypercore; Dat Protocol, Feb. 2018, [Online]. Available: https://www.datprotocol.com/deps/0002-hypercore/.

[46] A. Meyer, “Bamboo,” Sep. 16, 2019. https://github.com/AljoschaMeyer/bamboo (accessed Sep. 20, 2019).

[47] P. J. Leach, M. Mealling, and R. Salz, “A Universally Unique IDentifier (UUID) URN Namespace,” *Network Working Group*, Jul. 2005. https://tools.ietf.org/html/rfc4122 (accessed Sep. 20, 2019).

[48] R. Eremin, “A Redux-Inspired Backend,” *Medium*, Jan. 14, 2019. https://medium.com/resolvejs/resolve-redux-backend-ebcfc79bbbea (accessed Sep. 24, 2019).

[49] Facebook, “Flux: In-Depth Overview,” 2019. http://facebook.github.io/flux/docs/in-depth-overview (accessed Sep. 23, 2019).

[50] Redux, “Motivation,” *Redux*. https://redux.js.org/introduction/motivation (accessed Sep. 20, 2019).

[51] E. Evans, *Domain-driven design: Tackling complexity in the heart of software*. Addison-Wesley Professional, 2004.

[52] D. Abramov, “The Case for Flux,” *Medium*, Nov. 03, 2015. https://medium.com/swlh/the-case-for-flux-379b7d1982c6 (accessed Sep. 23, 2019).

[53] A. Wright, H. Andrews, B. Hutton, and G. Dennis, “JSON Schema: A Media Type for Describing JSON Documents,” Internet Engineering Task Force; Internet Engineering Task Force, Internet-Draft draft-handrews-json-schema-02, Sep. 2019. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-handrews-json-schema-02.

[54] T. Haerder and A. Reuter, “Principles of Transaction-oriented Database Recovery,” *ACM Comput. Surv.*, vol. 15, no. 4, pp. 287–317, Dec. 1983, doi: 10.1145/289.291.

[55] J. M. Snell and P. E. Hoffman, “JSON Merge Patch.” RFC 7396; RFC Editor, Oct. 2014, doi: 10.17487/RFC7396.

[56] P. C. Bryan and M. Nottingham, “JavaScript Object Notation (JSON) Patch.” RFC 6902; RFC Editor, Apr. 2013, doi: 10.17487/RFC6902.

[57] M. Kleppmann and A. R. Beresford, “A Conflict-Free Replicated JSON Datatype,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 10, pp. 2733–2746, Oct. 2017, doi: 10.1109/TPDS.2017.2697382.

[58] C. Shapiro, S. Carl, and H. R. Varian, *Information rules: A strategic guide to the network economy*. Harvard Business Press, 1998.

[59] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman, “Role-based access control models,” *Computer*, vol. 29, no. 2, pp. 38–47, Feb. 1996, doi: 10.1109/2.485845.

[60] HashiCorp, “Key Rotation.” https://www.vaultproject.io/docs/internals/rotation.html (accessed Sep. 20, 2019).

[61] M. Marlinspike, “The Double Ratchet Algorithm,” vol. Revision 1, p. 35, Nov. 2016, [Online]. Available: https://signal.org/docs/specifications/doubleratchet.

[62] N. Unger *et al.*, “SoK: Secure Messaging,” in *2015 IEEE Symposium on Security and Privacy*, May 2015, pp. 232–249, doi: 10.1109/SP.2015.22.