LoFiRe: Local-First Repositories for Asynchronous Collaboration over Community Overlay Networks

System Design & Protocol Specifications

TG x Thoth

Draft, Jan 2023

Table of Contents

1 Introduction

LoFiRe is a decentralized, local-first [11] application platform built on *local-first data repositories* and a *local-first network architecture*. Data repositories offer end-to-end encrypted data storage with public key authentication, access control, and change validation. Repositories store operation-based Conflict-free Replicated Data Types (CRDTs) [3] that are replicated with Byzantine Fault Tolerance & Strong Eventual Consistency guarantees [7]. Transactions with operations are organized in Directed Acyclic Graphs (DAG) with cryptographic authentication. The asynchronous network relies on store-and-forward multicast communication that can be provided by either Peer-to-Peer (P2P) publish-subscribe (pub/sub), Delay/Disruption Tolerant Networking (DTN), or mix networks. Nodes in the network rely on end-to-end encrypted object storage and end-to-end encrypted publish-subscribe (pub/sub) notifications with location privacy for end-user devices.

LoFiRe is built on local-first data storage, synchronization, and change notification protocols that aim to protect privacy by minimizing metadata exposed to intermediaries. It enables local-first, asynchronous collaboration and data storage within communities while respecting privacy and maintaining data ownership, and provides foundations for developing local-first decentralized applications and community overlay protocols.

Community members use local-first software to collaborate around a partition-tolerant, permissioned, tamper-proof data repository that contains branches of Directed Acyclic Graphs (DAG) of causally related transactions with operations on CRDTs.

In other words, a branch of the repository is a permissioned, DAG-structured distributed ledger, or blockchain, with partially ordered CRDT transactions. CRDTs require only a partial order on transactions, there's no need to determine a total order using a consensus protocol, which makes the protocol light-weight and low on resource use.

The DAG encodes a partial order of transactions through causality relations, and together with a reliable, causal publish-subscribe (pub/sub) protocol for change notifications and a DAG synchronization protocol, it provides strong eventual consistency of replicas, with persistence of transactions through a lightweight, quorum-based acknowledgement mechanism.

Each repository is synchronized within a private community overlay network that offers immutable block storage, data synchronization, and asynchronous publish-subscribe change notification services.

The two-tier network architecture consists of a stable core network with an overlay network for each repository with a low-latency P2P pub/sub protocol or mix network, and ephemeral edge networks that use P2P or DTN protocols for synchronization. On edge

networks, edge nodes can synchronize locally and directly between each other, and can also connect to designated core nodes that store and forwards encrypted objects and change notifications for them, with such a core node acting as a pub/sub broker and object store for edge nodes.

The system is composed of the following components:

Repository Data structures, encryption, permissions, authentication and access control.
Network Data synchronization, publish-subscribe change notification.
Applications CRDT state machine & change validation.

In the following we describe the repository structure and network protocols, as well as the interfaces for applications to interact with the system.

1.1 Requirements

We set the following requirements to guide our design:

Local first Store data locally, use local networks when available, and allow working even offline.c
Decentralized No centralized services are required to access and modify data.
Asynchronicity & partition tolerance Allow collaboration between users even if they are not online at the same time.
Identity & data ownership Users should have full control over their data and identities.
Causality Changes have causal relationships and are delivered in causal order.
Authenticity Changes are signed by their author.
Tamperproof Once a transaction is stored in a branch, it cannot be removed, except by forking the branch.
Access control Rules that specify which users can perform what operations on which part of the data.
Strong eventual consistency Replicas reach the same state after receiving the same set of changes.
Privacy Respect user privacy and minimize the amount of user data and metadata exposed to intermediaries.
End-to-end security Only the intended recipients should be able to read data stored and transmitted in the network.
Ephemerality Data can have expiration date after which it should be deleted from all replicas.
Multiple devices per user Allow users to use multiple devices, sync data, and share an identity among them.

Local-first software [11] stores data locally, allows offline data access and modification, and uses the network for data synchronization with local and remote nodes. Conflict-free Replicated Data Types (CRDTs) allow concurrent changes and conflict-free merges of a shared data structure.

Local-first networking allows local collaboration on edge networks among local nodes on the LAN or other kind of proximity network, while also allows remote collaboration over the internet when needed.

Data locality ensures control over data location and distribution, such that data shared in a repository remains private and only distributed to authorized peers. Data should not be dispersed over the network to arbitrary nodes, even in encrypted form, and nodes should only store and forward data they are interested in, providing incentives for participating in the network, and limiting damages in the event of an encryption key or algorithm compromise.

End-to-end security ensures only the intended recipients can decrypt data, i.e. intermediary nodes used for routing and storage cannot see the contents of communications, which is only visible to end-users who hold encryption keys on their personal devices.

Identity and data ownership entails affording users full control over their data and identities, by using cryptographic identities and allowing them to store local copies of their data and private keys. This is not possible in centralized systems that use centralized identifiers such as user names or phone numbers, neither in federated communication services that use DNS-based identities tied to a provider and keep user data on the provider's servers with no easy way for users to acquire a full copy of their data, or to migrate their data and identities to another service provider.

Asynchronicity is a common requirement for communication, and it's provided by both centralized (e.g. CryptPad, Nextcloud) and federated services (e.g. SMTP, XMPP, Matrix). However P2P systems often fail to offer this functionality and only provide synchronous means of communication or implement asynchronicity by storing data at arbitrary nodes in the network that violates the data locality requirement (e.g. libp2p, Holochain, GNUnet).

1.2 Related work

Vegvisir [6] proposes a partition-tolerant permissioned DAG blockchain with a similar structure to ours, but it is not encrypted, has a single branch, and it is synchronized via a higher latency pull-based gossip protocol rather than lower latency push-based pub/sub.

Automerge [9] uses operation-based CRDTs and a Bloom-filter-based DAG synchronization algorithm [[10]][8] and provides a JSON data model, however it is missing encryption and a permission system, and uses a synchronous P2P model. Their DAG synchronization algorithm is used in our design for branch synchronization. Its CRDT state

machine and JSON data model are complementary to the design presented here, and can be used in the application layer.

Hyper Hyper Space [4] uses immutable append-only Merkle-DAGs together with operation-based CRDTs, and a gossip-based P2P synchronization protocol for synchronization of JSON objects, with a permission system and cryptographic identities. The network is a single-tier P2P system with browsers and headless nodes synchronizing data through a gossip protocol. Storing private user data in browsers with complex codebases prone to security vulnerabilities is a security risk by itself, since nodes store data unencrypted. User data is thus also visible to third-party service providers, which are necessary for asynchronous communication when participants are not online at the same time. Permission revocations are not final, i.e. new operations with past dependencies can show up that have to be undone by an admin by adding and undo entry to the DAG, which becomes problematic especially with multiple admins with diverging histories, and malicious nodes that can cause the history to grow indefinitely.

Holochain [5] has an agent-centric approach to collaboration that does not require total ordering. It uses per-user append-only logs (Source Chains) that are stored in an application-specific overlay using a validating DHT, they cannot fork and do not allow the use of multiple devices per user. The single-tier overlay relies on synchronous communication between peers that requires connecting to a number of online peers to retrieve DHT records and send messages, and in case of a network partition certain DHT entries may become unavailable, and messages cannot be sent to offline peers. Data locality within an overlay is determined by the DHT algorithm and is not correlated with interest in that piece of data, and data deletion is not supported. It does not provide Strong Eventual Consistency, Byzantine Fault Tolerance, and partial ordering due to the lack of causal dependencies across user chains. The hosting system do not support end-to-end encryption by default, and do not allow controlled data locality as it relies on an open market of hosting providers, rather than users choosing which hosting providers they want to host their data at.

In contrast, LoFiRe focuses on privacy, security, and composability. It stores user data end-to-end encrypted, supports object expiry and deletion, branch forking, and snapshots for compacting operations, as well as a quorum-based operation acknowledgement mechanism. Permission revocations are implemented via forking, and are final, and support read access revocation, too. The system is composable: transactions are entirely application defined, including the data format and CRDT types used, and thus can be used together with application-defined CRDT types and libraries. Similarly, the immutable object store supports arbitrary binary data, it is not restricted to JSON. Each repository uses a two-tier P2P network with low-latency asynchronous pub/sub, which saves resources on end-user devices and offers them location privacy.

Radicle [2] is a decentralized code collaboration network with Git support that uses public keys for identifying repositories and users, and employs a gossip protocol for repository replication. As it is intended for code collaboration, it relies on manual merges and

does not use CRDTs that would allow generic collaboration for other use cases, not just code.

Distributed Mutable Containers (DMC) [13] stores CRDT operations in an encrypted immutable object store, ERIS [14]. Operations do not have any causal dependencies among each other, thus partial ordering of operations is not possible, in absence of which nodes can only rely on unsynchronized absolute timestamps when trying to establish an ordering, which cannot be trusted. Also, there's no guarantee that if a user receives an operation then all users will receive it, either because of network issues or a censorship attempt.

ERIS [14] is a content-addressed convergent encryption scheme that uses similar cryptographic primitives, however, in our design block size is defined by the application layer, and padding is only added in the transport protocol and on-disk storage, it is not part of content-addressed objects, making the protocol more efficient in terms of bandwidth and disk usage. Another difference in our design is that storage nodes can traverse (but not decrypt) the Merkle tree to be able to return an entire subtree in response to a request, and set an expiry time for an entire tree while keeping data deduplicated.

LoCaPS [12] is a localized causal pub/sub protocol that uses LoCaMu [15], a localized reliable causal multicast protocol. We use a pub/sub subscription and event routing protocol inspired by LoCaPS, but applied to topic-based pub/sub instead of content-based, only forwarding authenticated events received from publishers.

2 Protocol design

2.1 Data repositories

A data repository contains branches, each with a Directed Acyclic Graph of commits linked by causal dependencies among them. See fig. 1 for an example.

The repository definition is the root of trust for the repository, and contains a list of published branches. Similarly, the branch definition is the root of trust for the commits in a branch. It specifies the permissions and validation rules that every replica has to follow when applying commits for a given branch.

Each commit refers to a list of earlier commits it depends on, and it is signed by its author, that we also refer to as the publisher. CRDT operations are grouped in a transaction, which is then placed inside a commit. Edge nodes of users validate all incoming commits according to the permissions and validation rules set in the branch definition, and apply the changes to their local replica of the repository only if they are valid. Valid commits are acknowledged by a quorum of publishers in subsequent commits. The causal dependencies between commits form a DAG that enables partial ordering of operations, and provide strong eventual consistency guarantees of valid commits in a branch.

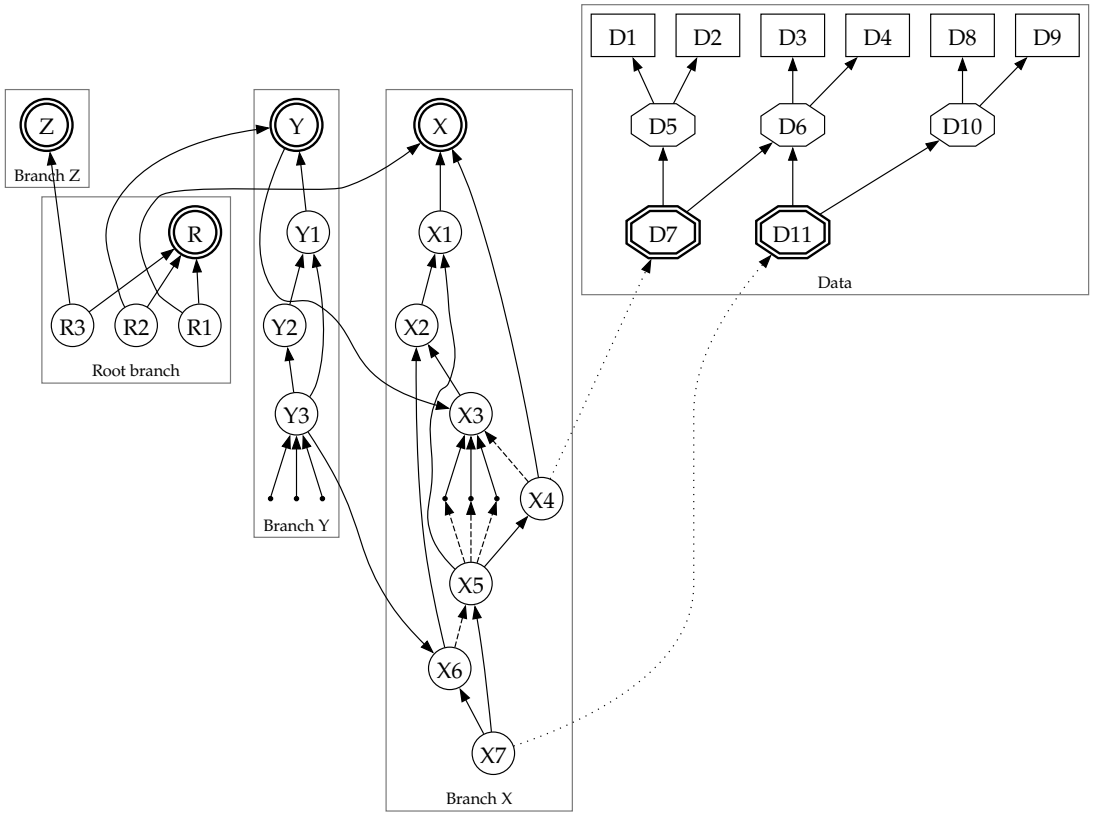

Figure 1: Repository R with branches X, Y, Z, commits R, *X*, Y, *acks of X3, Y3, and data D*. Branch Y was forked from X3, and Y3 merges X6. *deps* are solid lines, *acks* are dashed, *refs* are dotted.

Public keys are used to identify repositories, branches, and members, which allows the owner of a repository to grant permissions for adding & removing branches, and the owner of a branch to grant permissions for publishing commits.

The repository structure and storage objects are specified in the Data structures section.

2.1.1 Data storage

Objects are chunked to max. 2 MB chunks and stored encrypted in blocks of a Merkle tree. Merkle tree nodes use convergent encryption with a convergence secret that allows deduplication inside the repository, while defending against confirmation attacks of stored objects.

Both commits of a branch and files outside branches are stored in immutable objects. A `Block` contains either a leaf node of the Merkle tree with an encrypted data chunk, or an internal node with an unencrypted list of `BlockId` references to child nodes in the Merkle tree, and an encrypted list of keys for the referenced children. The unencrypted block header at the root of the Merkle tree further contains an optional list of dependencies of the object, which is used to list dependencies and acknowledgements of commit objects to allow efficient DAG synchronization, and an optional expiry time when replicas should delete the object, enabling ephemeral object storage.

Blocks are stored in a content-addressed immutable object store where the block ID (`BlockId`) is the BLAKE3 hash of the entire serialized block, and the object ID (`ObjectId`) is the ID of the root block of the Merkle tree of the object. In order to be able to decrypt and read the content of a block or a whole object, the user needs not only the `BlockId` or `ObjectId` (to retrieve it) but also the block or object key. These are combined in a `BlockRef` or `ObjectRef`, which enables its holder to retrieve and read the referenced block or object. To the contrary, sharing only the ID of a block or object does not provide a read capability for the recipient.

Implementations should employ an additional layer of encryption and padding before storing objects on disk in order to provide metadata-protection for data at rest, i.e. to hide stored block IDs and block sizes. Blocks can be packed together into larger storage blocks before applying padding in order to reduce storage space requirements.

2.1.2 Branches & Commits

The repository is organized in branches, which consist of a DAG of commits that contain transactions of CRDT operations. Commits are stored encrypted objects, the same way as files. The commit body is stored as a separate object, in order to be able to reference it directly, and to allow deduplication of commit bodies.

The first `Commit` in a branch contains the `Branch` definition that specifies the commit validation rules and permissions for the branch, as well as the public key for the pub/sub topic used for publishing commits of the branch. A member list defines the public keys of members and their permissions for publishing commits in the branch. Each `Commit` references other commits it directly depends on (via `deps`) and acknowledges non-dependent branch heads (via `acks`) known by the publisher in order to reduce branching in the DAG, and may reference files it depends on (via `refs`). The referenced dependencies (`deps`) and acknowledgements (`acks`) in a commit implies that the publisher has validated those commits and all their dependencies recursively. The object IDs (but not the keys) of `deps` & `acks` of the `Commit` are listed in the unencrypted `Block` header in order to allow storage nodes to traverse branches for efficient synchronization, without letting them to decrypt and read the content of the objects.

To ensure durability of commits, a quorum of acknowledgements is needed before a commit can be considered valid. The quorum is specified in the branch definition, and is calculated as the number of commits depending on, or acknowledging the commit either directly or indirectly. Acknowledgements can be either implicit (dependencies and acknowledgements listed in a subsequent commit) or explicit (a `CommitAck` is sent, with a list of dependencies). Explicit acknowledgements are only needed if a commit hasn't been acknowledged implicitly within a certain time frame from reception, which is specified in the branch definition (`ackDelay`).

The end of a branch is marked with an `EndOfBranch` commit, after which no more commits are accepted, except the acknowledgements of this last commit. This commit can reference a fork where the branch continues. This allows for changing the branch definition, migration to a new data schema, history truncation by making a new snapshot that does not depend on any previous operations (compaction), changing permissions, and removing access from past members by excluding those members from the newly encrypted branch definition.

A forked branch can either depend on the previous branch with the heads listed in its `deps`, or on a `Snapshot` listed in its `refs`. A `Snapshot` contains the data structures resulting from applying all the operations contained in the commits reachable from the heads the snapshot refers to. A `Snapshot` can also be part of a `Commit` for the purpose of exporting the current branch state but not for compaction, in which case other publishers validate it via `acks`, but no other commits would depend on it.

The branch definition also allows adding application-specific metadata about each member and about the branch itself. This enables applications to specify roles and permissions for members (who can change which part of the data structure), define custom validation rules for commits in the branch (or reference an executable validation code that runs in a sandboxed environment, such as WebAssembly), and reference an application that provides a user interface to display and interact with the data.

The branch definition cannot be changed after it has been commited, except for adding new members and adding new permissions for an existing member, along with an ad-

justed `quorum` and `ackDelay`, via the `AddMember` commit. For any other changes the branch needs to be forked. After forking, the new branch can either depend on the commits of the old branch, or the branch owner can make a snapshot and compact the commits of the previous branch and thus remove dependencies on earlier commits, after which the commits of the previous branch can be garbage collected.

The root branch is identified by the public key of the repository, and has a pub/sub topic assigned that corresponds to the overlay ID. The root branch contains the `Repository` definition, which references the branches available in the repository. Branches can be added and removed via the `AddBranch` and `RemoveBranch` commit types. After a branch is removed, its commits can be garbage collected, unless it was replaced with a new branch that depends on commits from the removed branch. When forking the root branch, the fork reference must be present unencrypted in the `EndOfBranch` commit, and the forked branch must depend on the previous branch, since the root branch serves as the entry point to the repository, and applications need to be able to find the latest fork of it. It is not possible to publish commits containing transactions in the root branch, as the root branch is only used for managing the other branches. A branch can remain private if it is never added to the root branch.

2.1.3 Deletion of data

Data deletion is possible by setting an expiry time for the storage objects. The expiry field and child node references are not encrypted in order to allow storage nodes to perform garbage collection without being able to decrypt object contents. In case of expiring commits, in order to keep the DAG intact, the commit object itself never expires, only the object in the commit body, and all commits depending on an expiring commit must expire at the same time as, or earlier than the one they're depending on.

Data that has been removed by a commit remains in the branch, since all commits are kept in the branch. Permanently deleting data from a branch is possible by making a snapshot, compacting the operations from all the commits in the branch, then ending the current branch with an `EndOfBranch` commit, and creating a forked branch that depends only on the snapshot. After this the old branch can be removed from the published list of branches in the root branch, and its commits can be deleted at the expiry time set in the `EndOfBranch` commit.

2.1.4 User repositories

Each user has a repository shared among their devices that stores the addresses and encryption keys for repositories, branches, and user identities, as well as brokers to connect to for each repository. This allows a user to share configuration, data and identities across their different devices.

To access this repository, the Argon2 key-derivation function is used to derive the encryption key from a password. This key is used to encrypt a `RepoKeys` data structure that contains the private key for the user repository.

2.2 Network architecture

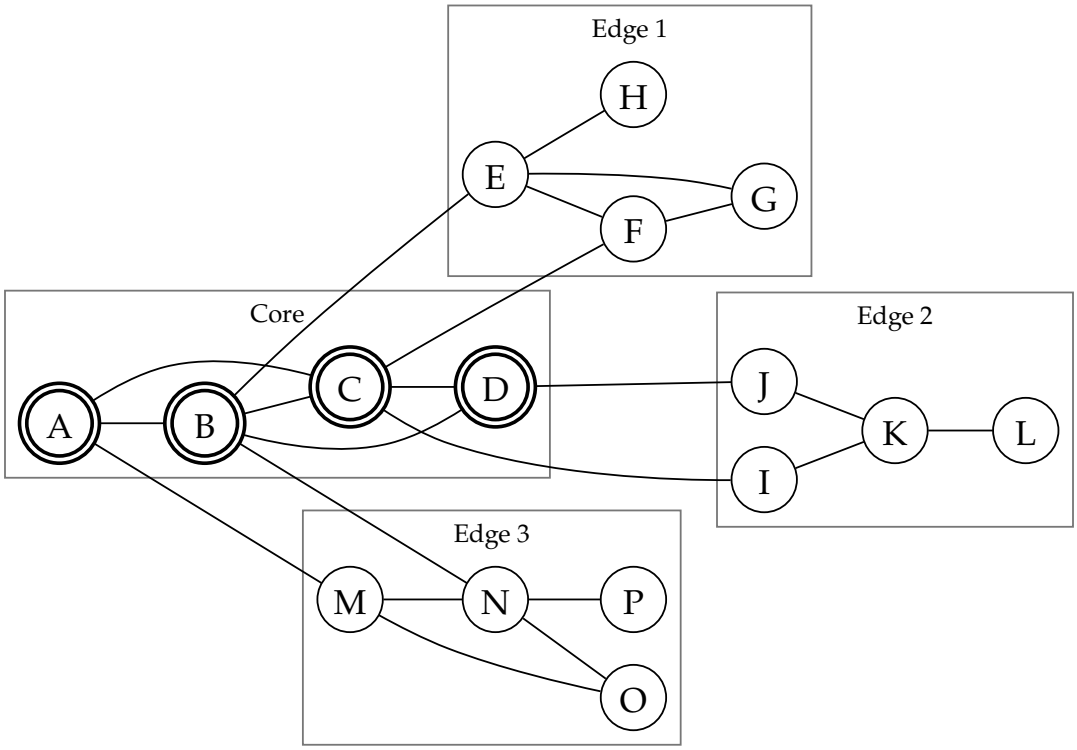

Figure 2: Core and edge networks of a two-tier community overlay network

The network is organized as an independent overlay network for each community, composed of community members' edge and core nodes (we use the terms nodes and peers interchangeably). See fig. 2 for an example.

A data repository is associated with each community overlay network, that is used by community members to share and collaborate on data. Each community is responsible for their own networking and storage, as opposed to storing data and relaying messages via non-interested nodes, as it is the case when a global overlay is used. Relying on community members' nodes instead of arbitrary non-interested and untrusted nodes increases efficiency, reliability, privacy, and security of the network, and establishes incentives for node participation, since nodes only contribute resources towards the communities they participate in.

The network architecture follows a two-tier peer-to-peer design that consists of local edge networks composed of end-user devices, and a core overlay network composed of brokers. The core network facilitates communication among remote nodes in different edge networks, and enables asynchronous communication via store-and-forward message brokers that provide routing and storage services. Each user is responsible for acquiring access to one or more core nodes of their choice, which can be self-hosted or offered by service providers. Each core node provides services only to the communities their users are part of.

The two-tier design also enables location privacy and reduces load on end-user devices, since they only connect to their brokers of choice in the core network and do not establish direct connections with remote nodes. This allows mobile devices to participate in the network without quickly draining their batteries, since they only have to maintain a single connection to a message broker, and they're not responsible for routing or storing messages for other nodes, while they can still participate in P2P protocols on edge networks on an on-demand basis.

The overlay messages are specified in the Data structures section.

2.2.1 Global overlay

For community overlays that want to be discoverable by ID, we employ a global overlay among core nodes, which consists of a trust-aware peer sampling protocol and a privacy-preserving interest clustering protocol based on community overlay membership, as described in [16]. The global overlay is optional. In the absence of it, peers rely on an explicit list of community overlay peers in order to join the overlay for the first time (see `OverlayJoin` and `RepoLink`). Peers store the list of connected peers for each community overlay, including the ones later discovered, and use it to rejoin the overlay the next time.

2.2.2 Community overlay networks

Each community overlay hosts a data repository and provides content-addressed object storage and pub/sub event dissemination services for their members. The pub/sub service is used for publishing commits in branches, and the object store allows synchronization between replicas.

The overlay network is identified by a BLAKE3 keyed hash over the repository public key, using a BLAKE3-derived key from the repository public key & secret. This helps to avoid correlation of overlays across edge networks.

Messages in the overlay (`OverlayMessage`) are first padded then encrypted using ChaCha20 with a key derived from the repository public key & secret and a per-peer random session ID, such that only peers in the overlay can decrypt them. Message

padding provides additional metadata protection against observing block sizes transmitted over the network.

Peers establish TLS or QUIC connections among each other when connecting over IP. However, the system does not depend on IP, and can function over other transports as well, such as overlay networks that provide public key routing.

2.2.3 Overlay construction & maintenance

A peer first establishes a connection following the reception of a `RepoLink` message, which contains the repository public key and secret, and the network addresses of one or more overlay peers to connect to.

Peers in the overlay discover each other via peer advertisement messages (`PeerAdvert`) that are sent periodically and follow random walks across the overlay with limited hops. Each node strives to reduce the number of connections by preferring to connect to peers with overlapping pub/sub topic subscriptions (interest clustering).

The `PeerAdvert` message contains a Bloom filter of a peer's topic subscriptions for the overlay it is sent to. This allows peers to find others with a similar subscription set, making pub/sub event dissemination in the overlay more efficient by reducing the number of connections required for a peer to cover all of its topic subscriptions, and by reducing the number of forwarding hops necessary from a publisher to subscribers.

Interest clustering also works across different overlays: when nodes receive multiple `PeerAdvert` messages from the same peer in different overlays, the subscriptions from all overlays are considered together in the interest clustering algorithm.

2.2.4 Publish-subscribe protocol

Pub/sub is used for sending low-latency push notifications about changes in a branch to subscribers of the corresponding pub/sub topic. Each commit and its referenced files are published via pub/sub.

The pub/sub protocol should satisfy the following requirements:

- Causal delivery :: Events should be delivered to the application in causal order.
- Reliability :: All published events should be delivered exactly once to subscribers (by detecting and retransmitting missed events and storing delivered events)
- Fault tolerance :: The system should tolerate and recover from faults in the event forwarding path (by establishing redundant paths)

An overview and detailed analysis of causal pub/sub can be found in [12].

A pub/sub topic is identified by a public key. Each published event must be signed with the corresponding private key in order to be routed by the pub/sub broker network. Events without a valid signature get dropped. This is necessary to enforce publishing permissions, to eliminate unsolicited events sent to a topic, and to avoid amplification attacks by unauthorized publishers that may result in denial of service.

Branches are mapped to pub/sub topics by the `Branch` definition. The topic private key is shared among all publishers, who publish each new commit as a signed event in the pub/sub topic. This is necessary in order to restrict the publishing of events to authorized publishers only, and at the same time hide the publisher's public key identities from the pub/sub brokers.

For each commit, change notifications are sent to the appropriate pub/sub topic as a `Change` inside a signed `Event` message. Each `Change` contains an encrypted `Block`, which is either a block of a commit, or a block of a file that a commit references. In case of the first block of a commit, it also contains the encryption key for the commit object, encrypted with a key derived from the branch public key & secret, and the publisher's public key. Brokers cannot decrypt this block, only subscribers, who need to look up the branch secret and the publisher's public key matching the publisher hash in the branch definition.

The `Event` is signed by the topic's private key, and contains a publisher hash that is a BLAKE3 keyed hash over the commit author's public key, the publisher's event sequence number used as encryption nonce and to detect missed events.

A pub/sub broker when subscribing to a topic first issues a `BranchHeadsReq` request to its upstream peers in the topic, then uses the synchronization protocol to synchronize the DAG for the branch based on the object IDs of the heads. Pub/sub brokers store and forward events for their subscribers, and recover missing events when they notice a missing sequence number via an `EventReq` request sent to upstream peers in the pub/sub topic.

We use a pub/sub subscription and event routing protocol inspired by LoCaPS [12]. Here follows a description of the base protocol without optimizations for subscription coverage and interest clustering that we leave for future work.

Publishers flood topic advertisements (`TopicAdvert`) to the network creating subscription routing table entries. A subscription request is forwarded from a subscriber to all publishers along subscription routing table entries, and creates an event routing table entry at each broker on the path. In response, each publisher sends an acknowledgement (`SubAck`) to all subscribers in an `Event`.

Events are forwarded from publishers to subscribers along event routing table entries. The subscription algorithm can establish multiple event routing paths when a subscriber sends subscription requests to multiple peers.

2.2.5 Branch synchronization

Replicas perform branch synchronization using a DAG synchronization protocol described in [[10]][8], by exchanging branch heads and a Bloom filter of known commits since the last synchronization with the given peer (`BranchSyncReq` & `BranchSyncRes`).

2.2.6 Block requests

Block requests either follow the reverse path of a pub/sub topic from a subscriber to publishers (`BlockSearchTopic`), or a random walk (`BlockSearchRandom`). The response (`BlockResponse`) contains one or more objects, and are sent either directly to the requesting node, or along the reverse path of the request.

Requests along a pub/sub topic are effective when a publisher refers to an object in a commit that some subscribers want to fetch, which is likely hosted by the publisher and its broker. The response gets cached on the way back to the requestor, in order to serve subsequent requests from other subscribers.

The other method involves issuing multiple requests along random walks across the overlay, and routing the response back to the requestor.

2.2.7 Broker protocol

The broker protocol is a client-server protocol that provides an interface to the message broker running on a core or edge node. The broker provides access to its authorized clients to the pub/sub network and object store. Applications use it to connect to their local edge node, while edge nodes use it to connect to a core node, and perform local-first processing on application requests: if the local node can answer the application request it does so, otherwise forwards the query to its configured broker in the core network, and/or sends the query to the edge network. In case a local node is not available, applications can talk directly to a broker in the core network.

Brokers running locally on an edge node have access to cryptographic keys to decrypt and process repository contents, while core nodes do not. This allows local nodes to decrypt and validate incoming commits, and to decrypt and assemble object contents from blocks in order to serve client requests.

The broker protocol is available via WebSocket over TLS. It requires authentication, which is initiated by the client using the `ClientHello` message, and finishes with an `AuthResult`.

After successful authentication, it and allows clients to request brokers to join & leave an overlay (`OverlayJoin` & `OverlayLeave`) , subscribe to & unsubscribe from topics

(`TopicSub` & `TopicUnsub`), publish & receive events (`Event`), synchronize branches (`BranchHeadsReq` & `BranchSyncReq`), as well as interact with the object store to: download & upload blocks (`BlockGet` & `BlockPut`), pin objects for permanent storage (`ObjectPin`), copy objects with a different expiry time before sharing (`ObjectCopy`), and delete objects from storage (`ObjectDel`).

The broker provides an interface to the object store via HTTP over TLS. HTTP requests require an authentication token which is returned in `AuthResult` by the broker protocol.

2.2.8 External requests

Sharing data from a repository for external clients not part of the overlay and not members of the repository is possible by including a Message Authentication Code (MAC) in the request (`ExtRequest`). For this purpose we use a BLAKE3 keyed hash over the request content, keyed with the repository public key and secret. This functionality is optional and can be enabled by setting the `allowExtRequests` flag in the `Repository` definition.

Object requests from external clients (`ExtObjectReq`) contain a list of object IDs to request, and a flag whether or not to include all children dependencies of the object, which allows cloning a branch at a specific commit, and an expiry time of the request after which it becomes invalid and overlay peers won't serve the request anymore.

Branch synchronization for external clients is provided via `ExtBranchHeadsReq` and `ExtBranchSync` that work the same way as the similarly named requests in the application protocol.

3 Future work

As future work we intend to research, design, and develop:

- Definition and execution of transaction validation rules.
- A name system that allows repositories to define named references to branches, commits, and files, as well as allow users to define named references to repositories they subscribe to.
- An URI scheme that allows references to repositories, branches, commits, and files, either by ID or name.
- External membership & authentication mechanisms that synchronize the repository membership with an external source
- A system for hosting providers with anonymous payments for hosting credits.
- A publishing workflow, starting from collaboration in a repository that results in published immutable artefacts.

- Further improvements and optimizations for the pub/sub protocol to leverage subscription coverage and interest clustering.
- LAN protocols based on IP multicast for peer discovery and pub/sub.
- Local-first search & discovery protocols that find information in locally stored data, in the community overlays, and in the global network.
- Additional transport and synchronization methods, including Delay/Disruption Tolerant Networking (DTN) and local file systems (removable storage).
- Storage and query of graph data models, semantic web data.

4 Data structures

In this section we document the data structures for repository objects and overlay messages as BARE [1] message schema. BARE is a compact binary encoding that does not encode schema information. An external schema is used instead that provides a human and machine-readable specification, which is the source of code generation for various programming language targets. Data structures are versioned in order to allow schema evolution with forwards and backwards compatibility of messages.

```
##
## COMMON DATA TYPES
##

# 32-byte BLAKE3 hash
type Blake3Digest32 data[32]

# Hash digest
type Digest union { Blake3Digest32 }

# ChaCha20 key
type ChaCha20Key data[32]

# Symmetric key
type SymKey union { ChaCha20Key }

# Curve25519 public key
type Curve25519PubKey data[32]

# Curve25519 private key
type Curve25519PrivKey data[32]

# Public key
type PubKey union { Curve25519PubKey }
```

```
# Private key
type PrivKey union { Curve25519PrivKey }

# Ed25519 signature
type Ed25519Sig data[64]

# Cryptographic signature
type Sig union { Ed25519Sig }

# Timestamp: absolute time in minutes since 2022-02-22 22:22 UTC
type Timestamp u32

# Relative time in seconds
type Seconds u8

# Relative time in minutes
type Minutes u8

# Relative time in hours
type Hours u8

# Relative time in days
type Days u8

# Relative time (e.g. delay from current time)
type RelTime union {
  | Seconds | Minutes | Hours | Days
}

##
## STORAGE OBJECTS
##

# Block ID
#
# BLAKE3 hash over the serialized Block with encrypted content.
type BlockId Digest

# Block reference
type BlockRef struct {
  # Block ID
  id: BlockId
```

```
  # Key for decrypting the Block
  key: SymKey
}

# ID of the root block of a Merkle tree
type ObjectId BlockId

# Object reference
type ObjectRef BlockRef

# Internal node of a Merkle tree
type InternalNode list<SymKey>

# Data chunk at a leaf of a Merkle tree
type DataChunk data

# Content of a Block
type BlockContentV0 union {
  # Internal node with references to children
  | InternalNode

  # Leaf node with encrypted data chunk
  | DataChunk
}

# List of Object dependencies stored in an Object
type DepList union {
  list<ObjectId>
}

# Dependencies of an Object,
# referenced from the root Block
type ObjectDeps union {
  # List of Object IDs
  | list<ObjectId>

  # Reference to an Object that contains a DepList
  | ObjectRef
}

# Immutable block with encrypted content
#
# Object content is chunked and stored as blocks in a Merkle tree.
# A Block is a Merkle tree node.
```

```
type BlockV0 struct {
  # BlockIds for child nodes in the Merkle tree
  children: list<BlockId>

  # Other objects this object depends on (e.g. Commit deps & acks)
  # only set for the root Block
  deps: ObjectDeps

  # Expiry time of this block and all of its children
  # when the block should be deleted by all replicas
  expiry: optional<Timestamp>

  # Encrypted BlockContentV0
  #
  # Encrypted using convergent encryption with ChaCha20:
  # - convergence_key: BLAKE3 derive_key ("LoFiRe Data BLAKE3 key",
  #                                       repo_pubkey + repo_secret)
  # - key: BLAKE3 keyed hash (convergence_key, plain_block)
  # - nonce: 0
  content: data # BlockContentV0
}
type Block union { BlockV0 }

# Repository definition
#
# Published in root branch, where:
# - branch_pubkey: repo_pubkey
# - branch_secret: BLAKE3 derive_key ("LoFiRe Root Branch secret",
#                                     repo_pubkey + repo_secret)
type RepositoryV0 struct {
  # Repo public key ID
  id: PubKey

  # List of branches
  branches: list<ObjectRef>

  # Whether or not to allow external requests
  allowExtRequests: bool

  # App-specific metadata
  metadata: data
}
type Repository union { RepositoryV0 }
```

```
# Add a branch to the repository
type AddBranchV0 ObjectRef
type AddBranch union { AddBranchV0 }

# Remove a branch from the repository
type RemoveBranchV0 ObjectRef
type RemoveBranch union { RemoveBranchV0 }

# Commit block types
type CommitType enum {
  REPOSITORY ADD_BRANCH REMOVE_BRANCH
  BRANCH ADD_MEMBERS END_OF_BRANCH
  TRANSACTION SNAPSHOT COMMIT_ACK
}

# Member of a branch
type MemberV0 struct {
  # Member public key ID
  id: PubKey

  # Commit types the member is allowed to publish in the branch
  commitTypes: list<CommitType>

  # App-specific metadata
  # (role, permissions, cryptographic material, etc)
  metadata: data
}
type Member union { MemberV0 }

# Branch definition
#
# First commit in a branch, signed by branch key
# In case of a fork, the commit deps indicate the previous branch heads.
type BranchV0 struct {
  # Branch public key ID
  id: PubKey

  # Pub/sub topic for publishing events
  topic: PubKey

  # Branch secret key
  secret: SymKey

  # Members with permissions
```

```
  members: list<Member>

  # Number of acks required for a commit to be valid
  quorum: map<CommitType><u32>

  # Delay to send explicit acks, if not enough implicit acks arrived by then
  ackDelay: RelTime

  # Tags for organizing branches within the repository
  tags: list<data>

  # App-specific metadata (validation rules, etc)
  metadata: data
}
type Branch union { BranchV0 }

# Add members to an existing branch
#
# If a member already exists, it overwrites the previous definition,
# in that case this can only be used for adding new permissions,
# not to remove existing ones.
# The quorum and ackDelay can be changed as well.
type AddMembersV0 struct {
  # Members to add, with permissions
  members: list<Member>

  # New quorum
  quorum: optional<map<CommitType><u32>>

  # New ackDelay
  ackDelay: optional<RelTime>
}
type AddMembers union { AddMembersV0 }

type PlainOrEncryptedObjectRef union {
  | ObjectRef
  | data       # Encrypted ObjectRef
}

# End of branch
#
# No more commits accepted afterwards, only acks of this commit.
# May reference a fork where the branch continues
# with possibly different members, permissions, validation rules.
```

```
type EndOfBranchV0 struct {
  # (Encrypted) reference to forked branch (optional)
  fork: optional<PlainOrEncryptedObjectRef>

  # Expiry time when all commits in the branch should be deleted
  expiry: Timestamp
}
type EndOfBranch union { EndOfBranchV0 }

# Transaction with CRDT operations
type TransactionV0 data
type Transaction union { TransactionV0 }

# Snapshot of a Branch
#
# Contains a data structure computed from the commits at the specified head.
type SnapshotV0 struct {
  # Branch heads the snapshot was made from
  heads: list<ObjectId>

  # Snapshot data structure
  content: data
}
type Snapshot union { SnapshotV0 }

# Acknowledgement of another Commit
type Ack union {
  void
}

# Commit body, corresponds to CommitType
type CommitBody union {
  | Repository | AddBranch | RemoveBranch
  | Branch | AddMembers | EndOfBranch
  | Transaction | Snapshot | Ack
}

# Content of CommitV0
type CommitContentV0 struct {
  # Commit author
  author: PubKey

  # Author's commit sequence number in this branch
  seq: u32
```

```
  # Branch the commit belongs to
  branch: ObjectRef

  # Direct dependencies of this commit
  deps: list<ObjectRef>

  # Not directly dependent heads to acknowledge
  acks: list<ObjectRef>

  # Files the commit references
  refs: list<ObjectRef>

  # App-specific metadata (commit message, creation time, etc)
  metadata: data

  # Block with a CommitBody inside
  body: ObjectRef

  # Expiry time of the body block
  expiry: optional<Timestamp>
}

# Commit Object
#
# Signed by branch key, or a member key authorized to publish this commit type.
type CommitV0 struct {
  content: CommitContentV0

  # Signature over the content by the author
   sig: Sig
}
type Commit union { CommitV0 }

# File Object
type FileV0 struct {
  contentType: data
  metadata: data
  content: data
}
type File union { FileV0 }

# Immutable object content stored in encrypted blocks of a Merkle tree
type ObjectContent union {
```

```
  | Commit | CommitBody | File | DepList
}

##
## COMMON DATA TYPES FOR MESSAGES
##

# Peer ID
type PeerId PubKey

# Overlay ID
#
# - for public overlays that need to be discovered by public key:
#   BLAKE3 hash over the repository public key
# - for private overlays:
#   BLAKE3 keyed hash over the repository public key
#   - key: BLAKE3 derive_key ("LoFiRe OverlayId BLAKE3 key", repo_secret)
type OverlayId Digest

# Overlay session ID
#
# Used as a component for key derivation.
# Each peer generates it randomly when (re)joining the overlay network.
type SessionId u64

# Topic ID
type TopicId PubKey

# Result code
type Result enum { OK ERROR }

# IP address
type IPv4 data[4]
type IPv6 data[16]
type IP union { IPv4 | IPv6 }

type IPTransportProtocol enum { TLS QUIC }

type IPTransportAddr struct {
  ip: IP
  port: u16
  protocol: IPTransportProtocol
}
```

```
# Network address
type NetAddr union { IPTransportAddr }

# Bloom filter (variable size)
type BloomFilter struct {
  # Number of hash functions
  k: u8

  # Filter
  f: data
}

# Bloom filter (128 B)
# (m=1024; k=7; p=0.01; n=107)
type BloomFilter128 data[128]

# Bloom filter (1 KiB)
# (m=8192; k=7; p=0.01; n=855)
type BloomFilter1K data[1024]

##
## OVERLAY MESSAGES
##

# Overlay connection request
#
# Sent to an existing overlay member to initiate a session.
type OverlayConnect union {
  void
}

# Overlay disconnection request
#
# Sent to a connected overlay member to terminate a session.
type OverlayDisconnect union {
  void
}

# Content of TopicAdvertV0
type TopicAdvertContentV0 struct {
  # Topic public key
  topic: PubKey

  # Peer public key
```

```
  peer: PeerId
}

# Topic advertisement by a publisher
#
# Flooded to all peers in overlay.
# Creates subscription routing table entries.
type TopicAdvertV0 struct {
  content: TopicAdvertContentV0

  # Signature over content by topic key
  sig: Sig
}
type TopicAdvert union { TopicAdvertV0 }

# Topic subscription request by a peer
#
# Forwarded towards all publishers along subscription routing table entries
# that are created by TopicAdverts.
# Creates event routing table entries along the path.
type SubReqV0 struct {
  # Random ID generated by the subscriber
  id: u64

  # Topic public key
  topic: PubKey
}
type SubReq union { SubReqV0 }

# Topic subscription acknowledgement by a publisher
#
# Sent to all subscribers in an Event.
type SubAckV0 struct {
  # SubReq ID to acknowledge
  id: u64
}
type SubAck union { SubAckV0 }

# Topic unsubscription request by a subscriber
#
# A broker unsubscribes from upstream brokers
# when it has no more subscribers left.
type UnsubReqV0 struct {
  # Topic public key
```

```
  topic: PubKey
}
type UnsubReq union { UnsubReqV0 }

# Topic unsubscription acknowledgement
# Sent to the requestor in response to an UnsubReq
type UnsubAckV0 struct {
  # Topic public key
  topic: PubKey
}
type UnsubAck union { UnsubAckV0 }

# Branch change notification
#
# Contains a chunk of a newly added Commit or File referenced by a commit.
type ChangeV0 struct {
  # Block with encrypted content
  content: Block

  # Encrypted key for the Commit object in content
  # Only set for the root block of the object
  # The key is encrypted using ChaCha20:
  # - key: BLAKE3 derive_key ("LoFiRe Change Object ChaCha20 key",
  #                           repo_pubkey + repo_secret +
  #                           branch_pubkey + branch_secret + publisher_pubkey)
  # - nonce: commit_seq
  key: optional<data[32]> # SymKey
}
type Change union { ChangeV0 }

# Body of EventContentV0
type EventBodyV0 union {
  | SubAck | Change
}

# Content of EventV0
type EventContentV0 struct {
  # Pub/sub topic
  topic: PubKey

  # Publisher pubkey encrypted with ChaCha20:
  # - key: BLAKE3 derive_key ("LoFiRe Event Publisher ChaCha20 key",
  #                           repo_pubkey + repo_secret +
  #                           branch_pubkey + branch_secret)
```

```
  publisher: data[32] # PubKey

  # Commit sequence number of publisher
  seq: u32

  # Event body
  body: EventBodyV0
}

# Pub/sub event published in a topic
#
# Forwarded along event routing table entries.
type EventV0 struct {
  content: EventContentV0

  # Signature over content by topic key
  sig: Sig
}
type Event union { EventV0 }

# Block search in a pub/sub topic
#
# Sent along the reverse path of a pub/sub topic
# from a subscriber to all publishers
type BlockSearchTopicV0 struct {
  # Topic to forward the request in
  topic: PubKey

  # List of BlockIds to request
  ids: list<BlockId>

  # Whether or not to include all children recursively in the response
  includeChildren: bool

  # List of Peer IDs the request traversed so far
  path: list<PeerId>
}
type BlockSearchTopic union { BlockSearchTopicV0 }

# Block search along a random walk
type BlockSearchRandomV0 struct {
  # List of BlockIds to request
  ids: list<BlockId>
```

```
  # Whether or not to include all children recursively in the response
  includeChildren: bool

  # Number of random nodes to forward the request to at each step
  fanout: u8

  # List of Peer IDs the request traversed so far
  path: list<PeerId>
}
type BlockSearchRandom union { BlockSearchRandomV0 }

# Response to a BlockSearch* request
#
# Follows request path with possible shortcuts.
type BlockResultV0 struct {
  # Response path
  path: list<PeerId>

  # Resulting Block(s)
  payload: list<Block>
}
type BlockResult union { BlockResultV0 }

# Request latest events corresponding to the branch heads in a pub/sub topic.
#
# In response an Event is sent for each commit chunk that belong to branch heads
# that are not present in the requestor's known heads.
type BranchHeadsReqV0 struct {
  # Topic public key of the branch
  topic: PubKey

  # Known heads
  knownHeads: list<ObjectId>
}
type BranchHeadsReq union { BranchHeadsReqV0 }

# Branch synchronization request
#
# In response a stream of Blocks of the requested Objects are sent
# that are not present in the requestor's known heads and commits
type BranchSyncReqV0 struct {
  # Heads to request, including all their dependencies
  heads: list<ObjectId>
```

```
  # Fully synchronized until these commits
  knownHeads: list<ObjectId>

  # Known commit IDs since knownHeads
  knownCommits: BloomFilter
}
type BranchSyncReq union { BranchSyncReqV0 }

# Events the requestor needs
type NeedEventsV0 struct {
  # Publisher ID
  publisher: Digest

  # First sequence number to request
  from: u32

  # Last sequence number to request
  to: u32
}

# Events the responder has
type HaveEventsV0 struct {
  # Publisher ID
  publisher: Digest

  # First sequence number to send
  from: u32

  # Last sequence number to send
  to: u32
}

# Request missed events for a pub/sub topic
# for the specified range of publisher sequence numbers.
#
# In response an EventResp then a stream of Events are sent.
type EventReqV0 struct {
  # Topic public key
  topic: PubKey

  # Events needed by the requestor
  need: list<NeedEventsV0>
}
type EventReq union { EventReqV0 }
```

```
# Response to an EventReq
type EventRespV0 struct {
  # Events the responder has
  have: list<HaveEventsV0>
}
type EventResp union { EventRespV0 }

# Content of OverlayRequestV0
type OverlayRequestContentV0 union {
  | EventReq
  | BranchHeadsReq
  | BranchSyncReq
}

# Request sent to an overlay
type OverlayRequestV0 struct {
  # Request ID
  id: u64

  # Request content
  content: OverlayRequestContentV0
}
type OverlayRequest union { OverlayRequestV0 }

# Content of OverlayResponse
type OverlayResponseContentV0 union {
  | Block
  | EventResp
  | Event
}

# Response to an OverlayRequest
type OverlayResponseV0 struct {
  # Request ID
  id: u64

  # Result
  result: u16

  # Response content
  content: optional<OverlayResponseContentV0>
}
type OverlayResponse union { OverlayResponseV0 }
```

```
# Content of PeerAdvertV0
type PeerAdvertContentV0 struct {
  # Peer ID
  peer: PeerId

  # Topic subscriptions
  subs: BloomFilter128

  # Network addresses
  address: list<NetAddr>

  # Version number
  version: u32

  # App-specific metadata (profile, cryptographic material, etc)
  metadata: data
}

# Peer advertisement
#
# Sent periodically across the overlay along random walks.
type PeerAdvertV0 struct {
  # Peer advertisement content
  content: PeerAdvertContentV0

  # Signature over content by peer's private key
  sig: Sig

  # Time-to-live, decremented at each hop
  ttl: u8
}
type PeerAdvert union { PeerAdvertV0 }

# Content of OverlayMessageContentPaddedV0
type OverlayMessageContentV0 union {
  | OverlayConnect | OverlayDisconnect
  | PeerAdvert | TopicAdvert
  | SubReq | SubAck
  | UnsubReq | UnsubAck
  | Event
  | BlockSearchTopic | BlockSearchRandom | BlockResult
  | OverlayRequest | OverlayResponse
}
```

```
# Padded content of OverlayMessageV0
type OverlayMessageContentPaddedV0 struct {
  content: OverlayMessageContentV0

  # Optional padding
  padding: data
}

# Overlay message
type OverlayMessageV0 struct {
  # Overlay ID
  overlay: OverlayId

  # Session ID
  session: SessionId

  # Padded content encrypted with ChaCha20
  #
  # - overlay_secret: BLAKE3 derive_key ("LoFiRe Overlay BLAKE3 key",
  #                                      repo_pubkey + repo_secret)
  # - key: BLAKE3 derive_key ("LoFiRe OverlayMessage ChaCha20 key",
  #                           overlay_secret + session_id)
  # - nonce: per-session message sequence number of sending peer
  content: data # OverlayMessageContentPaddedV0

  # BLAKE3 MAC
  # BLAKE3 keyed hash over the encrypted content
  # - key: BLAKE3 derive_key ("LoFiRe OverlayMessage BLAKE3 key",
  #                           overlay_secret + session_id)
  mac: Digest
}
type OverlayMessage union { OverlayMessageV0 }

##
## BROKER MESSAGES
##

# Content of AddUserV0
type AddUserContentV0 struct {
  # User pub key
  user: PubKey
}
```

```
# Add user account
type AddUserV0 struct {
  content: AddUserContentV0

  # Signature by admin key
  sig: Sig
}
type AddUser union { AddUserV0 }

# Content of DelUserV0
type DelUserContentV0 struct {
  # User pub key
  user: PubKey
}

# Delete user account
type DelUserV0 struct {
  content: DelUserContentV0

  # Signature by admin key
  sig: Sig
}
type DelUser union { DelUserV0 }

# Content of AddClientV0
type AddClientContentV0 struct {
  # Client pub key
  client: PubKey
}

# Add a client
type AddClientV0 struct {
  content: AddClientContentV0

  # Signature by user key
  sig: Sig
}
type AddClient union { AddClientV0 }

# Content of DelClientV0
type DelClientContentV0 struct {
  # Client pub key
  client: PubKey
}
```

```
# Remove a client
type DelClientV0 struct  {
  content: DelClientContentV0

  # Signature by user key
  sig: Sig
}
type DelClient union { DelClientV0 }

# Content of BrokerRequest
type BrokerRequestContentV0 union {
  | AddUser | DelUser
  | AddClient | DelClient
}

# Broker request
type BrokerRequestV0 struct {
  # Request ID
  id: u64

  # Request content
  content: BrokerRequestContentV0
}
type BrokerRequest union { BrokerRequestV0 }


# Response to an BrokerRequest
type BrokerResponseV0 struct {
  # Request ID
  id: u64

  # Result (including but not limited to Result)
  result: u16
}
type BrokerResponse union { BrokerResponseV0 }

# Request to join an overlay
type OverlayJoinV0 struct {
  # Overlay secret
  secret: SymKey

  # Public key for the repository the overlay belongs to.
  # Mandatory for local brokers.
```

```
  repoPubKey: optional<PubKey>

  # Peers to connect to
  peers: list<PeerAdvert>
}
type OverlayJoin union { OverlayJoinV0 }

# Request to leave an overlay
type OverlayLeave union {
  void
}

# Overlay status request
type OverlayStatusReq union {
  void
}

# Overlay status response
type OverlayStatusRespV0 struct {
  # Whether or not the broker has joined the overlay
  joined: bool

  # List of peers currently connected in the overlay
  peers: list<PeerAdvert>
}
type OverlayStatusResp union { OverlayStatusRespV0 }

# Request a block by ID
type BlockGetV0 struct {
  # Block ID to request
  id: BlockId

  # Whether or not to include all children recursively
  includeChildren: bool

  # Topic the block is referenced from
  topic: optional<PubKey>
}
type BlockGet union { BlockGetV0 }

# Request to store a block
type BlockPut union { Block }

# Request to pin an object
```

```
#
# Brokers maintain an LRU cache of objects,
# where old, unused objects might get deleted to free up space for new ones.
# Pinned objects are retained, regardless of last access.
# Note that expiry is still observed in case of pinned objects.
# To make an object survive its expiry,
# it needs to be copied with a different expiry time.
type ObjectPinV0 struct {
  id: ObjectId
}
type ObjectPin union { ObjectPinV0 }

# Request to unpin an object
type ObjectUnpinV0 struct {
  id: ObjectId
}
type ObjectUnpin union { ObjectUnpinV0 }

# Request to copy an object with a different expiry time
type ObjectCopyV0 struct {
  # Object ID to copy
  id: ObjectId

  # New expiry time
  expiry: optional<Timestamp>
}
type ObjectCopy union { ObjectCopyV0 }

# Request to delete an object
type ObjectDelV0 struct {
  id: ObjectId
}
type ObjectDel union { ObjectDelV0 }

# Request subscription to a topic
type TopicSubV0 struct {
  # Topic to subscribe
  topic: PubKey

  # Publishers need to include a signed TopicAdvert for the PeerId of the broker
  advert: optional<TopicAdvertV0>
}
type TopicSub union { TopicSubV0 }
```

```
# Request unsubscription from a topic
type TopicUnsubV0 struct {
  # Topic to unsubscribe
  topic: PubKey
}
type TopicUnsub union { TopicUnsubV0 }

# Connect to an already subscribed topic, and start receiving its Events
type TopicConnectV0 struct {
  # Topic to connect
  topic: PubKey
}
type TopicConnect union { TopicConnectV0 }

# Disconnect from a topic and stop receiving its Events
type TopicDisconnectV0 struct {
  # Topic to disconnect
  topic: PubKey
}
type TopicDisconnect union { TopicDisconnectV0 }

# Content of BrokerOverlayRequest
type BrokerOverlayRequestContentV0 union {
  | OverlayStatusReq | OverlayJoin | OverlayLeave
  | TopicSub | TopicUnsub
  | TopicConnect | TopicDisconnect
  | Event
  | BlockGet | BlockPut
  | ObjectPin | ObjectUnpin
  | ObjectCopy | ObjectDel
  | BranchHeadsReq | BranchSyncReq
}

# Broker overlay request
type BrokerOverlayRequestV0 struct {
  # Request ID
  id: u64

  # Request content
  content: BrokerOverlayRequestContentV0
}
type BrokerOverlayRequest union { BrokerOverlayRequestV0 }

# Content of BrokerOverlayResponseV0
```

```
type BrokerOverlayResponseContentV0 union {
  | Block
  | ObjectId
  | OverlayStatusResp
}

# Response to an BrokerOverlayRequest
type BrokerOverlayResponseV0 struct {
  # Request ID
  id: u64

  # Result (including but not limited to Result)
  result: u16
  content: optional<BrokerOverlayResponseContentV0>
}
type BrokerOverlayResponse union { BrokerOverlayResponseV0 }

# Content of BrokerOverlayMessageV0
type BrokerOverlayMessageContentV0 union {
  | BrokerOverlayRequest | BrokerOverlayResponse
  | Event
}

# Message for an overlay
type BrokerOverlayMessageV0 struct {
  overlay: OverlayId
  content: BrokerOverlayMessageContentV0
}
type BrokerOverlayMessage union { BrokerOverlayMessageV0 }

# Content of BrokerMessage
type BrokerMessageContentV0 union {
  | BrokerRequest | BrokerResponse
  | BrokerOverlayMessage
}

# Broker message
type BrokerMessageV0 struct {
  content: BrokerMessageContentV0

  # Optional padding
  padding: data
}
type BrokerMessage union { BrokerMessageV0 }
```

```
##
## EXTERNAL REQUESTS
##

# Request object(s) by ID from a repository by non-members
#
# The request is sent by a non-member to an overlay member node,
# which has a replica of the repository.
#
# The response includes all blocks of the requested objects,
# and optionally all object dependencies recursively.
type ExtObjectGetV0 struct {
  # Repository to request the objects from
  repo: PubKey

  # List of Object IDs to request, including their children
  ids: list<ObjectId>

  # Whether or not to include object dependencies
  includeDeps: bool

  # Expiry time after which the link becomes invalid
  expiry: optional<Timestamp>
}
type ExtObjectGet union { ExtObjectGetV0 }

# Branch heads request
type ExtBranchHeadsReq BranchHeadsReq

# Branch synchronization request
type ExtBranchSyncReq BranchSyncReq

# Content of ExtRequestV0
type ExtRequestContentV0 union {
  | ExtObjectGet
  | ExtBranchHeadsReq
  | ExtBranchSyncReq
}

# External request authenticated by a MAC
type ExtRequestV0 struct {
  # Request ID
  id: u64
```

```
  # Request content
  content: ExtRequestContentV0

  # BLAKE3 MAC over content
  # BLAKE3 keyed hash:
  # - key: BLAKE3 derive_key ("LoFiRe ExtRequest BLAKE3 key",
  #                           repo_pubkey + repo_secret)      # FIXME
  mac: Digest
}
type ExtRequest union { ExtRequestV0 }

# Content of ExtResponseV0
type ExtResponseContentV0 union {
  | Block
  | EventResp
  | Event
}

# Response to an ExtRequest
type ExtResponseV0 struct {
  # Request ID
  id: u64

  # Result code
  result: u16

  # Response content
  content: optional<ExtResponseContentV0>
}
type ExtResponse union { ExtResponseV0 }

##
## AUTHENTICATION MESSAGES
##

type ClientHelloV0 void
type ClientHello union { ClientHelloV0 }

# Initiate connection - choose broker or ext protocol
# First message sent by the client
type StartProtocol union { ClientHello | ExtRequest }

# Server hello sent upon a client connection
```

```
type ServerHelloV0 struct {
  # Nonce for ClientAuth
  nonce: data
}
type ServerHello union { ServerHelloV0 }

# Client authentication
type ClientAuthV0 struct {
  # Authentication data
  content: struct {
    # User pub key
    user: PubKey

    # Client pub key
    client: PubKey

    # Nonce from ServerHello
    nonce: data
  }

  # Signature by client key
  sig: Sig
}
type ClientAuth union { ClientAuthV0 }

# Authentication result
type AuthResultV0 struct {
  # Authentication result (Ok/Error)
  result: u16

  # Authentication token for HTTP
  token: optional<data>
}
type AuthResult union { AuthResultV0 }

##
## DIRECT MESSAGES
##

# Link/invitation to the repository
type RepoLinkV0 struct {
  # Repository public key ID
  id: PubKey
```

```
  # Repository secret
  secret: SymKey

  # Peers to connect to
  peers: list<PeerAdvert>
}
type RepoLink union { RepoLinkV0 }

# Owned repository with private key
type RepoKeysV0 struct {
  # Repository private key
  key: PrivKey

  # Repository secret
  secret: SymKey

  # Peers to connect to
  peers: list<PeerAdvert>
}
type RepoKeys union { RepoKeysV0 }

# Link to object(s) or to a branch from a repository
# that can be shared to non-members
type ObjectLinkV0 struct {
  # Request to send to an overlay peer
  req: ExtRequest

  # Keys for the root blocks of the requested objects
  keys: list<ObjectRef>
}
type ObjectLink union { ObjectLinkV0 }

##
## BROKER STORAGE
##

# A topic this node subscribed to in an overlay
type TopicV0 struct {
  # Topic public key ID
  id: PubKey

  # Signed TopicAdvert for publishers
  advert: optional<TopicAdvertV0>
```

```
  # Set of branch heads
  heads: list<ObjectId>

  # Number of local users that subscribed to the topic
  users: u32
}
type Topic union { TopicV0 }

# An overlay this node joined
type OverlayV0 struct {
  # Overlay ID
  id: OverlayId

  # Overlay secret
  secret: SymKey

  # Known peers with connected flag
  peers: list<PeerAdvert>

  # Topics this node subscribed to in the overlay
  topics: list<TopicId>

  # Number of local users that joined the overlay
  users: u32

  # Last access by any user
  lastAccess: Timestamp
}
type Overlay union { OverlayV0 }

# User accounts
#
# Stored as user_pubkey -> Account
type AccountV0 struct {
  # Authorized client pub keys
  clients: list<PubKey>

  # Admins can add/remove user accounts
  admin: bool

  # Overlays joined
  overlays: list<OverlayId>
}
type Account union { AccountV0 }
```

5 Acknowledgements

Thanks to Niko Bonnieure for feedback on a draft of this document.

References

[1] BARE message encoding.

[2] Radicle documentation.

[3] Baquero, C., Almeida, P.S. and Shoker, A. 2014. Making operation-based CRDTs operation-based. *IFIP international conference on distributed applications and interoperable systems* (2014), 126–140.

[4] Bazerque, S. Hyper hyper space.

[5] Brock, A., Atkinson, D., Friedman, E., Harris-Braun, E., Mcguire, E., Russell, J.M., Perrin, N., Luck, N. and Harris-Braun, W. 2018. Holo green paper. (2018).

[6] Karlsson, K., Jiang, W., Wicker, S., Adams, D., Ma, E., Renesse, R. van and Weatherspoon, H. 2018. Vegvisir: A partition-tolerant blockchain for the internet-of-things. *2018 IEEE 38th international conference on distributed computing systems (ICDCS)* (2018), 1150–1158.

[7] Kleppmann, M. 2022. Making CRDTs byzantine fault tolerant. *Proceedings of the 9th workshop on principles and practice of consistency for distributed data* (2022), 8–15.

[8] Kleppmann, M. 2020. Using bloom filters to efficiently synchronise hash graphs. (2020).

[9] Kleppmann, M. and Beresford, A.R. 2018. Automerge: Real-time data sync between edge devices. *1st UK mobile, wearable and ubiquitous systems research symposium (MobiUK 2018)* (2018), 101–105.

[10] Kleppmann, M. and Howard, H. 2020. Byzantine eventual consistency and the fundamental limits of peer-to-peer databases. *CoRR*. abs/2012.00472, (2020).

[11] Kleppmann, M., Wiggins, A., Van Hardenberg, P. and McGranaghan, M. 2019. Local-first software: You own your data, in spite of the cloud. *Proceedings of the 2019 ACM SIGPLAN international symposium on new ideas, new paradigms, and reflections on programming and software* (2019), 154–178.

[12] Pedrosa, F.S.R. 2020. LoCaPS: Localized causal publish-subscribe. (2020).

[13] pukkamustard Distributed mutable containers.

[14] pukkamustard Encoding for robust immutable storage (ERIS).

[15] Santos, V. and Rodrigues, L. 2019. Localized reliable causal multicast. *2019 IEEE 18th international symposium on network computing and applications (NCA)* (2019), 1–10.

[16] Thoth, T.G. x 2020. UPSYCLE: Ubiquitous publish-subscribe infrastructure for collaboration on edge networks. (2020).