Merkle-CRDTs (DRAFT)

Merkle-DAGs meet CRDTs

Héctor Sanjuán[1], Samuli Pöyhtäri[2], and Pedro Teixeira[1]

[1]Protocol Labs
[2]Haja Networks

May, 2019

Abstract

We study Merkle-DAGs as transport and persistence layer for Conflict free Replicated Data Types (CRDTs), coining the term *Merkle-CRDTs* and providing an overview of the different concepts, properties, advantages and limitations involved. We show how Merkle-DAGs can act as *logical clocks* giving Merkle-CRDTs the potential to greatly simplify the design and implementation of convergent data types in systems with weak messaging layer guarantees and a very large number of replicas. Merkle-CRDTs can leverage highly scalable distributed technologies like DHT and pub/sub algorithms running underneath to take advantage of the security and de-duplication properties of content-addressing. Examples of such content systems could include peer-to-peer content exchange and synchronisation applications between opportunistically connected mobile devices, IoT devices or user applications running in a web browser.

Keywords: CRDTs, Merkle DAGs, Distributed Systems, IPFS, logical clocks.

1 Introduction

The advent of blockchain technology has generalized the use of peer-to-peer networking along with cryptographically directed, acyclic graphs, known as Merkle-DAGs, to implement *globally distributed and eventually consistent data structures* in applications such as cryptocurrencies. In these systems, the Merkle-DAG is a content-addressed data structure used to provide both *causality information and self-verification* of objects that can be easily and efficiently shared in *trustless peer-to-peer environments.* The need to maintain and apply certain rules to add new blocks to the blockchains in adversarial scenarios usually warrants the use of consensus algorithms.

A different approach to obtaining eventual consistency in a distributed system is by using Conflict-Free Replicated Data Types (CRDTs) [26, 27]. CRDTs are useful in non-adversarial scenarios, where the participating replicas are known to behave correctly. CRDTs rely on some properties of the data objects themselves that enable convergence towards a global, unique state without the need for consensus. CRDTs come in two main flavours: *state-based CRDTs*[1]—where the states of replicas form a join-semilattice and are merged under the guarantees afforded by it— and *operation-based CRDTs*[2] —in which commutative operations are broadcast and applied to the local state by every replica. Additionally, δ*-CRDTs* are an optimization of state-based CRDTs to reduce the size of the payloads sent by the replicas.

Both Merkle-DAGs and CRDTs provide interesting properties: the former allows distributed systems to take advantage of a content-addressing layer for the resolution/discoverability and self-verification of data regardless of the source; the latter allows global-state convergence without the need of —usually complex and expensive— consensus mechanisms. By embedding CRDT objects inside Merkle-DAG nodes, we obtain the best properties of both worlds, that is, *we obtain a convergent system that can leverage the DAG as a logical clock*. This logical clock is provided and built by every replica, without the need for coordination and which can operate undisrupted in lose network environments with no delivery guarantees. As we will see, Merkle-CRDTs are fully agnostic to how the system announces and discovers data among replicas, thus being able to leverage different approaches like those provided by DHT and PubSub mechanisms without being tied to a particular version of them.

We conceive this approach as extremely useful for fully distributed peer-to-peer applications where the replicas are writers to a common dataset, usually in the form of a database. For example, a distributed and fully replicated file-system, chat group or package repository index. We have found that using IPFS (see Section 2.5) as a content-addressed, peer-to-peer decentralised file system and content distribution network, the system scales well to the order of thousands of replicas which can opportunistically join and depart – a very common condition when working with mobile and other low-power devices.

IPFS provides a content-addressed peer-to-peer filesystem [7] which supports seamless syncing of Merkle-DAGs with arbitrary formats and payloads, making it a robust building block for different types of distributed applications like PeerPad[3] or OrbitDB[4], both powered by CRDTs and IPFS.

In this paper we formalize what we refer to as *Merkle-CRDTs*. The goal

[1] Also known as *Convergent* CRDTs or *CvRDTs*.

[2] Also known as *Commutative* CRDTs or *CmRDTs*.

[3] PeerPad is realtime p2p collaborative editing tool (`https://peerpad.net`).

[4] OrbitDB is a peer-to-peer database for the decentralized web (`https://github.com/orbitdb/orbit-db`).

is to provide an overview of their properties, advantages and limitations, so that it can set the ground layer for future research and optimizations in the space.

As such the contributions of this paper are as follows:

- We define *Merkle-Clocks*, Merke-DAG-based logical clocks, to represent causality information in a distributed system. Embedding causality information using Merkle-DAGs is at the core of cryptocurrencies and source control systems like Git, but they are rarely considered separately as a type of logical clock. We demonstrate that Merkle-Clocks can be used in place of other logical clocks traditionally used by CRDTs like version vectors and vector clocks. We show that Merkle-Clocks can in fact be seen as CRDT objects themselves, which can be synced, merged and for which we can formally prove eventual consistency across different replicas.
- We define *Merkle-CRDTs* as a general purpose transport and persistency layer for CRDT payloads which leverages the properties of Merkle-Clocks, using the DAG-Syncer and the Broadcaster to provide per-object causal consistency by design. This enables the use of simple CRDT types in systems with weak messaging layer guarantees and large number of replicas.

The rest of the paper is organised according to the below. In Section 2, we start by introducing relevant background concepts and known research, in a way that can be easily understood by the reader, even when first approaching the field.

In Section 3, we expose the characteristics of our system model and introduce the facilities needed to store and sync Merkle-CRDTs. These are the *DAG-Syncer* and the *Broadcaster* components, both of them agnostic to the data payloads. While these components are conveniently available in the IPFS stack, we present them as an implementation-agnostic interface.

In Section 4, we introduce *Merkle-Clocks*, and building on the previous sections, in Section 5 we define *Merkle-CRDTs.* We discuss how different CRDT payloads (whether operation-based, state-based or δ-based) benefit from Merkle-CRDTs. Finally, we describe some of the limitations and inefficiencies of Merkle-CRDTs and introduce techniques to overcome them.

2 Background

2.1 Eventual consistency

The *Consistency, Availability, Partition-Tolerance* theorem, most widely known as the "CAP Theorem" [8] establishes that, in a distributed system, it is impossible to simultaneously obtain consistency, availability and partition-tolerance when it comes to maintaining a shared state.

This can be intuitively understood: if all replicas in the system accept arbitrary writes (Availability condition) during a network partition that keeps them from contacting one another (Partition Tolerance condition), there is no way that they can synchronize to a consistent state (Consistency condition). If the replicas instead stop accepting writes, they will maintain consistency but cannot be considered to be available. Consequently, replicas in a system in which partitions are tolerated cannot remain both consistent and available.

Since all three properties would be ideal to have in a distributed system, one way to get around the problem is to relax the consistency part and replace it with *eventual consistency* (EC)[5] [28], meaning that, at a certain moment, the state may not be the same across replicas —in fact it may be completely different— but, given enough time and perhaps after network partitions, downtimes and other eventualities have been resolved, the system design will ensure that the state becomes the same everywhere.

The main weakness of the eventual consistency definition is that it offers no guarantees as to when the shared state will converge or how much the individual states will be allowed to diverge until then[6]. *Strong eventual consistency* (SEC) addresses these issues by establishing an additional safety guarantee: if two replicas have received the same updates, their state will be the same.

Consensus algorithms or, more important to this paper, Conflict-Free Replicated Data Types (CRDTs) are ways to achieve (strong) eventual consistency in a distributed system.

2.2 Merkle DAGs

A *Direct Acyclic Graph (DAG)* is a type of graph in which edges have direction and cycles are not allowed. For example, a linked list like $A \to B \to C$ is an instance of a DAG where A references B and so on. We say that B is *a child* or *a descendant of A*, and that *node A has a link to B*. Conversely A is a *parent of B*. We call nodes[7] that are not children to any other node in the DAG as the *root nodes*.

A Merkle-DAG is a DAG where each node has an identifier and this is the result of hashing the node's contents —any opaque payload carried by the node and the list of identifiers of its children— using a cryptographic hash function like SHA256. This brings some important considerations:

[5] Also known as *optimistic replication*.

[6] EC only provides a *liveness* guarantee: the system will not become stuck when making progress to converge.

[7] Throughout the paper, we use the term *replica* to refer to the physical machine of a network node and *node* to refer to bundled content addressed by a single *identifier*.

a) Merkle-DAGs can only be constructed from the leaves, that is, from nodes without children. Parents are added after children because the children's identifiers must be computed in advance to be able to link them.

b) every node in a Merkle-DAG is the root of a (sub)Merkle-DAG itself, and this subgraph is *contained* in the parent DAG[8].

c) Merkle-DAG nodes are *immutable*. Any change in a node would alter its identifier and thus affect all the ascendants in the DAG, essentially creating a different DAG.

Identifying a data object (like a Merkle-DAG node) by the value of its hash is referred to as *content addressing*. Thus, we name the node identifier as *Content Identifier* or CID.

For example, in the previous linked list, assuming that the payload of each node is just the CID of its descendant would be: $A = Hash(B) \rightarrow B = Hash(C) \rightarrow C = Hash(\emptyset)$. The properties of the hash function ensure that no cycles can exist when creating Merkle-DAGs[9].

Merkle-DAGs are *self-verified* structures. The CID of a node is univocally linked to the contents of its payload and those of all its descendants. Thus two nodes with the same CID univocally represent exactly the same DAG. This will be a key property to efficiently sync Merkle-CRDTs without having to copy the full DAG, as exploited by systems like IPFS discussed later in Section 2.5.

Merkle-DAGs are very widely used. Source control systems like Git [11] and others [6] use them to efficiently store the repository history, in a way that enables de-duplicating the objects and detecting conflicts between branches.

In distributed databases like Dynamo [13], Merkle-Trees are used for efficient comparison and reconciliation of the state between replicas. In Hash Histories [16], content-addressing is used to refer to a Merkle-Tree representing a state[10].

Merkle-DAGs are also the foundational block of blockchains —they can be seen as a Merkle-DAG with a single branch— and their most common

[8] Merkle-DAGs are similar to Merkle Trees [20] but there are no balance requirements and every node can carry a payload. In DAGs, several branches can re-converge or, in other words, a node can have several parents, or be part of several Merkle DAGs.

[9] Hash functions are one way functions. Creating a cycle should then be impossibly difficult, unless some weakness is discovered and exploited.

[10] Hash Histories use a DAG to track the history of events in every replica. They decouple the size of causal information from the number of replicas like Merkle-Clocks, later presented here, but without using Merkle-DAGs. The nodes carry the hash of the state and an epoch number, in order to distinguish states which share the same hash at different moments in the history. With this information, replicas can establish if their versions of the state are dominant, exploit coincidental causality or extract deltas for diffing and merging.

application: cryptocurrencies. Cryptocurrencies like Bitcoin [21] benefit from the embedded causality information encoded in the chain: transactions in a block deeper in the chain always happened before those of earlier blocks. One of the main issues in cryptocurrencies is to make all participating peers agree about the tip/head/root of the chain. Among other things, the non-commutative nature of some transactions, like those originating from the same wallet[11], requires a consensus mechanism which enforces that only valid blocks become the new roots.

There are also DAG-based cryptocurrencies[12] like *DAG*[13], *Byteball*[14] or *IOTA*[15]. Like Merkle-CRDTs, they use a full-featured Merkle-DAG instead of a single chain. But, similarly to the rest, they end up needing to order conflicting transactions to ensure they follow the rules.

One commonality in many of these systems is that the Merkle-DAG implicitly embeds causality information[16]. The DAG can show that a certain transaction precedes another, or that a Git commit needs to be merged rather than fast-forwarded. This will be one of the properties that we use in Merkle-CRDTs and that this paper makes explicit and puts in contrast with other causality-encoding mechanisms known as *logical clocks*.

2.3 Logical clocks

The design of causally-convergent systems involves the reconciliation of diverging state versions among different replicas when, for example, events occur concurrently. This requires that we are able to identify whether two events actually happened concurrently and whether two states are actually different because of concurrent updates or other reasons, such as one replica having received more updates.

The problem is, essentially, tracking the order in which different events happened. For example, given multiple writes of a value to a register in different replicas, we would expect the final value in the registry to be that of the *last* write.

[11] A wallet must necessarily receive currency before being allowed to spend it.

[12] Also called *Blockless cryptocurrencies*

[13] `https://dagcoin.org`.

[14] `https://byteball.org`. Byteball's DAG [12] introduces the notion of *main chains* to order otherwise non-serial nodes in the DAG. How to build those chains in a way that they form a stable global view of causality is the main body of the *Byteball* specification.

[15] `https://iota.org`. In IOTA's *Tangle* [24], each node in the DAG represents a transaction which approves the transactions of its children and is approved by its parent. If a transaction B is part of a subDAG of A, then A *indirectly approves* B. The tip selection algorithm (which selects which transactions to approve) and the requirement that each peer needs to solve a cryptographic puzzle before issuing new transactions are the keys to establish order among concurrent transactions.

[16] The term *Causal Trees* denotes the same thing but refers to non-merkle tree structures and we rarely found it in literature related to distributed computing.

Ideally, we should be able to order all the events in the system[17] so that we can identify which was the actual *last* update to the register.

Tagging events with timestamps can give us this information: if all events are timestamped, any replica may establish the order in which they happened and use that information to decide what the final state should look like. However, in distributed systems, it is not possible to use timestamps reliably [22], as not every replica can be perfectly synced to a global time. "Wall clocks" can also easily be simulated or spoofed, which is problematic in peer-to-peer systems with no trust involved.

Logical clocks are the alternative to global time. They provide ways to encode causal information between events known to different actors in a distributed system.

The basic idea is that, although we may not know the order in which all events happened globally, every replica knows at least the order of events issued by itself. Any other replica that receives that information will then know that any events later issued by itself come after those. This is, in essence, what is known as *causal history.*

Logical clocks are representations of causal histories [5] which provide a *partial ordering* between events. That is, given two events a and b, logical clocks should be able to tell us if *a happened before* b ($a \to b$), or vice-versa ($b \to a$), or if both a and b happened concurrently ($a \parallel b$)[18].

The practical implementation of logical clocks usually involves metadata which travels attached to every event in the system. One of the most common forms of logical clocks are *version vectors* [23]: every replica maintains and broadcasts a vector that tracks on which version the state of all the replicas is. When a replica performs a modification of the state, it increases its version. When a replica merges a state from a different replica, it takes the highest between the local versions and the versions provided by the other replica along with the event. Thus, given two events a, b, with version vectors $\mathcal{V}^a$, $\mathcal{V}^b$: $a \to b$ if $\mathcal{V}^a_i \leq \mathcal{V}^b_i$ for each position i in the vectors. If $a \not\to b$ and $b \not\to a$, by that definition, a and b are concurrent.

As we see, version vectors are compact because they do not need to store the full causal history but merely a number indicating how long the history

[17] This means establishing a *total strict order* for all the events.

[18] We take a number of shorcuts in this description. Logical clocks were originally described by Lamport [18] as a function which, for every event, returns a value so that:

$$a \to b \Rightarrow Clock(a) < Clock(b)$$

While this can already be used to obtain a total order among the events in a system, as shown by the Lamport scalar clock, above we refer to logical clocks that meet the *Strong Clock condition* (which is two-way):

$$a \to b \Leftrightarrow Clock(a) < Clock(b)$$

is for every replica. Version vectors depend on the number of replicas, so they may need further optimizations to work well in scenarios with many replicas or where the number of replicas is not stable.

In addition to many proposed improvements, there are multiple types of logical clocks that are similar to version vectors but fulfil different needs or address some of their shortcomings: vector clocks [15], bounded version vectors [1], dotted version vectors [25], tree clocks [19] or interval tree clocks [2] are some of them.

In this paper we formalize that a Merkle-DAG can act as a logical clock and therefore replace some of the clocks above. *Merkle-Clocks*, as we will show, provide a different set of properties but encode the same causal information about events.

2.4 Conflict-Free Replicated Data Types (CRDTs)

CRDTs are data types which provide *strong* eventual consistency among different replicas in a distributed system by requiring certain properties from the state and/or the operations that modify it. Additionally, CRDTs also feature monotonicity. The concept of monotonicity applied to data types is the notion that every update is an inflation, making the state grow, not in size, but in respect to a previous state. This implies that there will always be an order between states[19]. Monotonicity implies that rollbacks on the state are not necessary regardless of the order in which updates happen.

There are two prominent types of CRDTs: *state-based* and *operation-based* CRDTs. In state-based CRDTs, all the states in the system —that is, the states in different replicas and different moments— form a monotonic join-semilattice. That means that, for any two states X and Y, both can be *"joined"*[20] ($\sqcup$) and the result is a new state corresponding to the Least-Upper-Bound (LUB) of the two [26]. In other words, every modification made to a state by a replica must be an inflation and the union of two states X and Y is the minimal state capable of containing both X and Y and not more (the LUB). A join-semilattice is thus a partially ordered set[21] and its LUB is the smallest state capable of *containing* all the states in the semilattice. This implies that the $\sqcup$ operation must be idempotent ($X \sqcup X = X$), commutative ($X \sqcup Y = Y \sqcup X$) and associative ($(X \sqcup Y) \sqcup Z = X \sqcup (Y \sqcup Z)$).

Replicas in a state-based CRDTs modify their state —or inflate it— and broadcast the resulting state to the rest of replicas[22]. Upon receiving the

[19] A good example is that a CRDT counter which can be increased and decreased (known as PN counter) is necessarily implemented using two counters which can only be increased.

[20] Also denoted *"union"* or *"merge"*.

[21] See `https://en.wikipedia.org/wiki/Partially_ordered_set`.

[22] An important note here is that CRDTs are just data types. The transmission of CRDT objects between replicas goes beyond it. Some CRDTs are, by design, better suited to some broadcasting mechanisms than others and can facilitate optimizations such

state, the other replicas *merge* it with the local state[23]. The properties of the state ensure that, if the replicas have correctly received the states sent by other replicas —and vice-versa—, they will eventually converge.

Operation-based CRDTs [26], on the other side, do not enforce any property on the state itself but on the operations used to modify it, which must be commutative[24]. The replicas broadcast the operations and not the states. If two operations happen at the same time in two replicas, the order in which other replicas apply them does not matter: the resulting states will be the same.

It follows that, if an operation broadcast does not arrive to a replica —for example due to a network failure—, that replica will never be able to apply it and the states will not converge. Thus, unlike state-based CRDTs, eventual consistency in operation-based CRDTs requires a reliable messaging layer that eventually delivers all operations [4]. Additional constraints may be necessary, for example, if operations are not idempotent: in that case the messaging layer should ensure that each operation is delivered exactly once. Some operation-based CRDTs may also require causal delivery: if a replica sends operation a before b ($a \rightarrow b$), then a should always be delivered before b to a different replica.

These properties and requirements in both state and operation-based CRDTs ensure *per-object causal consistency*: updates to a state will maintain the causal relations between them. For example, in a Grow-Only Set (G-Set), when a replica adds element A and then element B, every other replica will never have a set where B is part of the set but A is not[25].

Logical clocks, as seen in the previous section, are commonly used to implement CRDT types: they are useful to identify when two updates happen concurrently and need merging.

CRDTs have been successfully used and optimized in different applications and distributed databases, Basho's Riak [9, 10] being one of the most prominent examples[26].

2.5 IPFS: The InterPlanetary File System

IPFS [7] is a content-addressed, distributed filesystem. IPFS uses a Distributed Hash Table (DHT) to announce and discover which replicas (or

as broadcasting only to a random subset rather than to every replica.

[23] The *merge* can take several forms. In a CRDT counter, merging involves taking the maximum between the local and the remote values.

[24] At least in regard to a different operation issued at the same time (concurrently).

[25] This is clear for an operation-based implementation of a G-Set (assuming causal delivery of the operations). The state-based implementation of a G-Set involves sending the full set. Thus, the event adding B is a set which already contains A: there will not be a set where B is present but not A, even if the event that added A was lost or arrives later.

[26] `https://github.com/ipfs/research-CRDT/issues/40` provides other examples.

peers) provide certain Merkle-DAG nodes. It implements a node-exchange protocol called "bitswap" to retrieve DAG nodes from any *provider*.

IPFS is built on top of libp2p[27], a modular network protocol stack for P2P networks, which additionally provides efficient broadcasting mechanisms primarily based on *publish-subcribe* models[28].

IPFS also uses IPLD, the *InterPlanetary Linked Data Format*[29], a framework to describe Merkle-DAGs with arbitrary node formats and support for multiple types of CIDs[30], making it very easy to create and sync custom DAG nodes.

These features make IPFS a suitable layer on which to implement Merkle-CRDTs, as it provides the necessary mechanisms to discover, route and announce content in potentially very large networks.

3 System model & Assumptions

Our Merkle-CRDT approach is intended to be both simple and facilitate the use of CRDTs in peer-to-peer distributed systems with large number of replicas and no message delivery guarantees (i.e., unreliable transports).

We assume the presence of an asynchronous messaging layer which provides a communication channel between separate replicas. This channel is managed by two facilities which every replica exploits: the *DAG-Syncer* and the *Broadcaster* components (defined below).

We assume that messages can be dropped, reordered, corrupted or duplicated. It is not necessary to know beforehand the number of replicas participating in the system. Replicas can join and leave at will, without informing any other replica. There can be network partitions but they are resolved as soon as connectivity is re-established *and* a replica broadcasts a new event.

Replicas may have durable storage, depending on their own requirements and data types. Using Merkle-CRDTs new replicas and crashed replicas without durable storage will be able to *eventually re-construct the complete state of the system as long as at least one other replica is in the latest system state.*

3.1 The DAG-Syncer component

A *DAG-Syncer* is a component which enables a replica to obtain remote Merkle-DAG nodes from other replicas given their content identifiers (CIDs)

[27] `https://libp2p.io`).

[28] As of this writing, Floodsub and gossipsub (`https://github.com/libp2p/go-libp2p-pubsub`).

[29] For specifications and description, see `https://ipld.io`.

[30] The Multiformats project provides self-describing values for future-proofing (`https://multiformats.io/`).

and to make its own nodes available to other replicas. Since a node contains links to their direct descendants, given the root node's CID, the DAG-Syncer component can be used to fetch the full DAG by following the links to children in each node. Thus, we can define the DAG-Syncer as follows:

Definition 1. (DAG-Syncer). A DAG-Syncer is a component with two methods:

- `Get(CID) : Node`
- `Put(Node)`

We do not specify any more details such as how the protocol to announce and retrieve nodes looks like. Ideally, the DAG-Syncer layer should not impose any additional constraints on the system model. Our approach relies on the properties of the DAG-Syncer and Merkle-DAGs to tolerate all the network contingencies described above.

3.2 The Broadcaster component

A *Broadcaster* is a component to distribute arbitrary data from one replica to all others[31]. Ideally, the payload will reach every replica in the system, but this is not a requirement for every broadcast message:

Definition 2. (Broadcaster). A Broadcaster is a component with one method:

- `Broadcast(Data)`

3.3 IPFS as a DAG-Syncer and Broadcaster component

The components above can be realised by using the *InterPlanetary File System* (IPFS) [7] (as introduced in Section 2). IPFS can act as the DAG-Syncer, while one of the PubSub mechanisms provided by its *libp2p* layer can perform the tasks of the Broadcaster component.

Such an implementation should allow extreme scalability of the replica set in general. The peers in the network do not need to be fully connected to everyone else and the system is extremely modular and configurable to fit both small devices and large storage servers. The choice of settings and implementations will affect the performance of the system under different circumstances and network topologies, but is independent from the Merkle-CRDT objects and datatype as long as it provides the necessary components.

[31] The broadcasting strategy may or may not involve delivering the messages directly to other replicas. Messages could also be relayed.

4 Merkle-Clocks

4.1 Overview

A Merkle-Clock $\mathscr{M}$ is a Merkle-DAG where each node represents an event. In other words, given an event in the system, we can find a node in this DAG that represents it and that allows us to compare it to other events.

The DAG is built by merging other DAGs (those in other replicas) according to some simple rules. New events are added as new root nodes (parents to the existing ones)[32].

For example, given $\mathscr{M}_\alpha$ and $\mathscr{M}_\beta$ (α and β being the single root CIDs in those DAGs[33]):

1. If $\alpha = \beta$ no action is needed, as they are the same DAG.

2. else if $\alpha \in \mathscr{M}_\beta$, we keep $\mathscr{M}_\beta$ as our new Clock, since the history in $\mathscr{M}_\alpha$ is part of it already. We say that $\mathscr{M}_\alpha < \mathscr{M}_\beta$ in this case.

3. else if $\beta \in \mathscr{M}_\alpha$, we keep $\mathscr{M}_\alpha$ for the same reason. We say that $\mathscr{M}_\beta < \mathscr{M}_\alpha$ in this case.

4. else, we *merge* both Clocks by keeping both DAGs as they are and thus having two root nodes, those referenced by α and β. Note that $\mathscr{M}_\alpha$ and $\mathscr{M}_\beta$ could be fully disjoint or not, depending on whether they share some of their deeper nodes. If we wish to record a new event, we can do so by creating a new root γ with two children, α and β.

We can already see that, by looking if one Merkle-Clock is included in another, we are introducing the notion of *order among Clocks.* In the same way, we have a notion of order among the nodes in each clock, since events that happened earlier will always be descendants of events that happened later. Additionally, we have introduced a way *to merge Merkle-Clocks according to how they compare.* The resulting Clock always includes the causality information from both Clocks. This eventually means that *the causality information stored in Merkle-Clocks in every replica will converge to the same Merkle-Clock after merging.*

The causal order provided by Merkle-Clocks is embedded when building Merkle-DAGs with similar rules and usually overlooked as something very intuitive. It is important, however, to formalize how we define order between Merkle-Clocks and to prove that the causality information is maintained when they are synced and merged. This is the subject of the next section and will be an important property for Merkle-CRDTs.

[32] Root nodes of the DAG are nodes without any parents. The Merkle Clock may have several roots at a given time.

[33] In the example we assume, without loss of generality, that we start with DAGs containing a single root instead of several.

4.2 Merkle-Clocks as a convergent, replicated data type

This section formalizes the definition of Merkle-Clocks and their representation as Merkle-Clock DAGs. We will show that Merkle-Clock DAGs can be seen as a Growing-Set (G-Set) CRDT and therefore converge in multiple replicas[34].

Let $\mathscr{S}$ be the set of all system events:

Definition 3. (Merkle-Clock Node). A Merkle-Clock Node n_α is a triple:

$$(\alpha, e_\alpha, \mathcal{C}_\alpha)$$

which represents an event $e_\alpha \in \mathscr{S}$, with α being the node CID and $\mathcal{C}_\alpha$ being the CID-set of the direct desdendants of n_α.

Definition 4. (Merkle-Clock DAG). A Merkle-Clock DAG is a pair:

$$\langle \mathbb{N}, \leq \rangle$$

where $\mathbb{N}$ is a set of immutable DAG-nodes and a partial order $\leq$ on $\mathbb{N}$, defined as follows:

$$n_\alpha, n_\beta \in \mathbb{N} : n_\alpha < n_\beta \Leftrightarrow n_\alpha \text{ is a descendant of } n_\beta$$

In other words, $n_\alpha < n_\beta$ if there is a path of linked nodes which goes from n_β to n_α.

In order to maintain this relationship, the Merkle-Clock DAG must be built with the following *Implementation Rule*:

IR. Every new event in the system must be represented as a new root node to the existing Merkle-Clock DAG(s). In particular, the $\mathcal{C}$ set must contain the CIDs of the previous roots.

Definition 5. (Merkle-Clock). A Merkle-Clock ($\mathscr{M}$) is a function which given an event $e_\alpha \in \mathscr{S}$ returns a node from the Merkle-Clock DAG $\mathbb{N}$:

$$\mathscr{M} : \mathscr{S} \to \mathbb{N}$$

Remark. A Merkle-Clock satisfies the *Strong Clock condition* [18]. We see that every node represents a later event than that of its children:

$$\forall (\beta, e_\beta, \mathcal{C}_\beta) \in \mathbb{N} : \forall \alpha \in \mathcal{C}_\beta : e_\alpha \to e_\beta$$

[34] It is usually not mentioned that other common logical clocks are also CRDTs and were invented even before the term was coined. In particular, the operation of a vector clock is very similar to that of a state-based G-Counter CRDT and it is, in fact, just that: a grow-only counter that represents causality.

Since every event is the root of a (sub)DAG built using the implementation rule, we can immediately see that earlier Merkle-Clock values are descendants of the later ones:

$$\mathscr{M}(e_\alpha) < \mathscr{M}(e_\beta) \Leftrightarrow e_\alpha \to e_\beta$$

We can now define a *join-semilattice of Merkle-Clocks DAGs* as a pair:

$$\langle \mathbb{J}, \subseteq_{\mathbb{J}} \rangle$$

where $\mathbb{J}$ is a set of Merkle-Clocks DAGs and $\subseteq_{\mathbb{J}}$ a partial order over that set defined as follows. Given $\mathbb{M}, \mathbb{N} \in \mathbb{J}$:

$$\mathbb{M} \subset_{\mathbb{J}} \mathbb{N} \Leftrightarrow \forall m \in \mathbb{M}, \exists n \in \mathbb{N} \mid m < n \Leftrightarrow \mathbb{M} \subset \mathbb{N}$$

Note that $m < n$, means that m is a descendant of n and thus must belong to the same DAG, then $\subset_{\mathbb{J}}$ simply means that $\mathbb{M}$ is a subset of $\mathbb{N}$.

This allows us to define the Least-Upper-Bound of two Merkle-Clocks DAGs ($\sqcup_{\mathbb{J}}$) as the regular union of the sets:

$$\mathbb{M} \sqcup_{\mathbb{J}} \mathbb{N} = \mathbb{M} \cup \mathbb{N}$$

Unsurprisingly, the Merkle-Clock representation corresponds in fact to a Grow-Only-Set (G-Set) in the state-based CRDT form [27]. The elements of the set are immutable, cryptographically linked and represent the events in the system. When the DAGs are disjoint, the resulting DAG will include the roots from both $\mathbb{N}$ and $\mathbb{M}$. That is the equivalent of having several events without causal relationship. Causality information about DAG-merge events can be optionally included after the union of the DAGs by creating a new unique root following the *implementation rule.*

In the next section we will see how the properties of Merkle-DAGs allow syncing Merkle-Clocks in a more efficient manner than regular state-based G-Sets.

4.3 The Merkle in the Clocks: properties of Merkle-Clocks

We have so far defined a way to encode causality information per replica and ensured that two replicas can merge their Merkle-Clocks. Now we will see how the properties of Merkle-DAGs allow the use of a *pull* (or *fetch*) approach, rather than a *push* approach which, together with content-addressing, enables efficient clock sync between replicas and overcomes the effect of network partitions or contingencies. The steps to Merkle-Clock synchronisation between replicas are given below.

1. Broadcasting the Merkle-Clock requires broadcasting only the current root CID. The whole Clock is unambiguously identified by the CID of its root and its full DAG can be walked down from it as needed.

2. The immutable nature of a Merkle-DAG allows every other replica to perform quick comparisons and pull/fetch only those nodes that it does not already have.
3. Merkle-DAG nodes are self-verified, through their CID, and, therefore, immune to corruption and tampering. Hence, they can be fetched (pulled) from any source willing to provide them, trusted or not.
4. Identical nodes are de-duplicated by design: there can only be one unique representation for every event.

In practice, every replica just fetches the *delta* causal histories from other replicas without the need to build those deltas explicitly anywhere in the system. A completely new replica with no previous history will fetch the full history automatically[35].

Merkle-Clocks can replace version clocks and other logical clocks that are usually part of CRDTs. This comes with some considerations:

- By using Merkle-Clocks we can *decouple the causality information from the number of replicas*, which is a common limitation in version clocks. This makes it possible to reduce the size of the messages when implementing CRDTs and, most interestingly, solves the problem of keeping clocks working when replicas randomly join and leave the system.
- On the downside, the causal information grows with every event and replicas store potentially large histories even if the event information is consolidated into smaller objects.
- Keeping the whole causal history enables new replicas to sync events from scratch out-of-the-box, without having to explicitly send system snapshots to newcomers. However, that syncing may be slow if the history is very large. We will explore, along with Merkle-CRDTs, potential optimizations in this regard.

A significant advantage of Merkle-Clocks, over traditional version clocks is that they can also deal with network eventualities without much trouble:

- Dropped messages may prevent informing other replicas about new roots. But since every Merkle-Clock DAG is superseeded by future DAGs and every download fetches all the missing parts of a DAG, network partitions and replica downtimes do not have an effect on the overall system and will begin to heal automatically once the issues are resolved.

[35] This is precisely how peers participating in cryptocurrencies sync their ledgers.

- Messages arriving unordered pose no problem for the same reasons. The missing DAG will be fetched and processed in order.
- Duplicated messages are just ignored by replicas as they are already incorporated into their Merkle-Clocks.
- Corrupt messsages come in two fashions: a) if the message broadcasting a new root is corrupted, then it will be a hash corresponding to a non-existent DAG that cannot be fetched by the DAG-Syncer and will be eventually ignored; b) if a DAG node is corrupted on download, the DAG-Syncer component (or the application) can discard it if its CID does not match the downloaded content.

As we showed in the previous section, 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 first*.

A total order can be useful [18] 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 of their nodes or by any other arbitrary user-defined strategy based on additional information attached to the clock nodes. Any such approach would qualify as *data-layer conflict resolution.*

5 Merkle-CRDTs: Merkle-Clocks with payload

Definition 6. (Merkle-CRDT). A Merkle-CRDT is a Merkle-Clock whose nodes carry an arbitrary CRDT payload.

Merkle-CRDTs keep all the properties seen before for Merkle-Clocks. However, for the payloads to converge, they need to be convergent data types (CRDTs) themselves. The advantage is that Merkle-Clocks already embed ordering and causality information which would otherwise need to travel embedded in the CRDT objects[36] or be provided by a reliable messaging layer.

Thus, the implementation of a Merkle-CRDT node looks like:

$$(\alpha, P, \mathcal{C})$$

with α being the *content identifier*, P an opaque data object with CRDT properties and $\mathcal{C}$ the set of children identifiers[37].

[36] Usually in the form of other logical clocks.

[37] In the previous section we defined Merkle-Clock nodes as a triple $(\alpha, e, \mathcal{C})$. We included the event e to facilitate the definition of node ordering but it is easy to see that the causality information is directly embedded in the Clock: the existence of a node is the event itself.

5.1 Per-object Causal Consistency and Gap Detection

The directed-link nature of Merkle-CRDTs, which allows traversing the full causal history of the system in the order of events, provides all the necessary properties to ensure per-object *causal consistency* and *gap detection* by design without modifying our system model.

This means that Merkle-CRDTs are very well suited to carry operation-based CRDTs as they can ensure that no operation is lost or applied in disorder[38].

To facilitate the task of processing CRDT payloads in Merkle-CRDTs, in the next section we present a general and simple (non-optimized) anti-entropy algorithm that can be used to obtain per-object causal consistency for any CRDT embedded object.

5.2 General anti-entropy algorithm for Merkle-CRDTs

Definition 7. (General anti-entropy algorithm for Merkle-CRDTs).

Let $\mathcal{R}^A$ and $\mathcal{R}^B$ be two replicas using Merkle-CRDTs with $\mathscr{M}_\alpha$ and $\mathscr{M}_\theta$ respectively as their current Merkle-CRDT DAG.

1. $\mathcal{R}^B$ issues a new payload by creating a new DAG node $(\beta, P, \{\theta\})$ and adding it as the new root to its Merkle-CRDT, which becomes $\mathscr{M}_\beta$.

2. $\mathcal{R}^B$ broadcasts β to the rest of replicas in the system.

3. $\mathcal{R}^A$ receives the broadcast of β and retrieves the full $\mathscr{M}_\beta$. It does this by starting from the root β and walking down the DAG using the DAG-Syncer component to fetch all the nodes that are not in $\mathscr{M}_\alpha$, while collecting their CIDs in a CID-Set $\mathcal{D}$. Given the inherent properties of DAGs, for any CID already in $\mathscr{M}_\alpha$ the whole sub-DAG can be skipped.

4. If $\mathcal{D}$ is empty, no further action is required. $\mathcal{R}^A$ must have already processed all the payloads in $\mathscr{M}_\beta$. This means that $\mathscr{M}_\beta \subseteq \mathscr{M}_\alpha$.

5. If $\mathcal{D}$ is *not* empty, we sort the CIDs in $\mathcal{D}$ using the order provided by the Merkle-Clock[39]. We can skip the ordering if causal delivery is not a requirement in our system. The amount of items in $\mathcal{D}$ will depend on the amount of concurrency in the system and how long the two Merkle-CRDTs have been allowed to diverge, but should be small under normal circumstances.

[38] To re-iterate, the Merkle-Clock provides a strict partial order of events. In this case, two non-concurrent operations applied to an object will be sortable by the clock.

[39] To be precise, we are extending the order to a *total* order by considering incomparable nodes to be “equal”.

6. $\mathcal{R}^A$ processes the payloads associated with the nodes corresponding to the CIDs in $\mathcal{D}$, from the lowest to the highest.

7. If $\alpha \in \mathcal{D}$, then $\mathscr{M}_\alpha \subseteq \mathscr{M}_\beta$ and $\mathscr{M}_\beta$ becomes the new local Merkle-CRDT in $\mathcal{R}^A$.

8. else, $\mathscr{M}_\alpha \not\subseteq \mathscr{M}_\beta$ and $\mathscr{M}_\beta \not\subseteq \mathscr{M}_\alpha$. $\mathcal{R}^A$ keeps both nodes as roots.

5.3 Operation-based Merkle-CDRTs

Definition 8. Operation-based Merkle-CRDTs are those in which nodes embed an operation-based CRDT payload.

Operation-based Merkle-CRDTs are the most natural application of Merkle-CRDTs. Operations are easy to define, as they just need commutativity but, in their traditional form, require a reliable messaging layer [4] or complex workarounds, like additional causality payloads, buffering and retry mechanisms.

Merkle-DAGs provide all the properties of a messaging layer where messages are always delivered in order, verified and never repeated nor dropped. Thus, Merkle-CRDTs enable operation-based CRDTs in contexts where they could not be easily used before.

As we saw, thanks to the Merkle-DAG in which they are embedded, each replica only needs the missing parts of the DAG and these can be fetched once the root is known. This includes new replicas joining the system, which will be able to fetch and apply all operations. We do not need to keep knowledge of the full replica set and place the responsibility of efficient broadcast in the *Broadcaster* component.

It is worth noting that adding Lamport timestamps to each operation makes them usable to implement different replicated data types as proposed by the *OpSets*[40] *specifications* [17].

5.4 State-based Merkle-CRDTs

Definition 9. State-based Merkle-CRDTs are those in which nodes embed a state-based CRDT payload.

Embedding full states in each Merkle-CRDT node is counter-intuitive since state-based CRDTs already provide per-object causal consistency and can cope with unreliable message layers by design.

[40] OpSets introduce a replicated data type framework based on operations which are unique and stored as an ordered set based on their Lamport timestamps. The state is the interpretation of the full set. When operations arrive out-of-order, the state needs to be recomputed. OpSets bring some strengths at the cost of strong eventual consistency and space —all operations need to be stored in order to potentially re-compute the full state. Some OpSet types may benefit from a Merkle-CRDT transport which ensures causal delivery, potentially unlocking optimizations.

Moreover, although the final state would result from the merge of all the states in the Merkle-CRDT nodes, the *DAG-Syncer* component would still need to store those states, something prohibitive when working with large state objects. That said, Merkle-CRDTs remove the need to attach causality metadata and detach it from the number of replicas, which might be of interest for state-based CRDTs with very small states in comparison to the number of replicas.

A more interesting approach is that of δ-CRDTs [3] which, instead of broadcasting full states, are able to send smaller sections (deltas). δ-mutations, as these objects are called, can be merged downstream just like any full state would be, without the need for changing the semantics of the *union* operation. It follows that multiple deltas can be merged to form what is known as δ-groups and increase the efficiency of the broadcast payloads. As pointed out in [3], *"a full state can be seen as a special (extreme) of a delta-group"*.

In the vanilla form of δ-CRDTs, however, consistency is delayed ad-infinitum when a message is lost and the per-object causal consistency property of state-based CRDTs is lost. These issues can be addressed with an additional anti-entropy algorithm that groups, sorts, tracks delivery and resends missing deltas, as presented in [3], but in the case of δ-state-Merkle-CRDTs, the anti-entropy algorithm and any causal information attached to the original objects would not be necessary. In essence, this approach brings δ-state Merkle-CRDTs closer to their operation-based counterpart.

5.5 Limitations of Merkle-CRDTs

We have so far focused in explaining the different qualities that Merkle-CRDTs provide to traditional CRDT approaches, but we must also highlight what intrinsic and practical limitations they bring.

Ever-growing DAG-Size: The most obvious consequence of Merkle-CRDTs is that, while CRDTs normally merge, apply, consolidate and discard broadcast objects, Merkle-CRDTs build a permanent Merkle-DAG which must be stored and is ever-growing. As we have seen, this provides a number of advantageous properties, but also comes with some implications:

- The size of the DAG might grow larger than acceptable. The rate of growth will depend on the number of the events and the size of the payloads. This is very similar to how blockhains grow to large sizes in time[41]. This is especially problematic when the actual state might be much smaller. In some cases, it might be possible to express the

[41] Bitcoin chain uses more than 220GB and Ethereum (Parity) more than 165GB as of this writing.

state as a compact of the result of all the Merkle-CRDT operations, but this brings us to the next point.

- If replicas store the Merkle-DAG only, knowing that the full state can be rebuilt from it (and thus saving that space), starting replicas with very large Merkle-DAGs might be especially slow since they will need to reprocess the full DAG, even when available locally. If not, there will be redundant information stored in both the resulting state and in the Merkle-DAG.

- Merkle-CRDT syncs from scratch are possible and natural to the system when a new replica joins. However, Merkle-DAGs are not only ever-growing, but also tend to be deep and thin[42]. A new replica will learn the root CID from a broadcast operation and will need to resolve the full DAG from it. Because of the thinness, it will not be possible to fetch several branches in parallel. Cold-syncs may take significantly longer than it would take to ship a snapshot, thus rendering this embedded property of Merkle-DAGs of little value.

Very large DAGs and slow syncs are not a problem in some scenarios and can be seen as an acceptable trade-off, but do highlight the need of exploring garbage collection and DAG compaction mechanisms.

Merkle-Clock sorting: Merging two Merkle-Clocks requires comparing them to see if they are included in one another and finding differences. This may be a costly operation if DAGs have diverged significantly (or long ago).

DAG-Syncer latency: Replicas rely on a DAG-Syncer component to fetch and provide nodes from and to the messaging layer. To avoid keeping a static list of replicas participating in the system, peer-to-peer applications like Bittorrent and IPFS use a Distributed Hash Table (DHT)[43]. The DHT is used to collaboratively store and locate small pieces of information (*discovery*) and to discover peers and route other peers to them (*routing*). DHTs are massively scalable but introduce some overhead[44] that may make fetching DAG-nodes slower than receiving them directly from the issuer.

[42] The Merkle-DAGs will be thin in the absence of many concurrent events, or have a high branching factor otherwise. In both cases, branches are consolidated every time a new event is issued from a replica, thus creating *thin waists* in the DAG.

[43] The Wikipedia entry provides a good overview of how they work, out of the scope of this paper: `https://en.wikipedia.org/wiki/Distributed_hash_table`.

[44] This is particularly relevant when using an IPFS node connected to the global IPFS network, where the DHT will not just store the data associated to the Merkle-CRDT nor only be used by the replicas. It is possible, nevertheless, to use private IPFS networks (with a dedicated DHT) for the task.

The practical impact of these limitations depends on the requirements of the application. In particular, when thinking about adopting Merkle-CRDTs, users should consider whether Merkle-CRDTs are the best approach in terms of:

- Node count vs. state-size
- Time to cold-sync
- Update propagation latency
- Expected total number of replicas
- Expected replica-set modifications (joins and departures)
- Expected volume of concurrent events

In the following section we explore some optimizations which can address part of the problems seen here, but may also impose additional constraints.

5.6 Optimizing Merkle-CRDTs

The previous section listed some of the issues we must account for when using Merkle-CRDTs, especially in the vanilla, non-optimized version in which we have presented them. We will now describe potential optimizations to address some of those problems.

Delayed DAG nodes: In scenarios where replicas issue frequent updates, we can group multiple payloads before issuing a single node containing all of them. It is clear that this approach will bring some benefits, which however, comes together with tradeoffs: updates are not immediately sent out and will therefore, take longer to propagate.

Quick Merkle-DAG inclusion check: Merging the local replica DAGs with a remote one requires checking if one DAG includes the other. It is possible but inefficient to do so by walking down the first DAG looking for a node CID that matches the root of the second. Storing the CIDs of the local DAG in a key-value store that can quickly check whether a CID is part of the local DAG or not makes things significantly easier[45]. When walking the remote DAG to check for inclusion of the local DAG, the CIDs of the children of any of its nodes can be checked to see if they are part of the local DAG in which case their branches can be conveniently pruned. This implies, however, that the implementation must be aware and have access to the local storage system for nodes. The DAG-Syncer, as currently defined, cannot differentiate between nodes available locally or remotely. Bloom filters, caches and some data structures can also improve efficiency, but they are usually part of the chosen storage backend.

[45] Fast key-value stores, such as in-memory ones, will normally pay a high memory footprint penalty, while disk-backed ones will be slower.

A similar effect can be achieved by embedding *version vectors* in the payloads, as long as the application can tolerate the constraints they impose. Comparing version vectors between payloads is an inclusion check without the need to perform a DAG-walking.

Broadcast payload adjustments: Our standard approach reduces the size of the broadcasts by including only the CID of the new roots. Publishing mechanisms are complex enough and always benefit from smaller payloads.

However in some systems it may be beneficial[46] to send new Merkle-DAG nodes directly as broadcast payloads. Replicas that are offline or dropped messages will recover when they receive a future update and complete their DAGs, so this has no effects in that regard. Broadcasting the payloads (assuming they are small enough) will likely reduce the latency of the propagation of changes in the system.

Reducing the Merkle-DAG node size: We can attempt to reduce the size of the payloads as much as possible by compressing and removing redundant information not required by the CRDT itself. For example, instead of signing the CRDT payloads to ensure that they come from a trusted replica, we can sign the broadcast messages, thus leaving signatures out of the Merkle-DAG.

Another option is to make the payload (or parts of it) CIDs to reference the actual contents. If the payloads are big, this will greatly reduce the size of the Merkle-DAG and may increase the efficiency of the DAG fetching. This is especially relevant when some of the payloads are identical and can be de-duplicated.

Additional pointers in nodes: One of the ways to work around the thin-DAG problem is to regularly introduce references to deeper parts of the DAG when issuing new nodes. This is basically adding extra children to nodes. It allows more parallelism when fetching missing parts of the DAG by being able to *jump* to other sections of it. This can result in a much faster traversal. The actual number of extra links and their destination will depend on the needs of the application.

The above recommendations should be considered in any Merkle-CRDT implementation as they may provide significant advantages over the unoptimized version described previously. We leave the topics of DAG compaction and garbage collection for future work, although we intuitively note that discarding parts of the Merkle-DAG is not possible without knowing if every replica is aware of them. This, in turn, requires knowing the replica-set[47], a system constraint that we did not have before.

[46] Specially those with a rather small replica set and fast broadcast.

[47] Or agreeing, using some form of consensus or authority.

6 Related work in the IPFS Ecosystem

Merkle-CRDTs are very intuitive, even if they were not formalized before, and rely on well-known and widely used properties of Merkle-DAGs. Several projects in the IPFS ecosystem already use them[48], all embedding operation-based CRDTs in Merkle-DAGs:

- `ipfs-log`[49] is, to our knowledge, the first existing instance of a Merkle-CRDT as described here. It implements an operation-based, append-only log CRDT (similar to a grow-only set).
- `ipfs-hyperlog`[50] is utility to build and replicate Merkle DAGs.
- `Orbit DB`[51] is a distributed, peer-to-peer database. It uses `ipfs-log` and other CRDTs for different data models. It is used to build `Orbit`[52], a distributed, serverless chat application.
- `Tevere`[53] is an operation-based Merkle-CRDT key-value store.
- `peer-crdt`[54] and `peer-crdt-ipfs`[55] provide a generalistic operation Merkle-CRDT implementations of several CRDTs: counters, sets, arrays, registers and text (as well as composable CRDTs).
- `versidag`[56] is a proposed linked log with conflict resolution to store version information, similar to `ipfs-log`.
- `PeerPad`[57] is a real-time collaborative text editor based on `peer-crdt` and δ-CRDTs.
- `Textile.photos`[58] is a mobile, decentralized digital wallet for photos. Textile Threads (v1) [14] allow a group of users to share photos without a central database and are based on Merkle-CRDTs.
- `go-ds-crdt`[59] is a key-value distributed datastore implementation in Go using δ-state Merkle-CRDTs. It is used by IPFS Cluster[60].

[48] The dynamic data and capabilities working group has started many discussions on the topic: `https://github.com/ipfs/dynamic-data-and-capabilities`.

[49] `https://github.com/orbitdb/ipfs-log`

[50] `https://github.com/noffle/ipfs-hyperlog`

[51] `https://github.com/orbitdb/orbit-db`

[52] `https://github.com/orbitdb/orbit`

[53] `https://github.com/ipfs-shipyard/tevere`

[54] `https://github.com/ipfs-shipyard/peer-crdt`

[55] `https://github.com/ipfs-shipyard/peer-crdt-ipfs`

[56] `https://github.com/ipfs/dynamic-data-and-capabilities/issues/50`

[57] `https://github.com/ipfs-shipyard/peer-pad`

[58] `https://www.textile.photos/`

[59] `https://github.com/ipfs/go-ds-crdt`

[60] `https://cluster.ipfs.io`

7 Conclusion

In this paper we approached Merkle-DAGs as causality-encoding structures with self-verification and efficient syncing properties. This led us to introduce the concept of *Merkle-Clock*, demonstrating that they can be described as a state-based CRDT which, announced with a *Broadcaster* component and fetched with a *DAG-Syncer* facility, converges in all replicas.

We then presented *Merkle-CRDTs* as Merkle-Clocks with CRDT payloads, a technique used in the past by multiple projects in the IPFS ecosystem. We showed how Merkle-CRDTs work with almost no messaging layer guarantees and no constraints on the replica-set, which can be dynamic and unknown, while providing per-object causal consistency.

As we saw, Merkle-CRDTs can carry any type of CRDT payload, but their properties make them specially interesting for operation-based and δ-CRDTs.

We finished by studying the limitations of Merkle-CRDTs and by proposing a number of optimizations over the original description, leaving DAG compaction and garbage collection strategies as areas for future work.

Merkle-CRDTs are a marriage between traditional blockchains, which need consensus to converge, and CRDTs, which converge by design, and thus inherit positive and negative aspects from both worlds. With this work, we hope to have set a good foundation for future research on the topic.

8 Acknowledgments

The authors are grateful to Adrian Lanzafame, Sander Pick, Carson Farmer, David Dias, Rohit Grover, David A. Roberts, Victor Grishchenko, Stephen Whitmore, Gonçalo Pestana, André Cruz and Alexei Baboulevitch whose comments, previous work and suggestions have inspired and improved much of the content in this paper.

Special thanks go as well to the Protocol Labs Research team and Jorge Soares in particular, for all the help getting this paper into its final form.

References

[1] José Bacelar Almeida, Paulo Sérgio Almeida, and Carlos Baquero Moreno. Bounded version vectors. In *International Conference on Distributed Computing - ICDCS*, volume 3274, pages 102–116, Tokyo, Japan, March 2004. Springer, Springer.

[2] Paulo Sérgio Almeida, Carlos Baquero, and Victor Fonte. Interval tree clocks. In *Proceedings of the 12th International Conference on Principles of Distributed Systems*, OPODIS '08, pages 259–274, Berlin, Heidelberg, 2008. Springer-Verlag.

[3] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Efficient state-based CRDTs by delta-mutation. *CoRR*, abs/1410.2803, 2014.

[4] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. Making operation-based CRDTs operation-based. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, PaPEC '14, pages 7:1–7:2, New York, NY, USA, 2014. ACM.

[5] Carlos Baquero and Nuno Preguiça. Why logical clocks are easy. 14, April 2016.

[6] Petr Baudis. Current concepts in version control systems. *CoRR*, abs/1405.3496, 2014.

[7] Juan Benet. IPFS - content addressed, versioned, P2P file system (draft 3), 2014.

[8] Eric A. Brewer. Towards robust distributed systems, 2000.

[9] Russell Brown, Sean Cribbs, Christopher Meiklejohn, and Sam Elliott. Riak dt map: A composable, convergent replicated dictionary. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, PaPEC '14, pages 1:1–1:1, New York, NY, USA, 2014. ACM.

[10] Russell Brown, Zeeshan Lakhani, and Paul Place. Big(ger) sets: decomposed delta CRDT sets in riak. *CoRR*, abs/1605.06424, 2016.

[11] Scott Chacon and Ben Straub. *Pro Git.* Berkely, CA, USA, 4th edition, 2018.

[12] Anton Churyumov. Byteball: A decentralized system for storage and transfer of value, 2016.

[13] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store, 2007.

[14] Carson Farmer and Sander Pick. Textile Threads whitepaper... just kidding... a deeper look at the tech behind textile's Threads protocol, October 2018.

[15] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference*, 10(1):56–66, 1988.

[16] Brent ByungHoon Kang, Robert Wilensky, and John Kubiatowicz. The hash history approach for reconciling mutual inconsistency. In *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ICDCS '03, pages 670–, Washington, DC, USA, 2003. IEEE Computer Society.

[17] Martin Kleppmann, Victor B. F. Gomes, Dominic P. Mulligan, and Alastair R. Beresford. Opsets: Sequential specifications for replicated datatypes (extended version). *CoRR*, abs/1805.04263, 2018.

[18] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.

[19] Tobias Landes. Tree clocks: An efficient and entirely dynamic logical time system. In *Proceedings of the 25th IASTED International Multi-Conference: Parallel and Distributed Computing and Networks*, PDCN'07, pages 375–380, Anaheim, CA, USA, 2007. ACTA Press.

[20] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology — CRYPTO '87*, pages 369–378, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.

[21] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2009.

[22] Geroge Neville-Neil. Time is an illusion. *ACM Queue*, 13(9), 2016.

[23] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Trans. Softw. Eng.*, 9(3):240–247, May 1983.

[24] Serguei Popov. The Tangle, 2016.

[25] Nuno M. Preguiça, Carlos Baquero, Paulo Sérgio Almeida, Victor Fonte, and Ricardo Gonçalves. Dotted version vectors: Logical clocks for optimistic replication. *CoRR*, abs/1011.5808, 2010.

[26] Nuno M. Preguiça, Carlos Baquero, and Marc Shapiro. Conflict-free replicated data types (CRDTs). *CoRR*, abs/1805.06358, 2018.

[27] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, Inria – Centre Paris-Rocquencourt ; INRIA, January 2011.

[28] Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, January 2009.