An event-sourced and decentralized database with IPLD and materialized views

Pick, Farmer, Sutula and Hill

www.textile.io
contact@textile.io

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 mainly being captured by apps and stored away from the person's control by the companies building those apps. An alternative, called user-siloed data is one area of research that if implemented successfully, could reverse the flow of value derived from personal data, from apps to users. In this paper, we investigate the data formats, access control, and transfer protocols necessary to build a system for user ownership of large-scale digital datasets. The aim of the proposed system is to help power a new generation of web technologies. Our solution combines a novel use of event sourcing with Interplanetary Linked Data (IPLD) to provide a distributed, scalable, and flexible database solution for decentralized applications.

Keywords: Databases; Event Sourcing; IPFS; IPLD
revised 18 September 2019

1. INTRODUCTION

Compared to their predecessors, modern cloud-based apps and services provide an extremely high level of convenience. Users can entirely forget about the risk of data loss 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 the 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 may not be readily accessible to other apps and services. This is app-siloed data.

While companies collecting user data may not be itself a significant problem, it has been shown that over time company incentives often shift from providing value to users to extracting value from users [11]. When this incentive shift happens for companies that have also been hoarding user data, that data may be a new source of value or even revenue for the company. It may be possible to stop this trend through extreme privacy measures, government intervention/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 user's data.

One of the most exciting aspects of user-siloed data is the the ability to build data-driven apps and services while users remain in control of their own data. Other projects have identified app silo-ed data as a problem [6, 32], and some have identified user-siloed data as a solution [42]. However, none so far have provided a sufficiently interoperable protocol for scalable storage and transmission of user-siloed application data. This is the key to building great applications on user-siloed 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 format, efficient synchronization, conflict resolution, access-control, scalable storage, and network communication.

Based on our findings, we propose a novel architecture for event sourcing with Interplanetary Linked Data (IPLD), designed to store, share, and host user-sioled datasets at scale. Our proposed design leverages new and existing protocols to solve major challenges involved with building a secure and distributed network for user data while at the same time providing the flexibility and scalability required by today's applications.

2. 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 that can be learned from event-sourcing and discuss drawbacks to using these approaches in decentralized systems. We provide an overview of some important technologies related to IPFS that make it possible to rethink event sourcing in a decentralized network. Finally, we cover challenges of security and access control on an open and decentralized network and discuss how they are used in popular databases external to IPFS.

2.1. Data Synchronization

2.1.1. CQRS, Event Sourcing, and Logs

When developing large-scale software systems, it is common to store data in a relational database management system (RDBMS). To model realistic systems, this type of framework often requires complex techniques for mapping data between domain models and database tables, where the same data model is used to both query and update a database. A powerful alternative 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 [45]: 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 [21].

DEFINITION 2.1. *(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.*

In modern applications it is critical to have reliable mechanisms for publishing updates and events (i.e., to support event-driven architectures), scalability (optimized write and read operations), forward-compatible application updates (e.g., code changes, retroactive events, and others), auditing systems, etc. To support such requirements, developers have begun to utilize event sourcing 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 event sourcing and CQRS-based applications, including Event Store [15], Apache Kafaka [2] and Samza [3], among others (see [23]).

DEFINITION 2.2. *(Command query responsibility segregation). 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 [29].*

DEFINITION 2.3. *(Event sourcing). Event sourcing (ES) is a design pattern for persisting the state of an application as an append-only log of state-changing events.*

ES is particularly useful in contexts where system components are distributed or decentralized, such as in decentralized applications (DApps). This is because the design of systems that use event sourcing often necessitate infrastructure for brokering messages between loosely coupled software components and services. This feature will become increasingly important as we outline background concepts related to data synchronization in the following sections.

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 [16]. The same principal means a log can be viewed as a mechanism to support an infinite number of valid state interpretations (Section 2.1.2). In other words, with minimal conformity, a single log can model multiple application states [31].

2.1.2. Views & Projections

DEFINITION 2.4. *(View). A (typically highly de-normalized) read-only model of the data. Views are tailored to the interfaces and display requirements of the application, which helps to maximize both display and query performance. Views that are backed by a database or filesystem-optimized access are referred to as materialized views.*

DEFINITION 2.5. *(Projection). An event handler and corresponding reducer/fold function used to build and maintain a view from a set of (filtered) events. While projections may lead to the generation of new events, their reducer should be a pure function (see [14] and [38] for examples from existing systems).*

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 the use case or user interface they are building. Multiple views can be built from the same underlying event log, and they can be quite different from one another.

Views themselves are enabled by projections[1]. Projections can be thought of as transformations or reducers that are applied to each event in a stream of events. 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 peers operating on the same stream of events have a consistent representation of the data.

2.1.3. Eventual Consistency

The CAP theorem [10, 17] 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 again. While an active area of research, designing systems with provable eventual consistency guarantees remains challenging [1, 49].

Definition 2.6. *(CRDT). A 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 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 (see Definition 2.1.1), sequences of deterministic changes, and CRDTs (see Definition section 2.1.3). Logs work best when there is only a single-writer or complete event causality is known. In many distributed systems with multiple log writers, a minimum requirement for synchronization through logs is that the essential order of events is respected and can be determined [22, 46]. For these cases, logical clocks are a useful tool for eventual consistency and total ordering [25]. 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 below).

2.1.4. Logical Clocks

In a distributed system, where multiple machines, each with an independent clock, are creating events, local timestamps can't be used to construct global event causality. Machine clocks are never perfectly synchronized [26], meaning that one machine's concept of "now" is not necessarily the same as another machine's. 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 distributed systems achieve eventual consistency. Examples include various logical clocks (Lamport [26] Schwartz [46], and Bloom [37], Hybrid variants [25], etc), which use counter-based time-stamping to provide partial ordering.

Cryptographically linked events can also represent a clock (See Section 2.2.3). One such example is called the Merkle-Clock [44]. The Merkle-Clock relies on properties of a Merkle-DAG to provide strict partial ordering between events, which has limitations [44, 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.

2.1.5. Conflict-Free Replicated Data Types

In distributed models that use clock-based ordering as described above, a replica can arrive in a state that will require conflict resolution through consensus or rollback. Those operations can be expensive, having adverse effects on the scalability of distributed systems, especially ones attempting to synchronize an application state.

CRDTs (see Definition 2.1.3) 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 [12, 44] for informative reviews of these types of data structures.

Whether a system (e.g., an 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

[1] Terminology in this section may differ from some other examples of ES and CQRS patterns, but reflects the underlying architecture and designs that the Textile team will elaborate on in 3

[2] Though non-commutative CRDTs may require a specific ordering of events in certain cases [44]

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.

2.2. Content-based addressing

Internet application architecture today is often designed as a system of clients (users) communicating to endpoints (hosts). Communication between clients and endpoints (servers) happens via the TCP/IP protocol stack and therefore largely relies on a mechanism referred to as 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 of storage, inefficient use of bandwidth, link rot, centralized control, and authentication issues. An alternative to location addressing, called content addressing may provide a solution to those problems. Content addressing is where the content itself is used to create an address and retrieve that content from a network [33].

2.2.1. IPFS & Content-based addressing

There are a number of systems that utilize content addressing to access content and information (e.g. Git, IPFS, Perkeep, Tahoe-LAFS), and there is an active body of literature covering its design and implementation [4, 41, 48]. The Interplanetary File System (IPFS) — which is a set of protocols to create a content-addressed, peer-to-peer filesystem [4] — is one such system. In IPFS, the address for any content is determined based on a cryptographic hash of the content itself. In practice, the 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 + encoding considerations[3]". That addressing system confers several benefits to the network, including tamper resistance (we can be confident that a given piece of content has not been modified en route if its hash matches what we were expected/requested) and de-duplication (because the same content from different peers will produce the same hash address). Additionally, IPFS content addresses are immutable and universally unique.

While content addressing doesn't tell you how to get a file, IPFS (via libp2p[4]) provides a system for moving files 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 all of the benefits that come with this type of file system design.

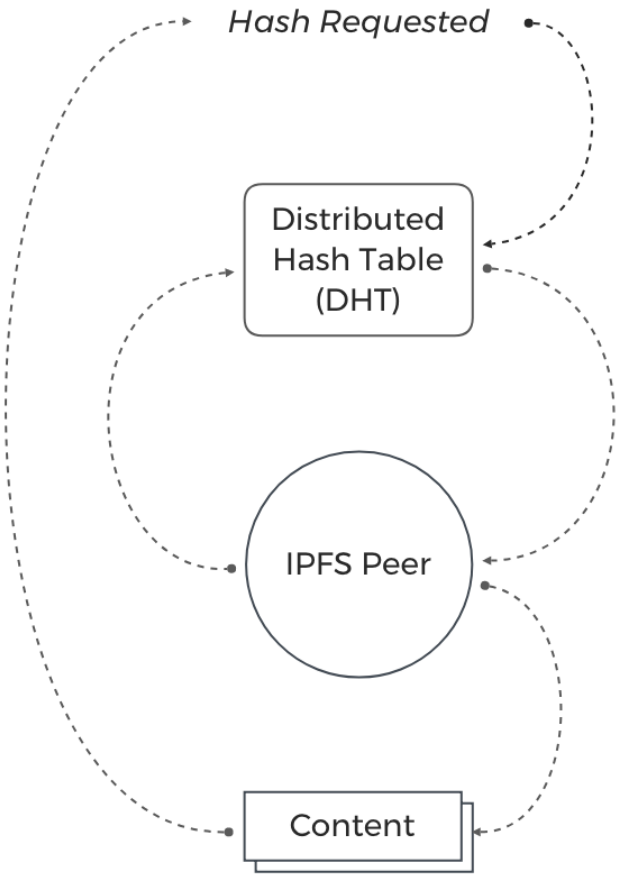


FIGURE 1: The crypographic hash of content is used to make a request to the network of IPFS peers. Using the built-in routing machanisms and the distributed hash table, a peer(s) hosting the requested content is identified and content is returned.

2.2.2. IPLD

As discussed above, IPFS uses the cryptographic hash of a given piece of content to define its content address (see [4] 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 others [36, e.g.,] MORE NEEDED), this common data format is called Interplanetary Linked Data (IPLD)[5]. As the name suggests, IPLD is based on principals of linked data [5, 8] with the added capabilities of a content-addressing storage network.

IPLD is used to publish linked data (subject, predicate, object triples in the linked data world [19]) spread across different hosts, and where everything

[3] `https://multiformats.io/`
[4] `https://libp2p.io/`
[5] `https://ipld.io/`

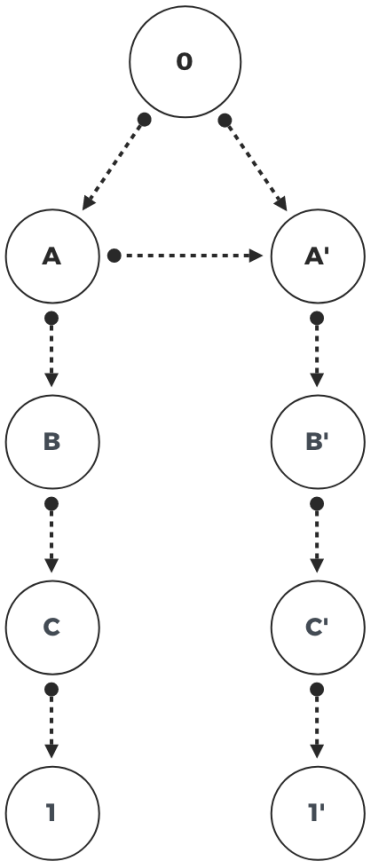


FIGURE 2: Divergent heads in a multi-writer Merkle-DAG

(e.g., entities, predicates, data sources) uses content addresses as unique identifiers. To form its structure, IPLD implements a Merkle DAG, or directed acyclic graph[6]. This allows all hash-linked data structures to be treated using a unified data model, analogous to linked data in the Semantic Web sense [9]. In practice, IPLD is represented as objects, each with `Data` and `Links` fields, where `Data` can be a small blob of unstructured, arbitrary binary data, and `Links` is an array of links to other IPLD objects.

2.2.3. Merkle-Clocks

A Merkle-Clock is a Merkle-DAG that represents a sequence of events. In other words, a Merkle-Clock is an append-only log [44]. When implemented on IPFS (or an equivalent network where content can be cryptographically addressed and fetched), Merkle-Clocks provide a number of benefits for data synchronization between replicas [44, sec. 4.3]:

1. Broadcasting the Merkle-Clock requires broadcasting only the current root CID. The whole Clock is unambiguously identified by the CID of its root and its full DAG can be walked down from it as needed.
2. The immutable nature of a Merkle-DAG allows every replica to perform quick comparisons and fetch only those nodes that it does not already have.
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.

On the downside, and as discussed above (Section 2.1.4), a Merkle-Clock cannot order divergent heads (or roots). For example, in Figure 2, two replicas (top row, bottom row) are attempting to write (left to right) 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, the two replicas stop receiving events from one another. To a third replica 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). See Section 2.1.3.

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 networks with many often offline replicas (mobile and IoT devices, laptops, etc.), these conditions are rarely met. Based on these observations, using a single Merkle-Clock to synchronize replicas can be problematic.

2.3. 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 database. 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 required by different use-cases. That is to say no matter what type of device we are talking about — be it a phone, desktop computer, browser, or Internet-enabled appliance — it should be able to communicate with other devices. However, each of the networking approaches described below comes with strengths and weaknesses when used to synchronize data.

2.3.1. Libp2p

The libp2p project provides a robust protocol communication stack. IPFS and a growing list of other

[6]Other examples of Dags include the Bitcoin Blockchain or a Git version history.

projects (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 peer-to-peer (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 [51] (NAT) traversal, peer discovery and handshake protocols, and even encryption and transport security — libp2p supports both unencrypted (e.g. TCP, UDP) and encrypted protocols (e.g. TLS, Noise) — 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 [34]. In addition to "transport layer" modules, libp2p provides several tools for sharing and/or disseminating data over a p2p network.

2.3.2. 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 they are not interested in — or *gossipsub* — which is a proximity-aware epidemic pubsub, where peers communicate with proximal peers, and messages can be routed more efficiently. In those implementations, there is a benefit to using pubsub in that no direct connection between publishers and subscribers is required.

Another benefit to using pubsub is the ability to publish topical sequences of updates to multiple recipients. Like libp2p, encryption is a separate concern and often added in steps prior to data transmission. However, like libp2p, pubsub doesn't offer any simple solutions for transferring encryption keys (beyond public keys), synchronizing datasets across peers (i.e. they aren't databases), or enforcing any measures for access control (e.g. anyone subscribed to a topic can also author updates on that topic). To solve some of those challenges, some systems introduce message echoing and other partial solutions. However, it makes more sense to use pubsub and libp2p as *building blocks* in systems that can effectively solve these issues, by choosing multi-modal communication strategies or leveraging tools such as deferred routing (e.g. inboxing) for greater tolerance of missed messages.

2.3.3. IPNS

Pubsub and libp2p have so far only dealt with *push-based* transfer of data, but IPFS also offers a useful technology for hosting *pull/request* based data endpoints called, IPNS. IPNS aims to address the challenge of mutable data within IPFS. IPNS relies on a global namespace (shared by participating IPFS peers) based on Public Key Infrastructure. By using IPNS, a content creator generates a new address in the global namespace and points that address to an endpoint (e.g. a CID). Using their private key, a content creator can update what static route the IPNS address refers to. IPNS isn't only useful for creating static addresses that can point to content addresses, IPNS is compatible with a naming system external from IPFS, such as DNS, onion, or bit addresses. However, many use-cases that require highly mutable data, require rapid availability of updates, or want flexible multi-party access control may not yet be suitable for IPNS. Taken together, libp2p, pubsub, IPNS, and IPFS more generally provide a useful toolkit for building robust abstractions to deliver fast, scalable, data synchronization in the decentralized network.

2.4. Data Access & Control

2.4.1. Identity

IPFS is an implementation of public key infrastructure, where every node on the IPFS network has a key-pair. In addition to using the key-pair for secure communication between nodes, IPFS also uses the key-pair as the basis for identity. Specifically, when a new IPFS node is created, a new key-pair is generated, and then the public key is transformed into the nodes Peer ID.

2.4.2. Agent-centric security

Agent-centric security refers to the maintainance of data integrity without leveraging a central or blockchain-based consensus. The general approach is just 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 state can be used to determine whether a given operation (e.g. delete data) is permitted. Many decentralized networks like Secure Scuttlebutt [47] and Holochain [13] make use of agent-centric security. Each of these systems leverage cryptographic signatures to validate peer identities and messages.

2.4.3. Access control

All filesystems 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 [50]. An ACL determines which users or processes can access an object and whether a particular user or process with access can modify or delete the object (see Figure 1).

Using ACLs in systems where identity is derived from various configurations of public-key infrastructure has been around for some time [20]. Still, many existing

TABLE 1: Example Access Control List.

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

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. ACLs should be 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 event-driven 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 (Section 2.1.2).

3. THE THREADS PROTOCOL

We propose Threads, a protocol and decentralized database that runs on IPFS meant to help decouple apps from user-data. The Threads protocol is designed based on an underlying, CQRS-inspired event-sourcing system for synchronizing data across collaborating peers on a network. Threads offer data ownership and a multi-role data access architecture where owners can set independent permissions for writing, reading, and following data updates. Threads differ from previous solutions by extending on the multiaddress addressing scheme to allow pull-based replica synchronization in addition to more common push-based synchronization seen in decentralized protocols. The flexible event-based structure enables client applications to model advanced states through aggregates, views, and custom CRDTs.

Threads are topic-based collections of single-writer logs. Taken together, these logs represent the current "state" of an object or dataset. The basic units of Threads, namely Logs and Events, provide a framework for users 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.

3.1. Event Logs

3.1.1. Single-writer Event Logs

In multi-writer systems, determining causal order in all replicas is particularity challenging. As previously discussed (Section 2.2.3), a solution based on a Merkle-Clock is only partially ordered. That is, this approach cannot achieve a total order of events without implementing a data-layer conflict resolution strategy [44]:

> 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).

In many cases, the total order of an event sequence is meaningful, and this type of deterministic merge strategy in not sufficient. A git merge highlights one such case, in which additional information is sometimes needed in order to resolve a line conflict. Ledger based transactions present a similar challenge where the absolute order of transactions between multiple peers really does matter.

Our solution to deal with divergent Merkle-Clocks is to institute a *single-writer rule*: A log can only be updated by a single replica or *identity*. An Event Log is a *single-writer Merkle-Clock* that can be totally ordered (Figure 3). *Separate* Event Logs can be composed into advanced structures, including CRDTs [12].

For any given Log, Events are authored by a single IPFS Peer, or *Writer*.

Definition 3.1. *(Writer). The single IPFS Peer capable of writing to an Event Log.*

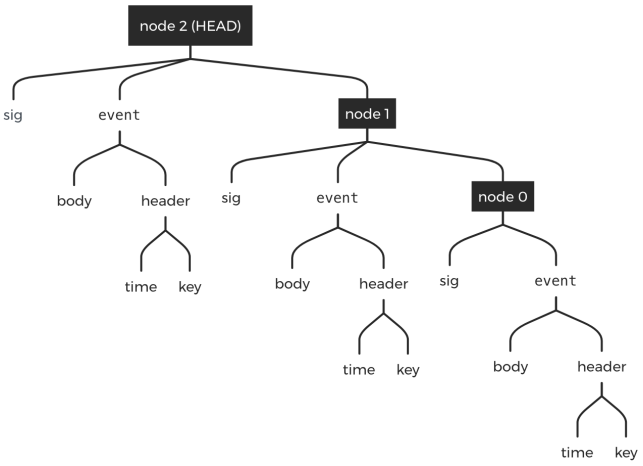


FIGURE 3: A single-writer Merkle-Clock (Event Log).

This single-writer setup is a core feature of Logs, and provides properties unique to the Threads protocol. For clarity, we can similarly define a *Reader* as any other Peer capable of reading a Log.

Definition 3.2. *(Reader). Any Peer capable of reading a Log. Practically speaking, this means any Peer with the Log's Read Key (Section 3.1.3).*

As seen above in Section 2.2.3, a Merkle-Clock is simply a Merkle-DAG of *Events*:

Definition 3.3. *(Event). A single node in a Merkle-Clock, stored on IPFS.*

3.1.2. Multi-addressed Event Logs

Together with a cryptographic signature, an Event is written to a log with an additional node (see Figure 3) enabling log verification by readers (Section 3.1.3).

At a minimum, a node must link to its most immediate ancestor. However, links to older ancestors are often included as well to improve concurrency during traversal and verification [30].

As shown Appendix A, an Event's actual content, or body, is contained in a separate node. This allows Events to carry any arbitrary node structure, from complex directories to raw bytes.

Much like IPFS Peers, Logs are identified on the network with addresses, or more specifically, with multiaddresses[34]. Here we introduce IPEL, or Interplanetary Event Log, as a new protocol tag to be used when composing Log multiaddresses. To reach a Log via it's IPEL multiaddress, it must be encapsulated in an IPFS Peer multiaddress (see Example 1).

In practice, to reach a Log via it's IPEL multiaddress, it must be encapsulated in an IPFS Peer multiaddress (Example 1).

Unlike peer multiaddresses, Log addresses are not stored in the global IPFS distributed hash table [4] (DHT). Instead, they are collected from Log Events. This is in contrast to mutable data via IPNS for example, which requires querying the network (DHT) for updates. Instead, updates are requested directly from the (presumably trusted) peers that produced them, resulting in a hybrid of content-addressed Events arranged over a data-feed[7] like topology. Log addresses are recorded in an address book, similar to IPFS Peer address book (Example 2).

Addresses can 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.

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 (like mobile devices), Textile Logs are designed with a built-in replication or follower mechanism.

Definition 3.4. *(Follower). Log Writers can designate other IPFS Peers to "follow" a Log, potentially replicating and/or republishing Events. A Follower is capable of receiving Log updates and traversing linkages via the Follow Key (Section 3.1.3), but is not able to read the Log's contents. Followers should be server-based — i.e., always online and behind a public IP address.*

Followers are represented as additional addresses, meaning that a Log address book may contain *multiple* multiaddresses for a single Log (Example 1).

In practice, Writers are solely responsible for announcing their Log's addresses. This ensures a conflict-free address list without additional complexity. Some Followers may be in the business of replicating Logs (Section 5.1.2), in which case Writers will announce the additional Log address to Readers. This allows them to *pull* (or subscribe to push-based) Events from the Follower's Log address when the Writer is offline or unreachable (Figure 6).

3.1.3. Keys & Encryption

Textile Logs are designed to be shared, composed, and layered into datasets (Figure 4). Therefore, Logs are encrypted by default in a manner that enables access control (Section 4.5.2) and the Follower mechanism discussed in the previous section.

Definition 3.5. *(Identity Key). Every Log requires an asymmetric key-pair that determines ownership and identity. The private key is used to sign each Event added to the Log, so down-stream processes can verify the Log's authenticity. Like IPFS peers, the public key of the Log is used as an identifier (Log ID).*

The body, or content of an event, is encrypted by a *Content Key*. Content Keys are generated per-content and never reused. The Content Key is distributed directly in the header of the Event Block. We define the Content Key as follows,

Definition 3.6. *(Content Key). The Content Key is a variable-format key used to encrypt the body (content) of an event. 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 per event,*

1. *When broadcasting events to many possible recipients, a single-use symmetric key is generated per unique content body.*
2. *When sending events to specific recipients, the recipient's public key can be used to restrict access*

[7] This is similar to the append-only message feeds used in Secure Scuttlebutt's global gossip network [47]

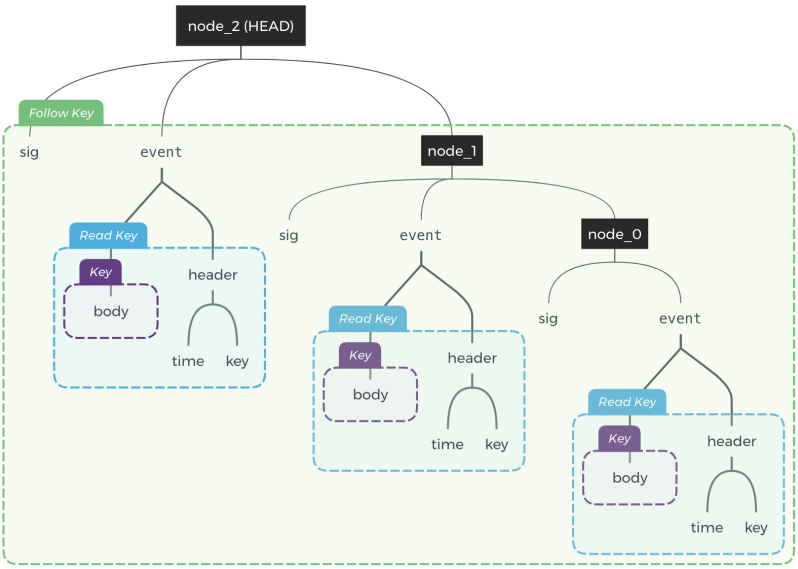


FIGURE 4: The three layers of Log Event encryption.

from all others[8].

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 Event Block. Therefore, the Event Block itself is further encrypted using a *Read* key. The Read Key is not distributed within the Log itself but is distributed to any peers to grant them access to the content of the Log.

Definition 3.7. *(Read Key). The Read Key is a symmetric key created by the Log owner and used to encrypt the Content Key in each event.*

Finally, the encrypted Event Block, its signature, and the IPLD linkage(s) from an Event to its antecedents are encrypted together using a Follow Key. Follow Keys allow Logs to be *followed* by peers on the network who do not have access to any content within the event, and can only see signatures and linkage(s) between Events.

Definition 3.8. *(Follow Key). The Follow Key is a symmetric key created by the Log owner and used to encrypt the entire event payload before adding the event to the Log.*

Much like the Log address book, Log keys are stored in a key book (Example 2).

3.2. Threads

In Textile, the *interface* to Logs is managed as a Thread. As mentioned previously, a Thread is a collection of Logs on a given topic. Threads are an event-sourced, distributed database, and can be used to maintain a single, collaboratively edited, watched, or hosted dataset across multiple Peers. Threads provide the mechanism to combine multiple Logs from individual authors into singular shared states through the use of either cross-Log sequencing (e.g. using a Bloom Clock, Merkle-Clock, or Hybrid Logical Clock [25]) or a CRDT (Section 2.1.5).

[8] Much like private messages in Secure Scuttlebutt (`https://www.scuttlebutt.nz/concepts/private-message`)

```
# Log multiaddress
 /ipel/12D3KooWC2zyCVron7AA34N6oKNtaXaZB51feG9rBkr7QbCcW8ab

# Encapsulated multiaddress
 /ip4/127.0.0.1/tcp/1234/p2p/12D..dwaA6Qe/ipel/12D..bCcW8ab

# Address book
[
  /p2p/12D..dwaA6Qe/ipel/12D..bCcW8ab,
  /p2p/12D..dJT6nXY/ipel/12D..bCcW8ab # Follower
]
```

Example 1: The Log Multiaddress.

3.2.1. Identity

A unique Thread IDentity (TID) is used to group together Logs which compose a single dataset and as a topic identifier within Pub/Sub-based synchronization. TIDs are defined with the format shown Figure 5.

TIDs share some similarities with UUIDs [27] (version and variant) and IPFS-based CIDs and are multibase encoded[9] for maximum forward-compatibility.

DEFINITION 3.9. *(Multibase Prefix). The encoding type used by the multibase encoder. 1 byte.*

Base32 encoding is used by default. However, any multibase-supported string encoding may be used.

DEFINITION 3.10. *(Version). ID format version. 8 bytes max. This allows future version to be backwards-compatible.*

DEFINITION 3.11. *(Variant). Used to specify thread-level expectations, like access-control. 8 bytes max. See section 3.2.2 for more about variants.*

DEFINITION 3.12. *(Random Number). A random number of a user-specified length. 16 bytes or more (see Example 3).*

3.2.2. Variants

Certain ID *variants* may be more appropriate than the others in specific use cases. 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.

DEFINITION 3.13. *(Raw). This variant declares that consumers are not expected to make additional assumptions. This is the default variant (See Example 3a).*

DEFINITION 3.14. *(Access-Controlled): This variant declares that consumers should assume an access control list is composable from Log Events. The ACL represents a permissions rule set that must be applied when reading data (Section 4.5.2 and Example 3b).*

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

```
type AddrBook interface {
   AddAddr(thread.ID, peer.ID, ma.Multiaddr,
       time.Duration)
   AddAddrs(thread.ID, peer.ID, []ma.Multiaddr,
       time.Duration)
   SetAddr(thread.ID, peer.ID, ma.Multiaddr,
       time.Duration)
   SetAddrs(thread.ID, peer.ID, []ma.Multiaddr,
       time.Duration)
   UpdateAddrs(t thread.ID, id peer.ID, oldTTL
       time.Duration, newTTL time.Duration)
   Addrs(thread.ID, peer.ID) []ma.Multiaddr
   ClearAddrs(thread.ID, peer.ID)
}
type KeyBook interface {
   PubKey(thread.ID, peer.ID) ic.PubKey
   AddPubKey(thread.ID, peer.ID, ic.PubKey) error
   PrivKey(thread.ID, peer.ID) ic.PrivKey
   AddPrivKey(thread.ID, peer.ID, ic.PrivKey) error
   ReadKey(thread.ID, peer.ID) []byte
   AddReadKey(thread.ID, peer.ID, []byte) error
   FollowKey(thread.ID, peer.ID) []byte
   AddFollowKey(thread.ID, peer.ID, []byte) error
}
```

Example 2: The AddrBook interface for storing log addresses and the KeyBook interface for storing log keys.

3.2.3. Log Synchronization

Log Writers, Readers, and Followers synchronize the state of their Logs by sending and receiving Events. Inspired by Git[10], a reference to the latest Event in a Log is referred to as the *Head* (or sometimes the *root*). When a new Event is received, Readers and Followers simply advance their Head reference.

Regardless of the network protocol, Events are transported between Peers in a standardized *Event Envelope*:

DEFINITION 3.15. *(Event Envelope). An over-the-wire message containing an Event and the sender's signature of the Event.*

A new Thread is created by generating a TID and Log. The Log's creator is the Writer, meaning it has possession of the Log's Identity, Read, and Follow Keys. All of these keys are needed to compose Events. At this point, the Thread only exists on the Writer's machine. Whether for collaboration, reading, or following, the process of sharing a Thread with other Peers starts by authoring a special Event called an *Invite*, which contains a set of keys from all of the Thread's Logs, called a *Key Set*.

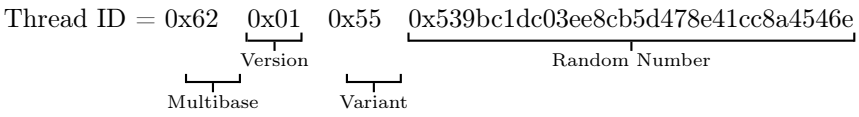
DEFINITION 3.16. *(Invite). An Event containing a mapping of Log IDs to Key Sets, which can be used to join a Thread. Threads backed by an access control list (Section 4.5.2) will also include the current ACL for the Thread in an Invite. This enables Peers to invite others to only read or follow a Thread, instead of becoming a full-on collaborator, i.e., a new Log Writer.*

DEFINITION 3.17. *(Key Set). A set of keys for a Log. Depending on the context, a Key Set may contain the Follow and Read Key, or just the Follow Key. Encrypted with the recipient's public key.*

The Invite is authored in the sender's Log. Because the recipient does not yet have this Log's Key Set, the Event is encrypted with the recipient's public key. If the recipient accepts the Invite, they will author another

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

FIGURE 5: Components of a Thread Identity.




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

# (b) ACL enabled identity. V1, 256bit.
bafyoiobghzefwlidfrwkqmzz2ka66zgmdmgeobw2mimktr5jivsavya
```

Example 3: Identity variants.

special Event called a *Join* in a new Log of their own.

Definition 3.18. *(Join). An Event containing an invitee's new Log ID and Key Set, encrypted with the Key Set of the* inviting Peer's Log.

For a Join to be successful, all Log Writers must receive a copy of the new Key Set so they can properly handle future Events in the new Log. Instead of encrypting a Join with the public key of each existing Writer, we can encrypt a single Join with the Key Set of the inviting Peer's Log, which the other Writers also have.

Once a Peer has accepted an Invite, it will receive new Events from Log Writers. In cases where the invitee becomes a collaborator (i.e., a Writer) it is also responsible for sending its own Events out to the network.

Sending

Sending is performed in multiple phases because, invariably, some Thread participants will be offline or unresponsive.

1. New Events are pushed[11] directly to the Thread's other Log Writers.
2. New Events are pushed directly to the target Log's Follower(s), who may not maintain their own Log.
3. New Events are published over Libp2p's gossip-based Pub/Sub infrastructure using TID as a topic, which provides potentially unknown Readers or Followers with an opportunity to consume Events in real-time.

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

1. New Events may be pushed directly to web-based participants over a WebSocket.
2. New Events 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 Events may trigger web-hooks[12], which could enable many complex (e.g., IFTTT[13]) workflows.

Receiving

There are multiple paths to receiving new Events, that together, maximize connectivity between Peers who are often offline or unreachable.

1. Log Writers can receive Events directly from the author.
2. Events can be pulled from replicating Followers via HTTP, libp2p, RSS, Atom, etc. a. In conjunction with push over WebSockets (seen in Step 2 of the additional push mechanisms above), this method provides web-based Readers and Followers with a reliable mechanism for receiving Log Events (Figure 6).
3. Writers and readers can receive new Events via a Pub/Sub subscription at the TID.

3.2.4. Log Replication

The notion of the Follow Key (Section 3.1.3) makes duplicating all Log Events trivial. This allows any Peer in the network to be granted the responsibility of replicating data from another Peer without having read access to the data contained within the Log entries. This type of Log replication can act as a data backup mechanism. It can also be used to build services that react to Log Events, potentially pushing data to disparate, non-Textile systems, especially if the replication service is granted read access to the Log Events (Section 3.2.3).

[11] Here push means "send to multiaddress(es)", which may designate different protocols, e.g., p2p, HTTP, Bluetooth, etc.

[12] https://en.wikipedia.org/wiki/Webhook

[13] https://ifttt.com

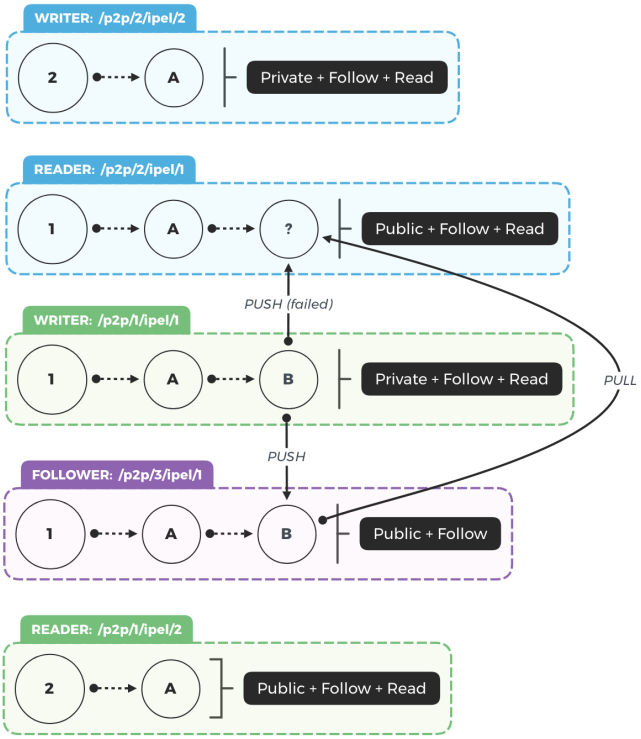


FIGURE 6: A pull-based request from a Follower.

4. THREAD INTERNALS

To make Threads as easy to adopt and use as possible, we've designed the internal event store to take advantage of ideas from several existing CQRS and ES systems, as well as ideas and designs from Flux[14]/Redux[15] [39] and domain driven design [...] (DDD).

One more linking sentence ...

4.1. Overview

For application developers familiar with common application state management issues, tools such as Flux/Redux provide opinionated frameworks for making state mutations *predictable* by imposing certain restrictions on how and when updates can happen. If you take a look at the architecture of a Flux (or Redux) application, you'll notice several core similaries to CQRS and ES systems (see Figure MISSING). In both cases, the focus is on unidirectional data flows that build downstream views from atomic updates in the form of events (or actions).

In Threads, Views are updated by subscribing to specific Event(s), and applying a Reducer function that is used to update the View state (which may be *materialized* to local persistent storage). This simple subscription system provides a flexible framework for building complex View-driven application logic.

4.2. Actions

Both Flux and Redux use 'Actions', which are essentially events. Actions cause predictable updates to state via 'Reducer' functions in both Flux and Redux, and in both cases, the 'Store' is the current state of the application (to which we can send queries or subscribe to changes). The differences arise in how they two handle actions: in Flux, 'Actions' are dispatched from a singleton 'Dispatcher', which then calls a set of 'Reducer' functions that mutate a set of 'Stores'. Conversely in Redux, 'Actions' are dispatched to a singleton 'Store', which then applies a pure (in the functional sense) 'Reducer' (which can be composed of multiple pure reducers) to replace the application state with a new immutable state.

4.2.1. Action Creators

Both Flux[16] and Redux[17] have the concept of 'Action' creators, which are simple functions that create 'Actions'. In Redux, these create and return a new 'Action', in Flux, they generally also dispatch said 'Action' (Redux folks call these bound 'Action' creators). As you might have noticed, if designed around a domain, Action creators provide bounded context that can be (roughly) compared to an aggregate root (i.e., a cluster of objects, that create a clear reference to the root aggregate).

4.3. Dispatcher

Dispatcher is used to store actions and dispatch events to stores. It would be a Datastore, which would be responsible for persisting to its own internal Event Store (it essentially is both the Event Store and the Event Bus/Dispatcher). It is based on a 'TxDatastore", so that Actions' are only ever 'Dispatch'ed if they were successfully stored in the Event Store. Additionally, all "side effects" should happen within the 'Transaction'. This is a big reason for the use of Flux rather than Redux architecture... because we need some way to consistently and transactionally both store and transmit actions. The Dispatcher is actually responsible for running a 'Store''s 'Reducer', once that 'Store' has registered with the Dispatcher. Its quite simple, and ends up being a pretty tiny class/object.

While a 'Store''s 'Reducer' can actually 'Dispatch' new 'Actions' along the way, this is generally discouraged due to the potential for race conditions and infinite loops. Having said that, borrowing concepts from ACID databases and some Redux middlewares, it is possible to avoid these issues by using 'Transactions'

[14] http://facebook.github.io/flux

[15] Flux/Redux build on concepts from CRQS and ES themselves, and Redux is arguably an implementation of the Flux application architecture (i.e., Flux is a pattern, Redux is a library).

[16] `https://facebook.github.io/flux/docs/in-depth-overview`

[17] `https://redux.js.org/basics/actions`

to some degree. Speaking of 'Transactions', if done right, this gets us essentially what you get using Redux "Sagas", but with a much simpler interface (basically no interface, the 'Transactions' are all hidden inside the 'Dispatcher'), which is also really cool. Another useful pattern here which works nicely with 'Transactions' is batched 'Actions' (which is the preferred way[18] to send multiple actions in Redux). This works by grouping Actions into meta 'Actions' ('BatchedActions'), and then running their 'Reducers' in series. An example of where this is useful is something like image resizing. If you 'Dispatch' an 'ImageAdded' 'Action', you'd ideally also batch that with a 'ImageResized' 'Action' or two along the way.

Commands are Actions, the Event Store is your Dispatcher, Projections are Reducers, Materialized Views are Stores, and Queries can be called from Views.

Ok, so that's pretty nice, but what about on the read side? Here's where we have to borrow some ideas and terminology from a combination of CQRS and the Resolve library (because Flux and Redux don't really have a read side, just state).

4.4. Read Models

Like many CQRS-based systems, the "public" API for Threads revolves around the concept of views, which are used to optimize queries on the local event store.

This focus on uni-directional data flows that produce immutable state updates via pure functions provides an intuitive framework for predictable state management (just like it says on the box). And it is really nice on the client side. However, it doesn't scale very well to large sets of data if you are updating your whole state tree each time a new event comes in. So how might we use this nice front-end pattern on the back-end? Others have also thought about this, and they [settled on](https://medium.com/resolvejs/resolve-redux-backend-ebcfc79bbbea)... you guessed it, CQRS+ES+DDD.

If we stick with the concept of dispatching 'Actions' (Flux and Redux do it differently, but that's an implementation detail), on the write side, then based on these 'Actions', the read side updates the read models, and these models provide data to the queries. In practice, a 'ReadStore' is defined by a projection function ('Reducer') and a set of query resolver functions ('Resolvers').

The 'Reducer' functions build a 'ReadStore''s state based on incoming 'Actions'. Query 'Resolvers' use data from the accumulated state to answer specific queries. 'Reducers' should be "pure" functions (where possible), though in practice they aren't really pure, because we are mutating (persistent) state here. So basically, we have a Flux 'Store' implementation here. The 'Store'(s) hold/define their own 'Reducer' and underlying state. Note that we may have multiple 'Stores' here (like in Flux).

We might also want to think about 'ViewStore's, which would be more like a Redux 'Store', in that they would be smaller, read-only (immutable), would be built on the fly, and defined by a pure reducer function. There are some additional nice things about 'ViewStore's (such as 'Snapshots') that could be easily baked in. The nice thing about them is they are very much in line with ES and Redux.

Like in Flux, 'Stores' (as they are outlined above) have several specific properties:

Cache data
Expose public getters to access data (never have public setters)
Respond to specific 'Actions' from the 'Dispatcher'
Always emit a change when their data changes
Only emit changes during a 'Dispatch'

The other thing about 'Stores' is their set of 'Resolver' methods. These are just pre-defined query functions (ideally the underlying storage is optimized to support these queries) that exist on the 'Store' instance (the public APIs so to speak). The interesting thing here is that these can similarly be defined around a domain to produce essentially domain aggregates like in DDD.

'ViewStores' are a special kind of 'ReadStore'. They are queried based on aggregate ID (maybe?), and regenerated on the fly from past events. 'ViewStores' are like Redux state. They should only ever replace their internal state, not mutate it.

4.4.1. Databse Views

For example, what if your domain was a database? In which case, our earlier thoughts about database wrappers might become something like this:

The public API for a Thread already provides several features that you would expect when operating on a dataset or table within a database. Indeed, each Thread is defined by a unique ID (its Identity), and provides facilities for access control and permissions, networking, sharing, invites, and more. As such, it is not difficult to imagine a simplified API in which Threads are exposed via interfaces compatible with *existing* datastores and/or communications systems.

The key here is to create simple, higher-level interfaces on top of Textile Threads that simplifies dealing with events and data, while still maintaining the power and flexibility of CQRS + ES. Developers should not have to learn a whole new set of terms and tools to take advantage of Textile's Threads capabilities. These simple, public-facing APIs will be comfortable to application developers looking to leverage highly-distributed systems that connect user-controlled data, with minimal configuration and maximum interoperability.

[18] https://twitter.com/dan_abramov/status/656074974533459968

```
type TextileKVStore interface {
  Put(key string, value Node) error
  Get(key string) (Node, error)
  Del(key string) error
}
```

Example 4: The Key-Value store interface.

```
type TextileDocumentStore interface {
  Put(doc Inedexable) error
  Get(key string) (Indexable, error)
  Del(key string) error
  Query(query Query) ([]Indexable, error)
}
```

Example 5: The Document store interface.

A database built on Threads would map database operations into `Actions`. Here, the *domain* is the databse. These `Actions` would then mutate a `ReadModel` via its corresponding `Reducer` function, as per above. The `Store` could be persisted or kept in-memory (as a `ViewModel`), and would be directly queried via a corresponding database-style `Resolver` metods. Because Textile uses ES patterns, any delete-type operation would simply lead to an internal *tombstone* `Action`, marking the database item for removal from the `ReadStore`.

Key/Value

A key/value store built on Threads would "map" these database `Actions` into key/value operations, such as `Put`, `Get`, and `Del`. These `Actions` would then mutate a map-like `ReadModel`. To complete the database-like API, the `ActionCreator` and `ReadModel` methods outlined in Section MISSING could be encapsulated in a database structure that satisfies a given interface (see for example, Figure 4).

No-SQL

Other database abstractions include a no-sql style document store for storing and indexing arbitrary structs and/or JSON documents. The interface for such as store, again built using a materialized `ReadStore`, might look like Figure 5, where `Indexable` could be satisfied by any structure with a `Key` field and `Query` might be taken from the `go-datastore` interface library[19] or similar.

Others

Similar abstractions could (and will) be used to implement additional database types and functions. Tables, feeds, counters, and other simple stores can also be built on Threads, and may require ORM-based solutions to support persistence to certain backing stores[20]. Each database style would be implemented as a standalone wrapper/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 "example" libraries.

CRDTs

Eventually consistent, CRDT-based stores can also be implemented on top of Threads. CRDT-based stores are particularly useful for managing Views of a document in a multi-peer collaborative editing environment (like Google Docs or similar). For example to support offline-first, potentially concurrent edits on a shared JSON document, one could implement a JSON CRDT datatype [24] that applies updates on a JSON document View model. Libraries such as Automerge[21] provide useful examples of reducer function that make working with JSON CRDTs relatively straightforward, and implementations in other programming languages are also available. A practical example of using a JSON CRDT in Textile is given in section 4.5.2, where it is used to represent updates to an ACL document for an access-controlled Thread implementation.

In practice, Redux doesn't care how a developer persists the application state, which means a Thread's event store and read-only materialized views are excellent candidates. This focus on unidirectional data flow in perfectly in line with architecture proposed in this paper, and as such, a compatible Redux API will be built on top of Textile Threads.

4.5. Thread Extensions

At an abstract level, the Textile protocol provides a highly distributed framework for building shared, offline first, datastores that are fault tolerant[22], 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+ES (and related) literature to "extend" the Threads protocol with additional features and controls. Some examples of extensions that are included by default in Textile's Threads implementation are outlined in this section to provide some indication of the extensibility of Threads.

[19] `https://github.com/ipfs/go-datastore`

[20] In-memory stores are almost always implementable using compositions of basic data structures.

[21] `https://github.com/automerge/automerge`

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

4.5.1. Snapshots and Compaction

Snapshots[23] are simply the current state of a View at a given point in time. They can be used to rebuild the state of a view 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 by replaying any events generated since the latest Snapshot (with a corresponding reduction in the total number of events that need to be processed) using the View's Reducer function. Multiple peers processing the same Log could create a Snapshot every 1000 events and be guaranteed to create the exact same Snapshot because each Peer's Event counts are identical[24].

In practice, Snapshots are written to their own Event log and stored locally. They can potentially be synced (Section 3.2.3) 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 state during recovery.

Compaction can similarly speed up re-hydration of state, and is useful when only the latest Event per Event type is required. In addition to being used as a starting point for initialization and recovery, compacted logs can be used to *replace* the Log from which they were created in order to free up local disk space. In Textile, compaction becomes much easier because a log only has one writer and conflicts are not possible. In practice, Log compaction is a local-only operation (i.e., other Peers do not need to be aware that Compaction was performed).

4.5.2. Access Control

One of the most important properties of a shared data model is the ability to apply access control rules. There are two forms of access control possible in Threads, Sequence-level ACLs and Thread-level ACLs. Thread-level access control lists (ACLs) allow creators to specify who can *follow, read, write, and delete* Thread data. Similarly, sequence-level ACLs provide more granular control to Thread-writers on a per-sequence (see Definition 6) basis. Both types of ACLs are materialized views, or more specifically, JSON CRDT documents, built from Log Events (Section 4.4.1). ACLs implemented as JSON documents provides two advantages over static or external ACL rules (although static and external ACLs are also possible). First, ACLs are fully mutable, allowing developers to create advanced rules for collaboration with any combination of readers, writers, and followers. Second, because ACLs are JSON documents, ACLs can specify their own editing rules (i.e. allowing multiple Thread participants to modify the ACL) in a self-referencing way.

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

[24]Assuming any network partitions are only short-lived (i.e., that peers are able to share events consistently).

Definition 4.1. *(Sequence). An Event Sequence is a series of ordered events referring to a specific entity or object. For example, an ACL JSON document is a single entity made up of a sequence of Thread Events. A Sequence might have a unique UUID (see Example 6) as in the example below.*

Textile's Threads includes ACL management tooling based on a *Role-based access control* [43] 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[25]: *No-access, Follow, Read, Write, and Delete.*

Definition 4.2. *(No-access). No access is permitted. This is the default role.*

Definition 4.3. *(Follow). Access to Log Follow Keys is permitted. Members of this role are able to verify Events and follow linkages. The Follow role is used to designate a "follower" peer for offline replication and/or backup.*

Definition 4.4. *(Read). Access to Log Read Keys is permitted in addition to Follow Keys. Members of this role are able to read Log Event payloads.*

Definition 4.5. *(Write). Members of this role are able to author new Events, which also implies access to Log Follow and Read Keys. At the Thread-level, this means authoring a Log. At the document-level, the Write role means that Events in this Log are able to target a particular document.*

Definition 4.6. *(Delete). Members of this role are able to delete Events, which implies access to Log Follow Keys. In practice, this means marking an older Event as "deleted".*

A typical Thread-level ACL JSON (see Example 7) can be persisted to a local document store as part of a materialized view.

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 follower, and `12D..P2c6ifo` has been given read access.

A Thread-level ACL has it's own document ACL, which also applies to all other document ACLs (see Example 8).

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

```
# UUID
bafykrq5i25vd64ghamtgus6lue74k
```

Example 6: Sequence ID.

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

Example 7: ACL JSON document with `_id` being the unique ID.

```
{
  "_id": "bafykrq5i25vd64ghamtgus6lue74k-acl",
  "default": "no-access",
  "peers": {
    "12D..dwaA6Qe": ["write", "delete"],
  }
}
```

Example 8: Thread and document ACL

This means that only `12D..dwaA6Qe` is able to alter the access-control list.

5. CONCLUSION

In this paper, we described the challenges and considerations when attempting to create 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 presented a novel solution to satisfy each of these six requirements. We have introduced a solution that extends on IPFS and prior research done by Textile 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 Views, we can support important features such as access control lists. 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.

TABLE

5.1. Future Work

The research and development of Textile 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.

5.1.1. Enhanced Log Security

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, on a network of Textile peers. Based on the Follow Key now available in Threads, Bots would be able to relay, replicate, or store data that is synchronized via real-time updates in a *trust-less* way. Bots can additionally enhance the IPFS network by providing a framework to build and deploy many kinds of trust-less or trusted services. Examples include simple archival, caching and republishing Bots. Other examples include payment, re-encryption, or bridges to Web 2.0 services to offer decentralized access to Web 2.0.

5.1.2. Textile: The Thread & Bot Network

The use of a single Read and Follow Key for an entire Log means that, should either of these keys be leaked via malicious (or other/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 [18], and/or incorporating the Double Ratchet Algorithm [28] for forward secrecy [52].

ACKNOWLEDGEMENTS

Thanks to our funders and the excellent Textile community for helpful suggestions, feature requests, and reviews.

REFERENCES

[1] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. “Delta State Replicated Data Types”. In: *Journal of Parallel and Distributed Computing* 111 (Jan. 2018), pp. 162–173. ISSN: 07437315. DOI: 10.1016/j.jpdc.2017.08.003.

[2] *Apache Kafka*. URL: https://kafka.apache.org/ (visited on 09/19/2019).

[3] *Apache Samza*. URL: http://samza.apache.org/ (visited on 09/19/2019).

[4] Juan Benet. “IPFS: Content Addressed, Versioned, P2p File System”. In: *arXiv preprint arXiv:1407.3561* (Draft 3) (2014).

[5] Tim Berners-Lee. *Linked Data*. June 18, 2009. URL: https://www.w3.org/DesignIssues/LinkedData.html (visited on 09/20/2019).

[6] Tim Berners-Lee and Kieron O'Hara. "The read–write Linked Data Web". In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 371.1987 (Mar. 2013), p. 20120513. DOI: 10.1098/rsta.2012.0513. URL: https://royalsocietypublishing.org/doi/full/10.1098/rsta.2012.0513 (visited on 09/23/2019).

[7] Dominic Betts et al. *Exploring CQRS and Event Sourcing: A Journey into High Scalability, Availability, and Maintainability with Windows Azure*. 1st. Microsoft patterns & practices, 2013. ISBN: 978-1-62114-016-0.

[8] Christian Bizer, Tom Heath, and Tim Berners-Lee. "Linked Data: The Story so Far". In: *Semantic Services, Interoperability and Web Applications: Emerging Concepts*. IGI Global, 2011, pp. 205–227.

[9] Brendan O'Brien and Michael Hucka. "Deterministic Querying for the Distributed Web". Whitepaper. Nov. 2017. URL: https://qri.io/papers/deterministic_querying/.

[10] Eric Brewer. "Towards Robust Distributed Systems". In: *19th ACM Symposium on Principles of Distributed Computing (PODC)*. Invited Talk. 2000. URL: http://pld.cs.luc.edu/courses/353/spr11/notes/brewer_keynote.pdf.

[11] Chris Dixon. *Why Decentralization Matters*. Oct. 26, 2018. URL: https://medium.com/s/story/why-decentralization-matters-5e3f79f7638e (visited on 09/19/2019).

[12] Vitor Enes, Paulo Sérgio Almeida, and Carlos Baquero. "The Single-Writer Principle in CRDT Composition". In: *Proceedings of the Programming Models and Languages for Distributed Computing on - PMLDC '17*. The Programming Models and Languages for Distributed Computing. Barcelona, Spain: ACM Press, 2017, pp. 1–3. ISBN: 978-1-4503-6356-3. DOI: 10.1145/3166089.3168733.

[13] Eric Harris-Braun, Nicolas Luck, and Arthur Brock. "Holochain: Scalable Agent-Centric Distributed Computing". Whitepaper. Ceptr LLC, Feb. 15, 2018. URL: https://holo.host/whitepapers/.

[14] Event Source. *Event Sourcing Basics*. URL: https://eventstore.org/docs/event-sourcing-basics/ (visited on 09/19/2019).

[15] *Event Store*. URL: https://eventstore.org/ (visited on 09/19/2019).

[16] Martin Fowler. *Event Sourcing*. URL: https://martinfowler.com/eaaDev/EventSourcing.html (visited on 09/19/2019).

[17] Seth Gilbert and Nancy Lynch. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services". In: *Acm Sigact News* 33.2 (2002), pp. 51–59.

[18] HashiCorp. *Key Rotation*. URL: https://www.vaultproject.io/docs/internals/rotation.html (visited on 09/20/2019).

[19] Tom Heath and Christian Bizer. "Linked Data: Evolving the Web into a Global Data Space". In: *Synthesis Lectures on the Semantic Web: Theory and Technology* 1.1 (Feb. 9, 2011), pp. 1–136. ISSN: 2160-4711. DOI: 10.2200/S00334ED1V01Y201102WBE001.

[20] A. Herzberg et al. "Access Control Meets Public Key Infrastructure, or: Assigning Roles to Strangers". In: *Proceeding 2000 IEEE Symposium on Security and Privacy. S P 2000*. Proceeding 2000 IEEE Symposium on Security and Privacy. S P 2000. May 2000, pp. 2–14. DOI: 10.1109/SECPRI.2000.848442.

[21] Jay Kreps. *The Log: What Every Software Engineer Should Know about Real-Time Data's Unifying Abstraction*. Dec. 16, 2013. URL: https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying (visited on 09/19/2019).

[22] Shmuel Katz and Doron Peled. "Interleaving Set Temporal Logic". In: *Theoretical Computer Science* 75.3 (1990), pp. 263–287.

[23] Martin Kleppmann. *Designing Data-Intensive Applications: The Big Ideas behind Reliable, Scalable, and Maintainable Systems*. " O'Reilly Media, Inc.", 2017.

[24] Martin Kleppmann and Alastair R. Beresford. "A Conflict-Free Replicated JSON Datatype". In: *IEEE Transactions on Parallel and Distributed Systems* 28.10 (Oct. 1, 2017), pp. 2733–2746. ISSN: 1045-9219. DOI: 10.1109/TPDS.2017.2697382. arXiv: 1608.03960.

[25] Sandeep S. Kulkarni et al. "Logical Physical Clocks". In: *Principles of Distributed Systems*. Ed. by Marcos K. Aguilera, Leonardo Querzoni, and Marc Shapiro. Vol. 8878. Cham: Springer International Publishing, 2014, pp. 17–32. ISBN: 978-3-319-14471-9 978-3-319-14472-6. DOI: 10.1007/978-3-319-14472-6_2. URL: http://link.springer.com/10.1007/978-3-319-14472-6_2 (visited on 09/19/2019).

[26] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Communications of the ACM* 21.7 (July 1, 1978), pp. 558–565. ISSN: 00010782. DOI: 10.1145/359545.359563.

[27] Paul J. Leach, Michael Mealling, and Rich Salz. *A Universally Unique IDentifier (UUID) URN Namespace*. July 2005. URL: https://

tools.ietf.org/html/rfc4122 (visited on 09/20/2019).

[28] Moxie Marlinspike. "The Double Ratchet Algorithm". In: Revision 1 (Nov. 20, 2016). Ed. by Trevor Perrin, p. 35. URL: https://signal.org/docs/specifications/doubleratchet.

[29] Martin Fowler. *CQRS*. July 14, 2011. URL: https://martinfowler.com/bliki/CQRS.html (visited on 09/20/2019).

[30] Aljoscha Meyer. *Bamboo*. Sept. 16, 2019. URL: https://github.com/AljoschaMeyer/bamboo (visited on 09/20/2019).

[31] Microsoft Corporation. *Azure Application Architecture Guide*. URL: https://docs.microsoft.com/en-us/azure/architecture/guide/ (visited on 09/19/2019).

[32] Yves-Alexandre de Montjoye et al. "On the Trusted Use of Large-Scale Personal Data." In: *IEEE Data Eng. Bull.* 35.4 (2012), pp. 5–8.

[33] Robert Mört. *Content Based Addressing : The Case for Multiple Internet Service Providers*. 2012. URL: http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-107174 (visited on 09/19/2019).

[34] Multiformats. *Multiaddr*. URL: https://multiformats.io/ (visited on 09/20/2019).

[35] D. S. Parker et al. "Detection of Mutual Inconsistency in Distributed Systems". In: *IEEE Transactions on Software Engineering* SE-9.3 (May 1983), pp. 240–247. DOI: 10.1109/TSE.1983.236733.

[36] Protocol Labs. "Filecoin: A Decentralized Storage Network". Whitepaper. July 19, 2017.

[37] Lum Ramabaja. "The Bloom Clock". In: (May 30, 2019). arXiv: 1905.13064 [cs]. URL: http://arxiv.org/abs/1905.13064 (visited on 09/19/2019).

[38] Redux. *Getting Started with Redux*. URL: https://redux.js.org/ (visited on 09/19/2019).

[39] Redux. *Motivation*. URL: https://redux.js.org/introduction/motivation (visited on 09/20/2019).

[40] Redux. *Reducers*. URL: https://redux.js.org/ (visited on 09/19/2019).

[41] Sean C. Rhea, Russ Cox, and Alex Pesterev. "Fast, Inexpensive Content-Addressed Storage in Foundation." In: *USENIX Annual Technical Conference*. 2008, pp. 143–156.

[42] Andrei Vlad Sambra et al. *Solid: A Platform for Decentralized Social Applications Based on Linked Data*. 2016. URL: http://emansour.com/research/lusail/solid_protocols.pdf.

[43] R. S. Sandhu et al. "Role-Based Access Control Models". In: *Computer* 29.2 (Feb. 1996), pp. 38–47. DOI: 10.1109/2.485845.

[44] Hector Sanjuan, Samuli Poyhtari, and Pedro Teixeira. "Merkle-CRDTs". Whitepaper. May 2019.

[45] Fred B. Schneider. "Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial". In: *ACM Computing Surveys (CSUR)* 22.4 (1990), pp. 299–319.

[46] Reinhard Schwarz and Friedemann Mattern. "Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail". In: *Distributed Computing* 7.3 (Mar. 1994), pp. 149–174. ISSN: 0178-2770, 1432-0452. DOI: 10.1007/BF02277859.

[47] Secure Scuttlebutt. *Scuttlebutt Protocol Guide*. URL: https://ssbc.github.io/scuttlebutt-protocol-guide/ (visited on 09/11/2019).

[48] 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/BDCloud.2014.24.

[49] Marc Shapiro et al. *A Comprehensive Study of Convergent and Commutative Replicated Data Types*. report. Jan. 13, 2011. URL: https://hal.inria.fr/inria-00555588 (visited on 09/19/2019).

[50] Robert W. Shirey. *Internet Security Glossary, Version 2*. Aug. 2007. URL: https://tools.ietf.org/html/rfc4949 (visited on 09/20/2019).

[51] Pyda Srisuresh and Matt Holdrege. *IP Network Address Translator (NAT) Terminology and Considerations*. URL: https://tools.ietf.org/html/rfc2663 (visited on 09/20/2019).

[52] N. Unger et al. "SoK: Secure Messaging". In: *2015 IEEE Symposium on Security and Privacy*. 2015 IEEE Symposium on Security and Privacy. May 2015, pp. 232–249. DOI: 10.1109/SP.2015.22.

APPENDIX A. EVENT NODE INTERFACE

```
// Node is the most basic component of a log.
// Note: In practice, this is encrypted with the Follow Key.
type Node interface {
    ipld.Node

    // Event is the Log update.
    Event() Event

    // Refs are node linkages.
    Refs() []cid.Cid

    // Sig is a cryptographic signature of Event and Refs
    // created with the Log's private key.
    Sig() []byte
}

// Event represents the content of an update.
// Note: In practice, this is encrypted with the Read Key.
type Event interface {
    ipld.Node

    // Header provides a means to store a timestamp
    // and a key needed for decryption.
    Header() EventHeader

    // Body contains the content of an update.
    // In practice, this is encrypted with the Header key
    // or the recipient's public key.
    Body() ipld.Node

    // Decrypt is a helper function that decrypts Body
    // with a key in Header.
    Decrypt() (ipld.Node, error)
}

// EventHeader contains Event metadata.
type EventHeader interface {
    ipld.Node

    // Time is the wall-clock time at which the Event
    // was created.
    Time() int

    // Key is an optional single-use symmetric key
    // used to encrypt Body.
    Key() []byte
}
```