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

Pick[1], Farmer[1], Sutula[1], ?Gozalishivili others?[2] and Hill[1]

[1] *www.textile.io*
[2] *other?*
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 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 large-scale digital datasets. 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.

Keywords: Databases; Event Sourcing; IPFS; IPLD
revised 29 September 2019

1. INTRODUCTION

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 [12]. 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/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 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 [7, 38], and some have identified user-siloed data as a solution [49]. However, none so far have addressed the problem of *how data should be collected*, by providing 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), that is designed to store, share, and host user-siloed datasets 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.

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 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 how they are designed in popular database management (DBMS) systems outside the IPFS context.

2.1. Data Synchronization

To model realistic systems, apps often need to map data between domain models and database tables, where the same data model is used to both query and update a database. To solve *synchronization* it is often helpful to handle updates on just the database and then provide the query and interfaces only later in a DBMS. 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 [52]: 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 [26].

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.*

2.1.1. 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 (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 event sourcing and command query responsibility segregation (CQRS) patterns [8], 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 [18], Apache Kafaka [3] and Samza [4], among others [28].

Definition 2.2. *(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 [35].*

Definition 2.3. *(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 [20]. The same principal means a log can be viewed as a mechanism to support an infinite number of valid state interpretations (see Section 2.1.2). In other words, with minimal conformity, a single log can model multiple application states [37].

2.1.2. 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 the use case or user interface they are building. Multiple (potentially very different) views can be built from the same underlying event log.

Definition 2.4. *(View). A (typically highly denormalized) 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.*

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.*

Views themselves are enabled by projections[3], 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 peers operating on the same stream of events have a consistent representation of the data.

[3]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 Section 3

2.1.3. *Eventual Consistency*

The CAP theorem [11, 21] 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. While an active area of research, designing systems with provable eventual consistency guarantees remains challenging [2, 57].

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 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 (Definition 2.1) and CRDTs (Definition 2.1.3).

A common minimum requirement for log synchronization across multiple peers is that the essential order of events is respected and/or can be determined [27, 53]. For these cases, logical clocks are a useful tool for eventual consistency and total ordering [30]. 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 Section 2.1.5).

2.1.4. *Logical Clocks*

In a distributed system with multiple peers (each with an independent clock) creating events, local timestamps can't be used to determine "global" event causality. Machine clocks are never perfectly synchronized [31], 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 [31] Schwartz [53], Bloom [44], and Hybrid variants [30], etc.), which use "counter"-based time-stamps 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 [51], which relies on properties of a Merkle-DAG to provide strict partial ordering between events. This approach does have its limitations however [51, 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*

CRDTs (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*[4]. 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 [13, 51] 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 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 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 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 [39].

[4]Though non-commutative CRDTs may require a specific ordering of events in certain cases [51]

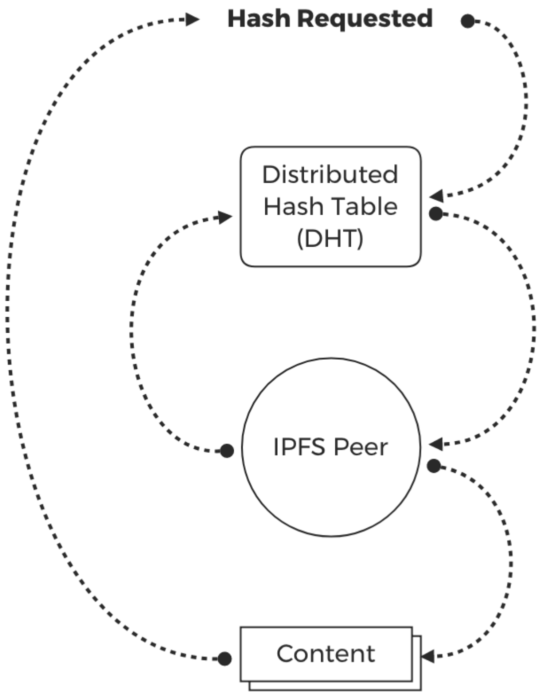

FIGURE 1: The cryptographic hash of content is used to make a request to the network of IPFS peers. Using the built-in routing mechanisms and a distributed hash table (DHT), peers hosting the requested content are identified and content is returned.

2.2.1. 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 [5, 48, 55]. 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 [5]. 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" [43]. 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/requested) and de-duplication (i.e., the same content from different peers will produce the same hash address). 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[5]) 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.

[5]`https://libp2p.io/`

2.2.2. IPLD

As discussed previously, IPFS uses the cryptographic hash of a given piece of content to define its content-based address (see [5] 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., 41] MORE NEEDED], this common data format is called Interplanetary Linked Data (IPLD)[6]. As the name suggests, IPLD is based on principals of linked data [6, 9] 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) [24] uses content-based addresses as unique identifiers. To form its structure, IPLD implements a Merkle DAG, or directed acyclic graph[7]. This allows all hash-linked data structures to be treated using a unified data model, analogous to linked data in the Semantic Web sense [10]. 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, or a log [51]. 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 [51, sec. 4.3]:

1. Sharing the Merkle-Clock can be done using only the *root* CID. The whole Clock is unambiguously identified by the CID of its root, 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

[6]`https://ipld.io/`
[7]Other examples of DAGs include the Bitcoin blockchain or a Git version history.

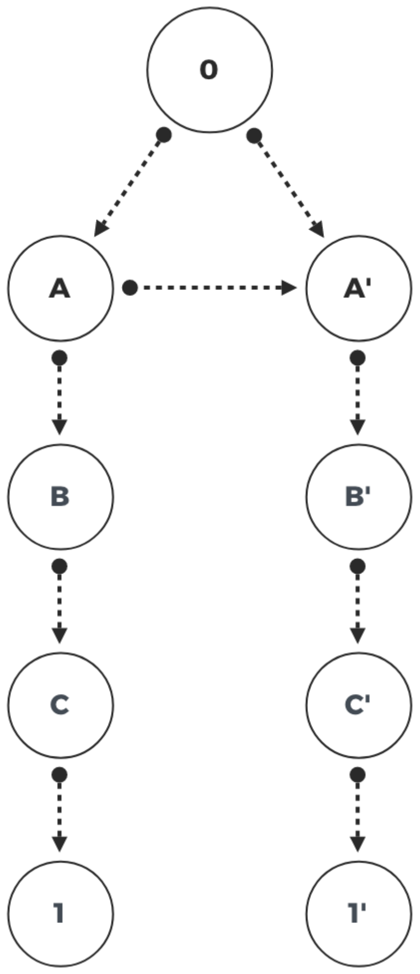

FIGURE 2: Divergent heads in a multi-writer Merkle-DAG

event.

On the downside (see also Section 2.1.4), a Merkle-Clock cannot order divergent heads (or roots). For example, in Figure 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 often 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 to synchronize replicas 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 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.

2.3.1. 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, File-Coin, 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 [59] (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 [42]. 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 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 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, also 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 these 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, such as using multi-modal communication strategies or leveraging tools such as deferred routing (e.g. inboxing) for greater tolerance of missed messages.

2.3.3. IPNS

Our discussion of Pubsub and libp2p has so far only dealt with *push-based* transfer of data; but IPFS also offers a useful technology for hosting *pull-/request-*based data endpoints based on an Interplanetary Name System (IPNS). IPNS aims to address the challenge of mutable data within IPFS. It relies on a global namespace (shared by all participating IPFS peers) based on Public Key Infrastructure (PKI). 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 the static route to which the IPNS address refers. But IPNS isn't only useful for creating static addresses pointing to content-based addresses; it is also compatible with external naming systems such as DNS, Onion, or bit addresses.

Unfortunately, many use-cases that require highly mutable data, also require rapid availability of updates, or need flexible multi-party access control, which is not currently viable using IPNS alone. However, taken together, libp2p, pubsub, IPNS, and IPFS more generally provide a useful toolkit for building robust abstractions to deliver fast, scalable, data synchronization on a decentralized network.

2.4. Data Access & Control

IPFS is an implementation of PKI, where every node on the 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 this public key is used to generate the node's Peer IDentity (Peer ID).

2.4.1. 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 [54] and Holochain [15], make use of agent-centric security. Each of these systems leverage cryptographic signatures to validate peer identities and messages.

TABLE 1: Example Access Control List.

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

2.4.2. 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 [58]. 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 Figure 1).

Using ACLs in systems where identity is derived from various configurations of PKI has been around for some time [25]. 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.

3. 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 queriable materialized views, and custom CRDTs.

In essence, 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 — Logs and Events — 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.

3.1. Event 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` [33]), a Merkle-Clock can be used to induce ordering. This approach cannot achieve a total order of events without implementing a data-layer conflict resolution strategy [51]:

> 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 the Merkle-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

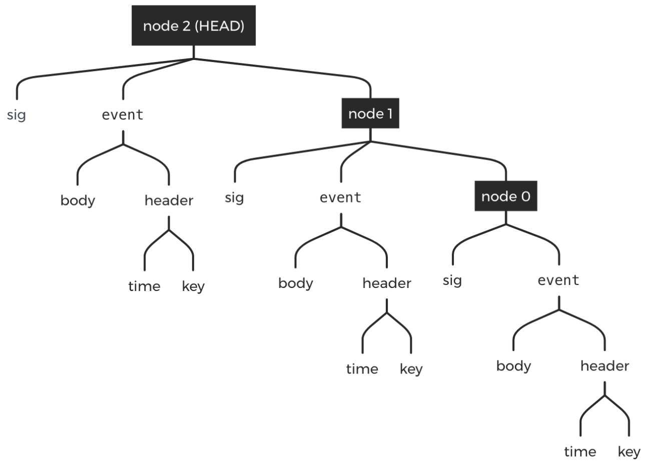

FIGURE 3: A single-writer Merkle-Clock (Event Log).

peers contributing to a dataset may not be interested in following or replicating the events of all other peers (e.g., in a Pubsub-based system).

3.1.1. Single-writer Event 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*. An Event Log is then a *single-writer Merkle-Clock* that can be totally ordered (Figure 3), and *separate* Event Logs can be composed into advanced structures, including CRDTs [13].

This presents a novel take on multi-writer systems [13], 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 actually 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 Section 2.1.4). Finally, even manual merge-type strategies are possible if this is the desired conflict resolution strategy.

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

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

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

Related, a Merkle-Clock (see Section 2.2.3) is simply a Merkle-DAG of *Events*.

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 [36].

As shown in 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 [42]. 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).

Unlike peer multiaddresses, Log addresses are not stored in the global IPFS DHT [5]. 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[8] like topology. Log addresses are recorded in an address book, similar to an IPFS Peer address book (see Example 2). 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.

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, Textile Logs are designed with a built-in replication or *Follower* mechanism. Followers are represented as additional addresses, meaning that a Log address book may contain *multiple* multiaddresses for a single Log (see Example 1).

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.*

[8]This is similar to the append-only message feeds used in Secure Scuttlebutt's global gossip network [54]

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 6.1.3), 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). As such, they are encrypted by default in a manner that enables access control (Section 5.5.2) and the Follower mechanism discussed in the previous section. Much like the Log address book, Log *keys* are stored in a key book (Example 2).

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 for each piece of content and never reused. The Content Key is distributed directly in the header of the Event Block.

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 for the Content Key of an 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 from all others[9].

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 via a separate (secure) channel to all Peers who require 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

[9]Much like private messages in Secure Scuttlebutt.

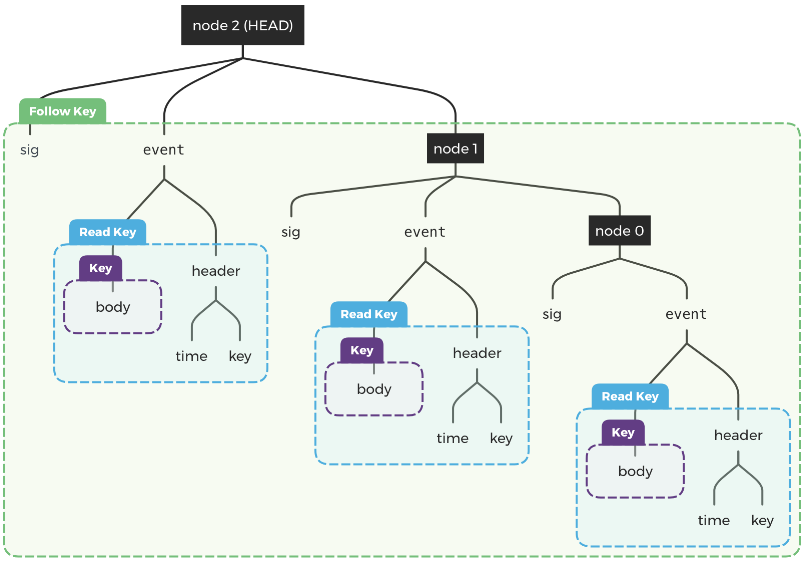

FIGURE 4: The three layers of Log Event encryption.

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. Followers 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.*

3.2. Threads

In Textile, the *interface* to Logs is managed as a Thread, which 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, followed, 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 [30]) or a CRDT (Section 2.1.5).

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 Pubsub-based synchronization. TIDs are defined with the format shown Figure 5.

TIDs share some similarities with UUIDs [32] (version and variant) and IPFS-based CIDs and are multibase encoded[10] for maximum forward-compatibility. Base32 encoding is used by default, but any multibase-supported string encoding may be used.

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

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-*

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

```
# 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.

FIGURE 5: Components of a Thread Identity.

$$\text{Thread ID} = \underbrace{\text{0x62}}_{\text{Multibase}}\ \underbrace{\text{0x01}}_{\text{Version}}\ \underbrace{\text{0x55}}_{\text{Variant}}\ \underbrace{\text{0x539bc1dc03ee8cb5d478e41cc8a4546e}}_{\text{Random Number}}$$

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 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.

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 5.5.2 and Example 3b).*

```
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.

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

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

Example 3: Identity variants.

3.2.3. Log Synchronization

Log Writers, Readers, and Followers synchronize the state of their Logs by sending and receiving Events. Inspired by Git[11], 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 for the given Log. This is similar to how a system such as OrbitDB [33] works, except we are tracking *multiple* Heads (one per Log), rather than a single Head.

Regardless of the network protocol, Events are transported between Peers in a standardized *Event Envelope*. 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 3.15. *(Event Envelope). An over-the-wire message containing an Event and the sender's signature of the Event.*

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 ACL (Section 5.5.2) 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).*

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

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. A Key Set is 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 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[12] 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 gossip-based Pubsub 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, 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.

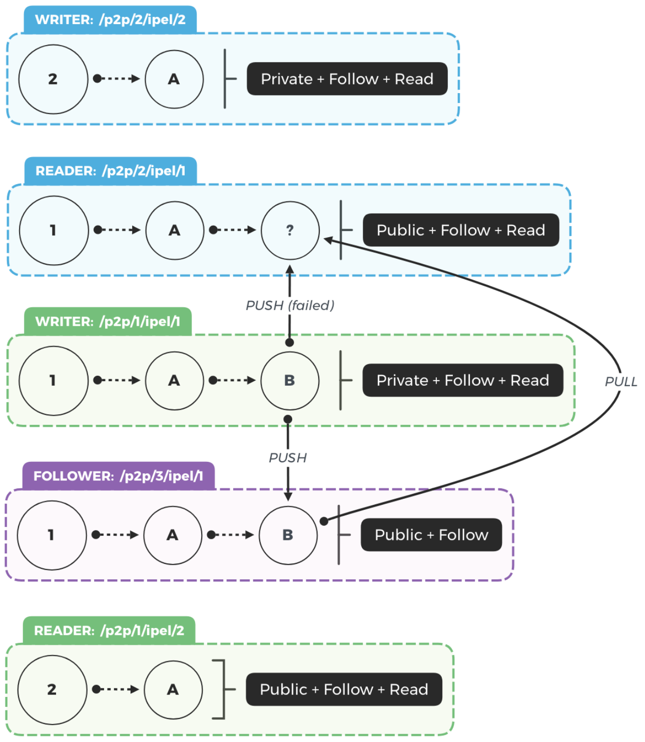

FIGURE 6: A pull-based request from a Follower.

1. Log Writers can receive Events directly from the Writer.
2. Events can be pulled from 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 on the network to be granted the responsibility of replicating data from another Peer without having read access to the raw 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).

4. THREADS INTERNALS

Previous sections have discussed the core features of the Textile Threads protocol. However, we have not

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

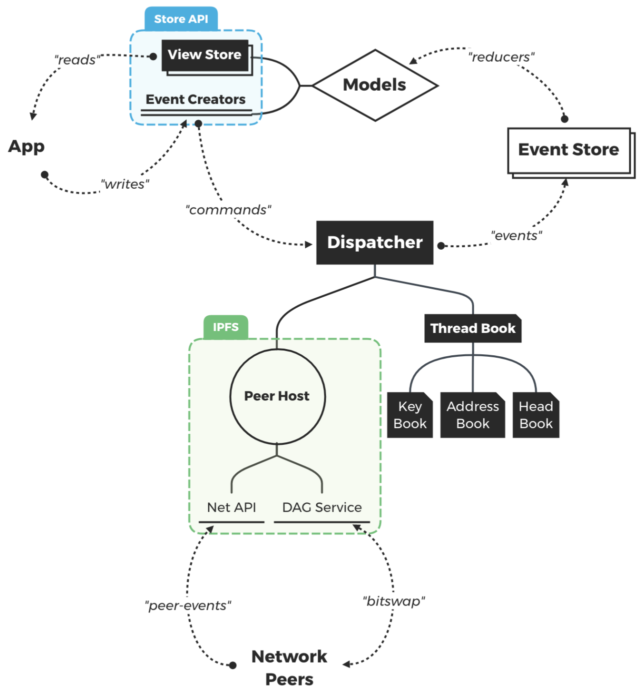

FIGURE 7: Architectural diagram for internal Event Store implementation.

yet discussed dealing with Log Events in practice. In this section, we provide a description of a Threads-compatible Event Store implementation. The Event Store outlined here takes advantage of ideas from several existing CQRS and ES systems (e.g., [14]), as well as concepts and designs from Flux [19], Redux[14] [46] and domain driven design [16] (DDD)[15]. Following this discussion of Threads *internals*, in Section 5 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.

4.1. 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 [19] and/or [1]). In both cases, the focus is on unidirectional data flows that build downstream views from atomic updates in the form of events (or actions).

We adopt a similar flow in Threads (see Figure 7). Like any ES-based system, Threads are built on *Events*. Events are similar to actions in a Flux-based system, and are used to produce *predictable* updates to downstream state. Similarly to a DDD-based pattern, to add an Event to the internal Threads system, we use *Event Creators*, which dispatch Events to the system via a singleton *Dispatcher*. The Dispatcher then stores the derived Event in an *Event Store*, and calls a set of registered *Reducer* functions that mutate a (set of) downstream view *Stores* defined by a corresponding set of view *Models*, all within a single *Transaction*. This unidirectional, transaction-based system, provides a flexible framework for building complex event-driven application logic.

4.1.1. Events & Creators

Like most CQRS and ES based systems, *Events* are at the heart of Threads. Every update to local and shared (i.e., across peers) application state happens via Events (see also Section 2.1.1). Events are used to describe "changes to an application state" [20] (e.g., a photo was added, an item was added to a shopping cart, etc). Related, *Event Creators* are used to send Events from a *local application* to an internal Event Store. If built around a specific *domain*, Creators provide bounded context that can be roughly compared to an aggregate root in DDD [16].

4.1.2. Dispatcher

In order to actually persist and dispatch Events to downstream view Models, a *Dispatcher* is used. As shown in Figure 7, the Dispatcher is at the center of the ES system. All Events must go through the singleton Dispatcher, whether these be from local Event Creators or Network Peers. The Dispatcher is responsible for ensuring that incoming Events are persisted to the Event Store (the "source of truth" for the system), as well as dispatched to downstream view Models by way of a set of Reducer functions.

4.1.3. Transactions

All Reducer function calls (and "side effects") due to a given Event happen within a single *Transaction* [22], in order to ensure consistently of both storage (Event Store) and Models. Transactions are similar to Redux *Sagas*[16] in terms of outcome, but with stronger consistency guarantees. Once an Event has been persisted in the internal Event Store, the Dispatcher is responsible for running a view Model's Reducer callback. In order to make this possible, Models must *register* their Reducer with the Dispatcher.

4.1.4. View Models/Stores

As in many CQRS-based systems, Events are dispatched on the write side, and are *reacted to* on the read side. Reactions happen via Reducer functions, which cause updates to view Models, which in turn provide interfaces for queries and accessing persisted view *Stores* (state). View Stores are then responsible for notifying downstream consumers (application logic) of changes to

[14]Redux builds on concepts from CQRS and ES itself, and is arguably an implementation of the Flux application architecture.
[15]`https://dddcommunity.org`

[16]`https://redux-saga.js.org/`

their state via a *Broadcaster* (i.e., event emitter). They are similar in some respects to a Redux Store, though it is possible to have *multiple* view Models/Stores as in the more general Flux pattern.

A view Model is generally defined by custom update logic (i.e., a Reducer), a (possibly ORM[17]-based) view Store for persistence, an Event Bus (or Broadcaster) for notifying downstream consumers, and a set of query/*Resolver* [14] functions. In practice, a view Model may *also* wrap the Event Creator application logic that is used to generate upstream Events. This provides an intuitive, singular access point to the *local* system, while also leaving room for updates via *external* Peer Events. In practice, the Store is a lightweight interface that can be implemented by one of many database management systems (Appendix B) providing end users/developers the greatest flexibility.

4.1.5. External Events

In addition to locally derived Events (i.e., from application logic and user interactions), Threads are designed so that Peers may collaborate in a given Thread via Events. Events generated by other, network peers are called Peer Events, and they enter the system via a Peer Host. The Peer Host is responsible for dealing with incoming Events (be they push or pull). These Events are no different from locally-derived Events, though in practice the Peer Host is required to validate incoming Events before they are dispatched to the system via the Dispatcher.

5. THREAD INTERFACES

To make Threads as easy to adopt and use as possible, Textile has 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. These simple, public-facing APIs 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 a Thread 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 pseudo-code for a hypothetical Photos app.

```
Photo = NewModel({
  _id: UUID,
  thumbnail: Buffer,
  original: Buffer,
});

Contact = NewModel({
  _id: UUID,
  name: {type: String, index: true},
  avatar: Photo,
});

// Photos may be grouped into messages.
Message = NewModel({
  _id: UUID,
  author: Contact,
  body: String,
  photos: [Photo],
})
```

Example 4: Create a photo entity and some way to represent a photo's author.

5.1. Illustrative Example

To create a useful application, developers start with view `Models`, as in Example 4. A Model is essentially the public API for the underlying view Models from Section 4.1.4). Building on this, developers might create a new Thread for a user to store `Contact` information, as well as their mobile phone's camera roll photos, as in Example 5. This would create a new view Store "under-to-hood" (with corresponding indexes, etc), to be mutated by incoming Events.

The next step is to actually *create* and add a `Message` object to a shared album. In example 6, a message instance is created via the custom `Message` class, and then added to the shared `Dogs` Thread (which represents an album here). Behind the scenes, the Model (which is providing an Event Creator interface) is internally responsible for dispatching the Event through the local Dispatcher.

An example Event is given in Example 7, and is the result of a new Message Event. Now, any updates to an existing Message instance will automatically generate the required underlying update Events. For example, Example 8 shows the body text of the previous example being updated, and saved (committed) to the Thread. Behind the scenes, this event will be added to the User's local Log, and pushed to any peers identified in the Thread's ACL document (see Sections 3.2.2 and 5.4). In practice, the Model generates another Event that carries the diff and a document identifier.

[17]Object-relational mapping
[18]`http://www.mongodb.com`
[19]`https://parseplatform.org`
[20]`https://realm.io`

```
AddressBook = NewDocumentStore("AddressBook")
// This store should take Contacts.
AddressBook.AddModel("Contact", Contact)
// If needed, additional models can be added...

// Create another store for camera roll photos.
CameraRoll = NewDocumentStore("CameraRoll")
// This store should take photos.
CameraRoll.AddModel("Photo", Photo)

// Create another store for a shared album.
MyDogsAlbum = NewDocumentStore("Dogs")
MyDogsAlbum.AddModel("Message", Message)

// Messages can also be nested.
Message.AddModel("Message", Message)
```

Example 5: Create address book and camera roll stores. This will create new Threads.

```
// Create a message with a photo.
MyMessage = Message.Create({
  author: <author_id>,
  body: "This is Lucas.",
  photos: [{
    thumbnail: <buffer>,
    original: <buffer>
  }]
})
// Now it can be added to Dogs "album".
MyDogsAlbum.Add(MyMessage)
```

Example 6: Adding data to a shared Thread.

```
{
  "body": {
    "data": {
      "author_id": <author_id>,
      "body": "This is Lucas.",
      "photos": [{
        "thumbnail": <buffer>,
        "original": <buffer>
      }]
    }
  },
  "header": {
    "time": 1569434034737,
    "key": "215bs...1DXJ"
  }
}
```

Example 7: A new Message Event.

```
MyMessage.body = "Actually, this is Fido."
MyMessage.Save()

// New Event with diff and document id
{
  "body": {
    "data": {
      "doc_id": MyMessage._id,
      "body": "Actually, this is Fido."
    }
  },
  "header": {
    "time": 1569434035737,
    "key": "iJMfqWy...1qfJyc29RS"
  }
}
```

Example 8: Message updates are persisted and transmitted automatically.

All instances and models in Threads have several special methods and properties specific to the Threads API, several of which we will explore here. Example 9 demonstrates several features common in a Threads-based workflow, including queries, creating invites and changing permissions/access control (see also Section 5.5.2), as well as subscribing to updates and changes at various levels of the Threads API. These subscriptions would enable downstream consumers (views, front-end stores, etc.) to receive updates as changes to the Thread are made via underlying Events.

5.2. 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 data Module 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 the Photos Module (which may be designed as in the example above). They may also extend this 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

```
// Query for message, select only thumbnail.
Dogs.FindOne({ "_id": MyMessage._id },
    "thumbnail")

// Every (Event increments version tag
MyMessage.Version()

// All doc changes in Dogs Thread
Dogs.subscribe()
// All changes to all Messages in all Threads
Message.Subscribe()
// Changes specific to this document
MyMessage.Subscribe()

// Create invite Event IFF User has permission
Dogs.Grant(<peer_id>, <role>)

// Alter ACL OR create invite if needed/allowed
MyMessage.Grant(<peer_id>, <role>)
```

Example 9: Additional Threads-based API functionality.

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 Models. 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* [56] produced by many interoperable apps.

5.3. Databases

With these interface simplifications, it is not difficult to imagine even higher-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 [33]) 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 an internal (i.e., private) Model as in the previous section, with similarly defined methods. The generated Events would then mutate the internal (map-like) view Model effectively encapsulating the entire Event Store in a database structure that satisfies a given interface (see Figure 10 for example). These too would be distributed as Modules, making it easy for developers to swap in/substitute existing backend infrastructure.

Other database abstractions include a no-sql style document store for storing and indexing arbitrary

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

Example 10: 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 11: The Document store interface.

structs and/or JSON documents. The interface for such as store, again built using a "wrapped" view Model, might look like Figure 11, where `Indexable` could be satisfied by any structure with a `Key` field and `Query` might be taken from the `go-datastore` interface library[21] or similar.

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. 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 *reference* libraries.

5.4. CRDTs

Eventually consistent, CRDT-based structures can also be implemented on top of Threads' Event-driven architecture. 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 [29] that merges updates to a JSON document in a view Model's Reduder function. Libraries such as Automerge[22] provide useful examples of reducer functions 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 Threads is given in section 5.5.2, where it is used to represent updates to an ACL document defined as a default view

[21] `https://github.com/ipfs/go-datastore`
[22] `https://github.com/automerge/automerge`

Model, with interfaces defined for an access-controlled Threads implementation.

5.5. Thread Extensions

The Textile protocol provides a distributed framework for building shared, offline first, Stores that are fault tolerant[23], 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.

5.5.1. Snapshots and Compaction

Snapshots[24] 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 using the Model'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[25].

In practice, Snapshots are written to their own internal Event Store 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 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 Event 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.

5.5.2. 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, Entity-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, Entity-level ACLs provide more granular control to Thread-writers on a per-Entity (see Definition 12) basis. Both types of ACLs are implemented as JSON CRDTs (see Section 5.4) wrapped in a custom view Model (see Section 5). ACLs implemented as JSON Models provide 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 essentially mutable JSON documents, they can specify their *own editing rules* (i.e. allowing multiple Thread participants to modify the ACL) in a self-referencing way.

Definition 5.1. *(Entity). An Entity is made of of 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 that describe a JSON document. An Entity might have a unique UUID (see Example 12) which can be referenced across/within Event updates.*

Textile's Threads includes ACL management tooling based on a *Role-based access control* [50] 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[26]: *No-access, Follow, Read, Write, and Delete.*

Definition 5.2. *(No-access). No access is permitted. This is the default role.*

Definition 5.3. *(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 5.4. *(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 5.5. *(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,*

```
# UUID
bafykrq5i25vd64ghamtgus6lue74k
```

Example 12: Sequence ID.

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

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

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

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

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

Example 13: ACL JSON document with `_id` being the unique ID.

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

Example 14: Thread and document ACL

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 5.6. *(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 13) can be persisted to a local Event Store as part of the flow described in Section 4. See also Section 5, and in particular Example 9 for the public API for editing ACL definitions.

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 14).

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

6. 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 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 view Models, 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.

6.1. 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.

6.1.1. Enhanced Log Security

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 [23], and/or incorporating the Double Ratchet Algorithm [34] for forward secrecy [60].

6.1.2. Tighter Coupling with Front End Models

Implementing Threads internals (see Section 4) using similar patterns to common frontend workflows (e.g., Redux) presents opportunities for tighter coupling between "backend" logic and fontend views. This is a major advantage of tools such as reSolve[27], where "system changes can be reflected immediately [on the frontend], without the need to re-query the backend" [14]. Textile developers will create frameworks to more directly expose the internals of Threads to frontend SDKs (or DMBS, see Appendix B), making it possible to sync application state across and between apps and services on the IPFS network.

[27] `https://reimagined.github.io/resolve/`

6.1.3. 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 Follow Key now available in Threads, Bots can relay, replicate, or store data that is synchronized via real-time updates in a *trustless*, 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.

ACKNOWLEDGEMENTS

Thanks to our funders and the excellent Textile community for helpful suggestions, feature requests, and reviews. The work presented here has also benefitted from numerous discussions with members of the decentralized web community.

REFERENCES

[1] Dan Abramov. *The Case for Flux.* Nov. 3, 2015. URL: `https://medium.com/swlh/the-case-for-flux-379b7d1982c6` (visited on 09/23/2019).

[2] 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`.

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

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

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

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

[7] 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).

[8] 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.

[9] 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.

[10] Brendan O’Brien and Michael Hucka. “Deterministic Querying for the Distributed Web”. Whitepaper. Nov. 2017. URL: `https://qri.io/papers/deterministic_querying/`.

[11] 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`.

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

[13] 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`.

[14] Roman Eremin. *A Redux-Inspired Backend.* Jan. 14, 2019. URL: `https://medium.com/resolvejs/resolve-redux-backend-ebcfc79bbbea` (visited on 09/24/2019).

[15] 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/`.

[16] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley Professional, 2004.

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

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

[19] Facebook. *Flux: In-Depth Overview.* 2019. URL: `http://facebook.github.io/flux/docs/in-depth-overview` (visited on 09/23/2019).

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

[21] 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.

[22] Theo Haerder and Andreas Reuter. "Principles of Transaction-Oriented Database Recovery". In: *ACM Comput. Surv.* 15.4 (Dec. 1983), pp. 287–317. ISSN: 0360-0300. DOI: 10.1145/289.291.

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

[24] 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.

[25] 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.

[26] 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).

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

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

[29] 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.

[30] 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).

[31] 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.

[32] 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).

[33] Mark Robert Henderson et al. "The OrbitDB Field Manual". Guide. Haja Networks Oy, Sept. 26, 2019. URL: https://github.com/orbitdb/field-manual (visited on 09/26/2019).

[34] 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.

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

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

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

[38] 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.

[39] 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).

[40] 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.

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

[42] Protocol Labs. *Multiaddr.* URL: https://multiformats.io/ (visited on 09/20/2019).

[43] Protocol Labs. *Multihash.* URL: https://multiformats.io/ (visited on 09/27/2019).

[44] 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).

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

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

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

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

[49] 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`.

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

[51] Hector Sanjuan, Samuli Poyhtari, and Pedro Teixeira. “Merkle-CRDTs”. Whitepaper. May 2019.

[52] 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.

[53] 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`.

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

[55] 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`.

[56] Carl Shapiro, Shapiro Carl, and Hal R. Varian. *Information Rules: A Strategic Guide to the Network Economy*. Harvard Business Press, 1998.

[57] 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).

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

[59] 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).

[60] 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
}
```

APPENDIX B. DATABASE MANAGEMENT SYSTEMS (DBMS)

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