# Merkle-CRDTs (DRAFT)

Merkle-DAGs meet CRDTs

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

[1]Protocol Labs
[2]Haja Networks

February, 2019

**Abstract**

Merkle-DAG-backed CRDTs have been used to build some distributed applications on top of the Interplanetary File System (IPFS). In this paper we study Merkle-DAGs as transport and persistence layer for CRDT data types, coining the term *Merkle-CRDTs* and providing an overview to the different concepts, properties, advantages and disadvantages involved. We show how Merkle-CRDTs have potential to greatly simplify the design and implementation of convergent data types in systems with very weak messaging layer guarantees and a potentially very large number of replicas.

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

## 1 Introduction

The advent of blockchain technology has generalized the use of Peer-To-Peer (P2P) along with cryptographically linked acyclic graphs, known as Merkle-DAGs, to implement globally distributed and eventually consistent data structures such as crypto-currencies. In these systems, the Merkle-DAG is 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 anti-entropy algorithms (consensus) used in blockchains and *blockless* crypto-currencies are however too inefficient (or expensive) for high-performance data storage as offered by distributed databases.

*Conflict Free Replicated Data Types (CRDTs)*[1, 2] are an alternative mechanism to obtain eventual consistency. CRDTs rely on some properties of the data objects themselves that help eventually converging towards a

global, unique state without the need of consensus. CRDTs come in two main flavours: *state-based CRDTs*[1]–where replica states form a join semi-lattice and are merged under the guarantees afforded by it–, and *operation-based CRDTs*[2] –in which commutative operations are broadcasted and applied to the local state by every replica–.

The marriage between CRDTs and Merkle-DAGs has for long been a matter of discussion and study in the IPFS[3] Ecosystem[4]. IPFS provides a content-addressed peer-to-peer filesystem[3] 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[5], or OrbitDB [6], both powered by CRDTs on IPFS.

In this paper we gather a wide amount of information which until now has been spread across multiple Github repositories and online discussions and, for the first time, attempt to formalize what we name as *Merkle-CRDTs*. The goal is to provide an overview of their properties, advantages and disadvantages, so that it can set the ground layer for future research and optimizations in the space.

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 a *DAG-Sync* and a *Broadcaster* component, 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*, a Merke-DAG-based logical clock to represent causality information in a distributed system. Embedding causality information using Merkle-DAGs is at the core of crypto-currencies 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 a CRDT object, which can be synced, merged and for which we can formally prove eventual consistency across different replicas.

Building on the previous sections, in Section 5. we define *Merkle-CRDTs* as a general purpose transport and persistency layer for CRDT payloads which leverages the properties of Merkle-Clocks, the DAG-Sync and the

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

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

[3]The Interplanetary File System: `https://ipfs.io`

[4]One of the earliest references in 2015: `https://github.com/ipfs/notes/issues/40`

[5]PeerPad: realtime P2P collaborative editing: `https://peerpad.net`

[6]OrbitDB: Peer-to-Peer Databases for the Decentralized Web: `https://github.com/orbitdb/orbit-db`

Broadcaster components to provide per-object causal consistency out of the box. This enables the use of simple CRDT types in systems with very weak messaging layer guarantees and very large number of replicas. We further discuss how different CRDT payloads (operation-based, state-based and $\delta$-based) benefit from Merkle-CRDTs. Finally, we describe some of the limitations and inefficiencies of Merkle-CRDTs and introduce some techniques to overcome them.

# 2 Background

## 2.1 Eventual consistency

The CAP Theorem[4] establishes that, in a distributed system, it is impossible to have 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 during a network partition which keeps them from contacting each others, there is no way that they can synchronize to a consistent state. If they, however, stop accepting writes to prevent this situation, they cannot be considered to be available any more, but they can maintain consistency. Similarly, replicas in a system cannot stay consistent and available in a system where partitions are tolerated.

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)[7][5], meaning that, at a given moment, the state may not the same across replicas –in fact it may be completely different–, but that 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 things will converge or how much the shared states will be allowed to diverge until then[8]. *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.

[7] Also known as *optimistic replication*.

[8] EC only provides a *liveness* guarantee: the system will not get stuck when it comes to converging.

## 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 $A \rightarrow B \rightarrow C$ is an instance of a DAG where $B$ is a descendant of $A$ and so on. In general, we would say that $B$ *is a child or descendant of* $A$, and that *node* $A$ *has a link to* $B$. Conversely $A$ *is a parent of* $B$.

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:

a) Merkle-DAGs can only be constructed from the leaves, that is, from nodes without children. Thus, parents are added after children because their identifiers must have been computed in advance.

b) every node in a Merkle-DAG is the root of a Merkle-[sub]DAG itself, and that this subDAG is *contained* in the parent DAG[9].

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 their hash is referred to as *content addressing*. Thus, we name the node identifier as *Content Identifiers* or CIDs.

For example, 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 graphs[10].

Merkle-DAGs are a *self-verified structure*. The CID of a node is univocally linked to the contents of its payload and those of all of 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.

Merkle-DAGs are very widely used. Source control systems like Git[6] and others[7] use them to efficiently store the repository history, content-addressing the objects and being able detect conflicts between branches.

In distributed databases like Dynamo[8], Merkle-Trees are used for efficient comparison and reconciliation of the state between replicas. In Hash

[9] Merkle-DAGs are similar to Merkle Trees (`https://en.wikipedia.org/wiki/Merkle_tree`) but there are no balance requirements and every node can carry a payload. In DAGs, several branches can re-converge or, otherwise said, a node can have several parents.

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

Histories[9], content-addressing is used to refer to a Merkle-Tree representing a state[11].

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 application: crypto-currencies. Crypto-currencies like Bitcoin[10] benefit from the embedded causality information encoded in the chain: a block deeper in the chain always happened before its parent. Their main problem is to make all participating peers agree about the tip/head of this chain. Among other things, the non-commutative nature of some transactions, like those originating from the same wallet, requires a consensus mechanism which enforces that only valid blocks become the new roots.

There are also DAG-based crypto-currencies[12] like *DAG*, *Byteball* [13] or *IOTA*[14]. 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[15]. 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 casually convergent systems involves the reconciliation of diverging state versions among different replicas when, for example, events occur(red) concurrently. This requires that we are able to identify whether

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

[12]Also called *Blockless crypto-currencies*

[13]Byteball: A cryptocurrency platform ready for real world adoption: `https://byteball.org/`. *Byteball*'s DAG[11] 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 *Byteballs* specifications.

[14]IOTA: A permissionless distributed ledger for a new economy: `https://iota.org`. In IOTA's *Tangle*[12], 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.

[15]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.

two events actually happened concurrently and whether two states are actually different because of concurrent updates or other reasons, like simply having received more updates in one of the replicas.

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

Ideally, we should be able to order all the events in the system[16] 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 stablish in which order 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[13], 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. An alternative is to use *logical clocks*.

*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, 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 from itself come after those. This is, in essence, what is known as *causal histories*.

Logical clocks are representations of causal histories[14] 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* $b$ ($a \rightarrow b$), or vice-versa ($b \rightarrow a$), or if both $a$ and $b$ happened concurrently ($a \parallel b$)[17].

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*[16]: every replica maintains and broadcasts a vector that tracks on which version the state of all replicas is, as far as it can tell. When a replica performs a modification of the

[16] This would mean having a *total strict order* for all the events.

[17] We take a number of shorcuts in this description. Logical clocks were originally described by Lamport[15] as a function which, for every event, returns a value so that:

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

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

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

state, it increases its version. When a replica merges a state from a different replica, it takes the highest version between the local versions and the versions provided by the other replica along with the event. Thus, given two events $a$, $b$, with two version vectors $\mathcal{V}^a$, $\mathcal{V}^b$, $a \rightarrow b$ if for each position i in the vectors $\mathcal{V}_i^a \leq \mathcal{V}_i^b$ (and vice-versa). If $a \not\rightarrow b$ and $b \not\rightarrow a$, 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 a single number indicating how long the history is for every replica. Version vectors depend on the number of replicas, so they may also need further optimizations to work well in some scenarios with many replicas or systems where the number of replicas is not stable.

Along with 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[17], bounded version vectors[18], dotted version vectors[19], tree clocks[20] or interval tree clocks[21].

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 some properties from the state and/or the operations applied to modify it. Additionally, CRDTs also feature monotonicity[18], meaning that roll-backs on the state are not necessary regardless of the order in which updates happen.

There are two prominent types of CRDTs: *state-based*[19] and operation-based CRDTs[20]. 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"*[21] ($\sqcup$) and the result is a new state corresponding to the Least-Upper-Bound of both (LUB)[1]. 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

[18]The notion 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. A good example is that a CRDT counter which can be increased and decreased –PN Counter–, is necessarily implemented using two counters which can only be increased.

[19]Also known as CvRDTs (*Convergent*)

[20]Also known as CmRDT (*Commutative*)

[21]Also referrered *"union"*, or *"merge"*.

not more (the LUB). A join semilattice is thus, a partially ordered set[22] and its Least-Upper-Bound 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. Upon receiving the state, the other replicas *merge* it with the local state. 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[1], on the other side, do not enforce any property on the state, but instead consist of operations that modify the state, requiring that they are commutative[23]. In operation-based CRDTs the replicas broadcast the operations. Because operations are commutative, if two operations happen at the same time in two replicas and then broadcasted, it does not matter the order in which replicas apply them: the resulting states in both replicas will be the same.

It follows that, if an operation does not arrive to its destinations –for example due to a network failure–, the replicas will never be able to apply it and their states will no converge. Thus, unlike state-based CRDTs, eventual consistency in operation-based CRDTs requires a reliable messaging layer that eventually delivers all operations[22]. Additional constraints may be necessary, for example, if operations are not idempotent: in that case the messaging layer should ensure that each operation is only delivered 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 casual 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 $A$ is not part of the set but $B$ is[24].

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 improved in different applications and distributed databases, Basho's Riak[23, 24] being one of the most

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

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

[24] This is clear for an operation-based implementation of a G-Set, where there is causal delivery of operations. The state-based implementation of a G-Set consists in sending the full set. Thus, the event adding $B$ is a set which already contains $A$. Thus 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.

prominent examples[25].

# 3 System model

Our Merkle-CRDT approach is oriented to be both simple and facilitate the use of CRDTs in P2P distributed systems very large peer sets (replicas) and no pre-conditions regarding the messaging layer.

We just 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: a *DAG-sync* component and a *Broadcaster* component (defined below).

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 replicas. There can be partitions due to network failures, but they are resolved as soon as connectivity is re-established and a replica broadcasts a new event.

Replicas may have durable storage attached or not, depending on their own requirements and data types. New replicas, or crashed replicas without durable storage can eventually re-construct the state of the system as long as there is at at least one other replica which has the complete state or full history of operations.

## 3.1 The DAG-Sync component

A *DAG-Sync* is a component which enables a replica to obtain remote Merkle-DAG nodes from other replicas given their CIDs[26] and to provide its own nodes to the rest. Since a node contains links to their direct descendants, given the root node's CID, the DAG-Sync component can be used to fetch the root node and their children. Following the links to children in each node, we can traverse the full graph and retrieve all nodes in the DAG. Thus, we can define the DAG-Sync as follows:

**Definition 1.** (DAG-Sync). A DAG-Sync 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-Sync layer should not impose any additional constraints on the system model. Our approach relies on the properties of the DAG-Sync and Merkle-DAGs to tolerate all the network contingencies described above.

[25] Check `https://github.com/ipfs/research-CRDT/issues/40` for a few examples.

[26] *Content-Identifiers*. See *Merkle-DAGs* in the background secion.

### 3.2 The Broadcaster component

A *Broadcaster* component sends arbitrary data from one replica to all the rest. Ideally, the payload will reach every replica in the system, but this is not a must for every broadcasted message:

**Definition 2.** (Broadcaster). A Broadcaster is a component with one method: `Broadcast(Data)`.

### 3.3 IPFS as a DAG-Sync and Broadcaster component

The problem of implementing the components above is addressed by *Inter-Planetary File System (IPFS)*[3]. IPFS provides a content-addressed filesystem which can act as an optimized DAG-Sync component. IPFS uses a Distributed Hash Table (DHT) to announce and discover which replicas (or peers) provide certain Merkle-DAG nodes. It implemets a DAG-fetching protocol called "bitswap" which can recursively retrieve full DAGs from multiple *providers*.

For the Broadcaster part, IPFS includes, brought by its P2P layer – libp2p[27]–, broadcasting mechamisms based on *Publisher Subscriber (Pub-sub)* models[28]. These allow to efficiently broadcast information to all replicas in our system without the need to implement an ad-hoc broadcasting mechanism.

IPFS also integrates IPLD, the *Inter-Planetary Linked Data Format*[29], a framework to describe Merkle-DAG with arbitrary node formats and support for multiple types of CIDs[30], making it very easy to create and sync custom DAG nodes, as needed to implement the CRDTs mechanisms described in this paper.

## 4 Merkle-Clocks

### 4.1 Overview

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

We build this DAG merging other DAGs (those in other replicas) according to some simple rules. New events are added as root nodes (or parents) to the existing ones[31].

[27] libp2p: A modular network stack. `https://libp2p.io`

[28] Floodsub and gossipsub `https://github.com/libp2p/go-libp2p-pubsub`

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

[30] Multiformats: self-describing values for Future-proofing. `https://multiformats.io/`

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

For example, given $\mathscr{M}_\alpha$ and $\mathscr{M}_\beta$ ($\alpha$ and $\beta$ being the single root CIDs in those DAGs[32]):

1. If $\alpha = \beta$ no action is needed, as they are the same tree.
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 that $\mathscr{M}_\beta < \mathscr{M}_\alpha$ in this case.
4. else, we *merge* both Clocks simply keeping both DAGs as they are and thus having two root nodes. We say that $\mathscr{M}_\alpha = \mathscr{M}_\beta$ in this case. Note that $\mathscr{M}_\alpha$ and $\mathscr{M}_\beta$ could be or not, if they share some of their deeper nodes. If we wish to record a new event, we can do so by creating a new root $\gamma$ with two children, $\alpha$ and $\beta$. .

We can already see that, by looking if a Merkle-Clock is included in another, we have a 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 before will always be descendants of events that happened later. Additionally, we have provided a way to merge Merkle-Clocks according to how they compare. The resulting Clock always includes the causality information from both Clocks. What this means is that the causality information stored in Merkle-Clocks in every replica will converge to the same Merkle-Clock.

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 however important that we formalize how we define order between Merkle-Clocks and that we prove that the causality information is maintained when they are synced and merged. This will be an underlying property in Merkle-CRDTs.

### 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 represented as a G-Set CRDT and therefore are able to converge in multiple replicas[33].

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

[32]For simplicity reasons in the example, we assume that we start working with DAGs containing a single root, but it is generalizable.

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

**Definition 3.** (Merkle-Clock Node). A Merkle-Clock Node $n$ is a triple:

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

which represents an event $e_\alpha \in \mathscr{S}$, with $\alpha$ being the node CID and $\mathcal{C}_\alpha$ being the set of CIDs which reference direct descendants of $n$, or *links of* $\alpha$.

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

$$\langle \mathbb{N}, \leq_{\mathscr{M}} \rangle$$

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

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

Otherwise said $n_\alpha < n_\beta$ if there is a path of linked nodes which can take from $n_\alpha$ to $n_\beta$.

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 with a new Merkle-Clock DAG root to the existing Merkle-Clock DAG(s). The new root must reference any previous roots or nodes without ascendants.

**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 $\mathbb{N}$:

$$\mathscr{M} = \mathscr{S} \hookrightarrow \mathbb{N}$$

*Remark.* A Merkle-Clock satisfies the *Strong Clock condition*[15]. We see that every node represents a later event than that of its children:

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

Since every event is the root of a [sub]DAG built with the rules above, we can immediately see that earlier Merkle-Clock values are descendants of the later ones:

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

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 $\mathbb{M}, \mathbb{N} \in \mathbb{J}$:

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

Note that $m <_{\mathscr{M}} n$, 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 simply 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[2]. 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 between them. Causality information about DAG-merge events can be included after the union the DAGs by creating a new unique root, as the *implementation rule* states.

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

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

We have defined so far a way to encode causality information per replica and we have ensured that two replicas can merge their Merkle-Clocks. Merkle-DAGs are, however, not a must for this task: a regular DAG structure could serve for the same purpose. The problem with a non-Merkle DAG is that the system would either have to ensure that messages are received, or the implementers provide protocols to sync the Clocks when the messaging layer is unreliable.

The properties of Merkle-DAGs allow, however, to have a *fetch* rather than a *push* approach, which, together with content-addressing, allows to efficiently sync clocks between replicas:

- Broadcasting the Merkle-Clock requires to broadcast only the current root CID. The whole Clock is unambiguously identified by the CID of its root and all its nodes can be walked down from it as needed.
- The Merkle-DAG allows every replica, given a new root, to fetch the associated Merkle-DAG nodes that it does not have already, and if the DAG-Sync component allows, to fetch them from any provider in the system and not from a particular replica.
- Nodes are de-duplicated. That means that there can only be one unique representation for every event which facilitates optimized fetching and processing of the DAG.

In practice, every replica just fetches the *delta* causal histories from other replicas without the need to build those deltas anywhere in the system. Similarly, if a replica is completely new and has no previous history, it will fetch the full history automatically[34].

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

- Using Merkle-Clocks decouples the causality information from the number of replicas, which is a common limitation in version clocks. This allows to reduce the size of the messages when implementing CRDTs and, most interestingly, resolves the problem of keeping clocks working with replicas randomly joining and leaving the system.
- On the downside, this translates into making the causal information grow with every event and having all 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 new comers. However, that syncing may be too slow if the history is very large. We will explore, along with Merkle-CRDTs, potential optimizations in this regard.

Merkle-Clocks can also deal with network eventualities without much trouble:

- Dropped messages may involve the inability to fetch parts of a DAG, or to inform 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, replica downtimes etc. do not have an effect on the overall system and heal automatically when the issues are resolved.
- 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, since their Merkle-Clock already incorporated them if they were processed before.
- 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 from the DAG-Sync layer, therefore they are ignored. And b) if a DAG node is corrupted

[34]This is precisely how peers participating in crypto-currencies sync their ledgers.

on download, the DAG-Sync component (or the application) can easily verify that its CID corresponds to the downloaded contents, and discard it if not.

# 5 Merkle-CRDTs: Merkle-Clocks with payload

**Definition 6.** (Merkle-CRDT). A Merkle-CRDT is a Merkle-Clock where 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 here is that the Merkle-Clock already provides ordering and causality information which otherwise needs to either travel attached as version vectors or be provided by a reliable messaging layer.

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

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

with $\alpha$ being the *content identifier*, $P$ an opaque data object with a CRDT properties and $\mathcal{C}$ the set of children identifiers[35].

## 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 order of events, provides all the necessary properties to ensure per-object *causal consistency* and *gap detection* out of the box and without modifying our system model in any way.

This means that Merkle-CRDTs are very well suited to carry operation-based CRDTs and ensure that no operation is lost or applied in disorder: the total order associated to the Merkle-Clock can be used to sort events (or payloads) in the order they where issued[36].

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

[35] 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.

[36] To re-iterate, this is a non-strict total order ($\leq$), where concurrent events are considered equal.

### 5.2 General anti-entropy algorithm for Merkle-CRDTs

The following algorithm has been used in very similar forms in many of the projects implementing Merkle-CRDTs mentioned before. We generalize it here in terms that are simple to understand.

**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 new the root to its Merkle-CRDT, which becomes $\mathscr{M}_\beta$
2. $\mathcal{R}^B$ broadcasts $\beta$ to the rest of replicas in the system.
3. $\mathcal{R}^A$ receives the broadcast of $\beta$ and retrieves the full $\mathscr{M}_\beta$, starting by the root $\beta$ and walking down the DAG using the DAG-Sync component to fetch all the nodes that are not in $\mathscr{M}_\alpha$ and collecting their CIDs in a CID-Set $\mathcal{D}$. For any CID already in $\mathscr{M}_\alpha$ the whole subDAG 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 Merkle-Clock non-strict order $\leq_{\mathscr{M}}$ as defined before. 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 concurrence in the system and how long two Merkle-CRDTs have been allowed to diverge, but in normal conditions it should be low.
6. $\mathcal{R}^A$ processes the payloads associated to the nodes from $\mathcal{D}$ 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, $\mathscr{M}_\alpha \not\subset \mathscr{M}_\beta$ and $\mathscr{M}_\beta \not\subset \mathscr{M}_\alpha$. $\mathcal{R}^A$ keeps both nodes as roots.
9. When a new event happens, we proceed again from the first step.

### 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[22] or complex workarounds, like additional causality payloads, buffering and re-try 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 parts of the DAG that it is missing without any extra information than the root of the DAG. 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 defer efficient broadcasting (via PubSub mechanisms) to the *Broadcaster* component.

We should note that adding Lamport timestamps to each operation means they can be used to implement different replicated data types as proposed by the *OpSets specifications*[25][37].

### 5.4 State-based Merkle-CRDTs

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

A not so *natural* use of Merkle-CRDTs is to embed state-based CRDTs in them. This is because state-based CRDTs already provide per-object causal consistency and can cope with unreliable message layers by design.

Moreover, although the final state would be resulting from the merge of all the states in the Merkle-CRDT nodes, the *DAG-Sync* component would still need to store those, something intuitively prohibitive when working with large state objects.

On the plus side, Merkle-CRDTs remove the need to attach causality metadata and detach it from the number of replicas, which might be of interest for state-CRDTs with very small states in comparison to the number of replicas.

An interesting twist here is provided by $\delta$-CRDTs[26] which, instead of broadcasting full states, are able to broadcast the smaller sections of them (deltas). $\delta$-mutations, as these objects are called, can be merged downstream

[37] 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 quite benefit from a Merkle-CRDT transport which ensures causal delivery, potentially unlocking some optimizations.

just like any full state would be, without the need of changing the semantics of the *union* operation. It follows then that multiple deltas can be merged to form what is known as $\delta$-groups and increase the efficiency of broadcasted payloads. As pointed out in their paper, *"a full state can be seen as a special (extreme) of a delta-group"*.

Among the issues with $\delta$-CRDTs is that, in their vanilla form, consistency is delayed ad-infinitum when a message is lost. They lose the per-object casual consistency property of state-based CRDTs. These issues can be addressed with an additional anti-entropy algorithm that groups, sorts, tracks delivery and re-sends missing deltas, as presented in the paper.

In the case of $\delta$-mutation Merkle-CRDTs, said algorithm and any causal information attached to the original objects would not be necessary. In essence, this approach brings $\delta$-state Merkle-CRDTs quite close to their operation-based counterpart.

### 5.5 Downsides in Merkle-CRDTs

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

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

- The size of the DAG might grow too big to what is 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 enormous sizes in time[38]. This is specially problematic when the actual state might be much smaller. In some cases, it might be possible to express the state as the result of all the Merkle-CRDT operation (or their union), but that brings us to the next point.
- If only the Merkle-DAG is stored, knowing that the full state can be rebuilt from it (and thus saving that space), start up of replicas with very large Merkle-DAGs might be specially slow since they will need to reprocess a potentially very large DAG, even when available locally. If not, there will be redundant information in the state and the Merkle-DAG.
- Merkle-CRDT syncs from scratch are possible and natural to the system when a new replica partakes in it. However, Merkle-DAGs are

[38] Bitcoin at 185GB and Ethereum well over 1TB as of this writing.

not only ever-growing, but also will tend to be very deep[39]. 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 then take significantly more time 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 and transformation 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 differed a lot (or long ago).

**DAG-Sync latency:** Replicas rely on a DAG-Sync 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)[40]. It allows, on one side, to collaborative store and locate small pieces of information (*discovery*) and, on the other, to be able to discover peers and route other peers to them (*routing*). DHTs are massively scalable but introduce some overhead[41] which will potentially make fetching DAG-nodes a slower operation than it would have been to receive them directly from the issuer.

The problems above are more or less relevant depending on the requirements of the system being built. 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)

[39]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.

[40]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`.

[41]This is specially relevant when using an IPFS node connected to the global IPFS network, where the DHT will not only store the data associated to the Merkle-CRDT or be only participated by the replicas. It is however perfectly possible to use private IPFS networks for the task.

- Expected amount of event concurrency

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 lists some of the issues we have to account for when using Merkle-CRDTs, specially in their vanilla, non-optimized fashion in which we have presented them. We will describe now potential optimizations that can be used to address some of those problems.

**Optimized nodes:** Similar to delayed merge-nodes, we can, in scenarios where replicas issue frequent updates, wait for multiple payloads before issuing a single node containing all of them. The benefits are clear and the downside is that updates are not immediately sent out and will take longer to arrive.

We should also attempt to reduce the size of the payloads as much as possible by compressing and removing any redundant information not required by the CRDT itself. For example, instead of signing the CRDT payloads to ensure that they become from a trusted replica (when sharing a wider network), we can sign the broadcast messages, thus leaving signatures out of the Merkle-DAG.

**Quick Merkle-DAG inclusion check:** Merging the local replica DAGs with a remote one requires checking if one DAG includes the other, and vice-versa. It is possible to do this by walking down the one DAG looking for a node CID that matches the root of the second DAG, but it is very inefficient. Storing the CIDs of the local DAG in a map that can quickly tell if a CID is part of the local DAG or not makes things significantly easier. 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 before fetching them, and those branches conveniently pruned.

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 any 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. In cases when the broadcast takes long to reach all the replicas, this also

allows them to fetch the new nodes from different replicas that already received them.

However in some systems it may be beneficial[42] 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 on that regard. Broadcasting the payloads (assuming they are small enough) will like reduce the latency of the propagation of changes in the system.

**Content-addressing the payloads:** It is possible to reduce the total size of the Merkle-CRDT DAG by including only a CID as payload, which can then be fetched separately. This may increase the efficiency of the DAG fetching, specially if sometimes the payloads carried are identical.

The above recommendations should be considered in any Merkle-CRDT implementation, as they may provide significant advantages over the unoptimized version describe previously. We leave topics of DAG compaction and garbage collection out for future research, although we can intuitively see that discarding parts of the Merkle-DAG is not possible without knowing if every replica is aware of them and this, in turn, requires knowing the replica-set[43], a system constraint that we did not have before.

# 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. In the Javascript IPFS[44] ecosystem several projects have already applied them[45], all embedding operation-based CRDTs in Merkle-DAGs –the most natural approach as we will see:

- `ipfs-hyperlog`[46] is a Merkle DAG build and replication utility.
- `ipfs-log`[47] 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 grow-only G-Set).

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

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

[44] `https://js.ipfs.io/`

[45] Many of them under the scope of the dynamic data and capabilities working group: `https://github.com/ipfs/dynamic-data-and-capabilities`.

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

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

- `Orbit DB`[48] is a distributed, peer-to-peer database. It uses `ipfs-log` and other CRDTs for different data models. It is used to build `Orbit`[49], a distributed, serverless chat application.
- `Tevere`[50] is an operation based Merkle CRDT key-value store.
- `peer-crdt`[51] and `peer-crdt-ipfs`[52] provide a generalistic operation Merkle-CRDT implementations of several CRDTs: counters, sets, arrays, registers and text (as well as composable CRDTs).
- `versidag`[53] is a proposed linked log with conflict resolution to store version information, similar to `ipfs-log`.
- `PeerPad`[54] is a real-time collaborative text editor based on `peer-crdt` and $\delta$-CRDTs.

In general, Merkle-CRDT implementations in the IPFS ecosystem focused on real-time browser applications like Orbit Chat[55] and PeerPad[56]. These scenarios are characterized by:

- Small-sized operations (i.e. a single letter typed by a user).
- Potentially long or very long histories with not many concurrent operations (narrow DAGs).
- Relatively small number of replicas (i.e. the number of users in a chat room).
- Opportunistic joining and departures from the replica set (i.e. a user closing the browser tab).
- Full operability under network partitions.
- Very low latency requirements.

In the following sections, we will see that Merkle-CRDTs support part of these requirements but pose certain difficulties with others.

[48] `https://github.com/orbitdb/orbit-db`
[49] https://github.com/orbitdb/orbit
[50] `https://github.com/ipfs-shipyard/tevere`
[51] `https://github.com/ipfs-shipyard/peer-crdt`
[52] `https://github.com/ipfs-shipyard/peer-crdt-ipfs`
[53] `https://github.com/ipfs/dynamic-data-and-capabilities/issues/50`
[54] `https://github.com/ipfs-shipyard/peer-pad`
[55] `https://orbit.chat`
[56] `https://peerpad.net`

# 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 efficiently fetched with a *DAG-Sync* facility, converge 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 provide per-object causal consistency with no almost no messaging layer guarantees and no constraints on the replica-set, which can be dynamic and unknown. As we saw, Merkle-CRDTs can carry any type of CRDT payload, but their properties make them specially interesting for operation-based and $\delta$-CRDTs.

We finished by studying the different downsides of Merkle-CRDTs and proposing a number of optimizations over the original description. We pointed out DAG compaction and garbage collection strategies as areas for future research.

Merkle-CRDTs are a marriage between traditional blockchains, which need consensus to converge, and CRDTs, which converge by design, and thus get good and bad things from both worlds. With this work, we hope to have set a good foundation to both learn about the topic and iterate on it on future publications.

# 8 Acknowledgments

TODO.

Among github users which made interesting comments or tools or contributions:

@davidar @haadcode @gritzco @noffle @gpestana @satazor @daviddias @archagon +probably a few more

# References

[1] N. M. Preguiça, C. Baquero, and M. Shapiro, "Conflict-free replicated data types (crdts)," *CoRR*, vol. abs/1805.06358, 2018. [Online]. Available: http://arxiv.org/abs/1805.06358

[2] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "A comprehensive study of convergent and commutative replicated data types," 2011. [Online]. Available: https://ipfs.io/Qmcuf84CJrjpbjK71SiBtwetxUNCEKArfWT5BYfHb2TBGE

[3] J. Benet, "Ipfs - content addressed, versioned, p2p file system (draft 3)," 2014. [Online]. Available: https://ipfs.io/QmV9tSDx9UiPeWExXEeH6aoDvmihvx6jD5eLb4jbTaKGps

[4] E. A. Brewer, "Towards robust distributed systems," 2000. [Online]. Available: https://ipfs.io/QmQaNoXRXz2PHip9ABjhWnii3WYKxmX4U6uxf7STpRXQSV

[5] W. Vogels, "Eventually consistent," *Commun. ACM*, vol. 52, no. 1, pp. 40–44, Jan. 2009. [Online]. Available: https://ipfs.io/QmZ1kXcgiMLtNGADjn9T1mJjB77Q4hw66SkWcqgnBotMNP

[6] S. Chacon and B. Straub, *Pro Git*, 4th ed., Berkely, CA, USA, 2018. [Online]. Available: https://ipfs.io/QmaX8iXr5GeHBVY1AQoPeV32Vh7NsU8paAAVzZzDVCL38h

[7] P. Baudis, "Current concepts in version control systems," *CoRR*, vol. abs/1405.3496, 2014. [Online]. Available: https://ipfs.io/QmRj9raoi5Ga35bBkz3irRnKpitASdVS876P7zUt2hk2Ss

[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," 2007. [Online]. Available: https://ipfs.io/QmVvPVah8EoNJeUzibtq63mnHxKHfF3kg4yVoMEEQztFkM

[9] B. B. Kang, R. Wilensky, and J. Kubiatowicz, "The hash history approach for reconciling mutual inconsistency," in *Proceedings of the 23rd International Conference on Distributed Computing Systems*, ser. ICDCS '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 670–. [Online]. Available: http://dl.acm.org/citation.cfm?id=850929.851951

[10] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," Bitcoin.org, 2009. [Online]. Available: https://ipfs.io/QmRA3NWM82ZGynMbYzAgYTSXCVM14Wx1RZ8fKP42G6

[11] A. Churyumov, "Byteball: A decentralized system for storage and transfer of value," 2016. [Online]. Available: https://ipfs.io/QmXVrvD7jt767Cq4K5gQrgjGrHnnBJAufCskDdnzfMX5FS

[12] S. Popov, "The tangle," 2016. [Online]. Available: https://ipfs.io/QmUjGuHFjBJQTGub8yznuoBacU49QVJnJdvKMm6D1a9J7C

[13] G. Neville-Neil, "Time is an illusion," *ACM Queue*, vol. 13, no. 9, 2016. [Online]. Available: https://ipfs.io/QmWCC9Y6hT3SgvEnDENF6657qp1rb2Y8gc22aZXpHgQuZw

[14] C. Baquero and N. Preguiça, "Why logical clocks are easy," vol. 14, 4 2016. [Online]. Available: https://ipfs.io/QmZdmmwq2qXq4hj3ZU2fznkLMXY52aB3ogPp4VNuqeW8Vr

[15] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978. [Online]. Available: https://ipfs.io/QmYyTys6WKRtudeLQVULgSY61Lyo7mXtod67u4DtkTiots

[16] 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.*, vol. 9, no. 3, pp. 240–247, May 1983. [Online]. Available: https://ipfs.io/ipfs/QmW2bWrdDDbZ8nYZpp9tHYP8dFoRTHpcnRkv43hBFYTm4k

[17] C. J. Fidge, "Timestamps in message-passing systems that preserve the partial ordering," *Proceedings of the 11th Australian Computer Science Conference*, vol. 10, no. 1, p. 56–66, 1988. [Online]. Available: https://ipfs.io/ipfs/Qmb1HzUdv1Wwo9yizsJuAz6Hv97svcihHRQFybtSfSvvky

[18] J. B. Almeida, P. S. Almeida, and C. B. Moreno, "Bounded version vectors," in *International Conference on Distributed Computing - ICDCS*, vol. 3274, Springer. Tokyo, Japan: Springer, March 2004, pp. 102–116. [Online]. Available: https://ipfs.io/QmRvkzLddazVRNFR3YJpgGrpwBRerWsuAc6rQACqq6Limf

[19] N. M. Preguiça, C. Baquero, P. S. Almeida, V. Fonte, and R. Gonçalves, "Dotted version vectors: Logical clocks for optimistic replication," *CoRR*, vol. abs/1011.5808, 2010. [Online]. Available: https://ipfs.io/QmeoUxMvGCWQq6JdySDsKscA3BhnyoHHoTQu6WpUbWm6Mj

[20] T. Landes, "Tree clocks: An efficient and entirely dynamic logical time system," in *Proceedings of the 25th Conference on Proceedings of the 25th IASTED International Multi-Conference: Parallel and Distributed Computing and Networks*, ser. PDCN'07. Anaheim, CA, USA: ACTA Press, 2007, pp. 375–380. [Online]. Available: https://ipfs.io/ipfs/QmRiduykvjMEKNNF3V4T7G7RLCPRGvN4PBb3Lid5c14K1V

[21] P. S. Almeida, C. Baquero, and V. Fonte, "Interval tree clocks," in *Proceedings of the 12th International Conference on Principles of Distributed Systems*, ser. OPODIS '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 259–274. [Online]. Available: https://ipfs.io/ipfs/Qmad9MpEitiU45RyxHsZxzpb1aRCBPFzoX5ecHJa5LFoZz

[22] C. Baquero, P. S. Almeida, and A. Shoker, "Making operation-based crdts operation-based," in *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, ser. PaPEC '14. New York, NY, USA: ACM, 2014, pp. 7:1–7:2. [Online]. Available: https://ipfs.io/QmdG7zAeFH1Yf84bFwkAm3hJ9WKypUXaauXMHQUdVQxAeL

[23] R. Brown, S. Cribbs, C. Meiklejohn, and S. Elliott, "Riak dt map: A composable, convergent replicated dictionary," in *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, ser. PaPEC '14. New York, NY, USA: ACM, 2014, pp. 1:1–1:1. [Online]. Available: http://doi.acm.org/10.1145/2596631.2596633

[24] R. Brown, Z. Lakhani, and P. Place, "Big(ger) sets: decomposed delta CRDT sets in riak," *CoRR*, vol. abs/1605.06424, 2016. [Online]. Available: https://ipfs.io/QmT2m8TXMEMj8KrxbrHGawfzZFJJsuNzUcPeuVoitqpjtq

[25] M. Kleppmann, V. B. F. Gomes, D. P. Mulligan, and A. R. Beresford, "Opsets: Sequential specifications for replicated datatypes (extended version)," *CoRR*, vol. abs/1805.04263, 2018. [Online]. Available: https://ipfs.io/ipfs/QmdQnHZfZ97jx5n2GsnBLiRVbxzyzVfH9rTAoySE7WN3cD

[26] P. S. Almeida, A. Shoker, and C. Baquero, "Efficient state-based crdts by delta-mutation," *CoRR*, vol. abs/1410.2803, 2014. [Online]. Available: http://arxiv.org/abs/1410.2803