ERA: Epoch-Resolved Arbitration for Duelling Admins in Group Management CRDTs

Kegan Dougal
kegan@element.io
Element Creations Ltd
London, United Kingdom

Abstract

Conflict-Free Replicated Data Types (CRDTs) are used in a range of fields for their coordination-free replication with strong eventual consistency. By prioritising availability over consistency under partition, nodes accumulate events in different orders, and rely on an associative, commutative and idempotent merge function to present a materialised view of the CRDT. Under some circumstances, the state of the materialised view over time can appear to "roll back" previously applied events. When the materialised view is used to manage group permissions such as ones found in instant messaging applications, this can lead to surprising behaviour. This can occur when there are multiple concurrent events, such as in the Duelling Admins problem where two equally permissioned admins concurrently revoke each other's permissions. Who wins? This article argues that a Byzantine admin can exploit concurrency to win the duel. As a result, an external arbiter is required to arbitrate an immutable happens-before relation between concurrent events. Arbitration occurs asynchronously in batches via optional "epoch events", preserving availability. This introduces a bounded total order within epochs, and the resulting "finality" improves on the level of consistency CRDTs can provide.

CCS Concepts: • **Theory of computation → Distributed algorithms**; • **Information systems** → *Data structures*; • **Security and privacy → Distributed systems security**; • **Software and its engineering → Access protection**; • **Computer systems organization** → *Availability*.

Keywords: Byzantine fault tolerance, CRDTs, Finality, Causal Stability

1 Introduction

Conflict-Free Replicated Data Types (CRDTs) [25] are in widespread use in a variety of fields including databases [2, 3], consumer messaging [4] and collaborative apps [26]. Many products rely on a separate system for authorising access to the CRDT e.g. OIDC / LDAP servers. This can subvert the incentives [22] for using a CRDT in the first place as it reduces availability under partition should the authorisation system be unavailable. Keyhive [26] and Matrix [4] attempt to implement authorisation based on Byzantine fault tolerant (BFT) CRDTs using hash chronicles [17] (hash DAGs). Whilst they implement decentralised access control differently [16] [9], both systems present to the application-level the ability to manage a group of users. Users may be in different membership states such as *invited*, *joined*, or *left*. Users may also have different roles such as *Administrator* which enables the ability to remove other users and change their roles. In these systems, the group creator starts with the *Administrator* role and they can subsequently promote or demote other users to this role. Demotion (or revocation) is problematic in these systems because it is a non-monotonic operation [18]. This means some previously authorised events may need to be *rolled back* due to concurrent revocations. This introduces the "Duelling Admins" problem: what should be the result of processing two concurrent revocations where Admin A revokes Admin B whilst concurrently Admin B revokes Admin A? This forms a revocation cycle which must be automatically resolved by the CRDT merge function.

Kleppmann explained *seniority ranking* at PaPoC 2025's keynote [19] which provides a strategy to solve this problem: execute the more senior admin operations first e.g. A < B. However, this enables retaliation where a revoked Admin A can retaliate against Admin B by *backdating* a revocation to make it appear as if the two admins concurrently revoked each other. This allows Admin A to *roll back* their revocation by abusing their seniority.

In this article, we describe the Duelling Admins problem, explain how Keyhive and Matrix solve it, and explore how it can be solved in a fair way by the introduction of a mutually trusted third party node called a *finality node*. This node is responsible for arbitrating the order of events via the transmission of epoch events. We conclude with an outline of how CRDT designs may use the arbitration characteristics of finality nodes in novel ways.

2 Background

CRDTs are updated by exchanging *events* between all participating *nodes*. These events are merged together by a *merge function* to form a *materialised view* of the CRDT. The merge function must be associative, commutative and idempotent in order to ensure all nodes converge on the same materialised view, regardless of the order in which they received the events from the network. This presents a problem because group memberships are not commutative: in order for Alice to leave the group, she must already be joined to the group. To handle this, both Keyhive and Matrix encode the causal past in the events they exchange. This encodes the happens-before relation which is required in order to apply access control rules correctly. The happens-before relation is often expressed via dotted version vectors [24]. However, it is unsafe to use version vectors to represent causality in a Byzantine environment due to *equivocation attacks* [15]. When two nodes synchronise events in a CRDT, they typically exchange event IDs as a compressed representation of the underlying event data. Equivocation is the act of a Byzantine node sending different event data to different nodes with the same event ID. This results in convergence failures as correct nodes believe they are in-sync with each other, as both have events with the same ID, but the values in the event are different. To address this, event IDs can be hashes of the event values, and the values can include the event IDs of the causal predecessors. This recursive hash linking forms a Hash Directed Acyclic Graph (hash DAG) of events which can be used to provide Byzantine eventual consistency [21]. To calculate the materialised view of a hash DAG, the merge function is applied to the accumulated state at the sources. The term "sources" is also called "forward extremities" in Matrix or "heads" in Automerge / Keyhive. Only the sources need to be merged because every node in the DAG is an ancestor of at least one source, meaning all history is represented by the CRDT state at those sources.

More formally, for a DAG $G = (V, E)$ and join function $\bigsqcup$, the materialised view MV is:

$$\mathrm{Sources}(G) = \{\, v \in V \mid \deg^{+}(v) = 0 \,\}$$

$$\forall v \in V,\ \exists \ell \in \mathrm{Sources}(G) : \ v \preceq \ell$$

$$\implies \quad \mathrm{MV}(G) = \bigsqcup_{\ell \in \mathrm{Sources}(G)} \ell$$

The act of fetching concurrent events fetches new sources on the DAG. As a result, this may change the *materialised view* of the CRDT. This change may *roll back* previously valid group membership events.

2.1 Finality and Causal Stability

Finality and causal stability are mechanisms for guaranteeing that a given event cannot be rolled back. Almeida defines *causal stability* [6] as:

Definition 2.1 (Causal Stability [6])**.** A message [event] with timestamp t is causally stable at node i when all messages [events] subsequently delivered at i will have timestamp $t' \geq t$.

Timestamps are used in this definition to specify causality between events, whereas hash DAGs use the position of each event in the DAG to specify causality. In contrast, *finality* encapsulates the desired property that an event is "final" and cannot be *rolled back* due to missed concurrent events. In blockchains, this refers to the time when it becomes impossible to roll back a block that has previously been appended [23]. Causal stability is one technique to achieve finality in CRDTs. It does this by ensuring that there are no unknown concurrent events. However, finality only requires that there are no concurrent events which would *roll back* a given event m.

For concurrent events $m \parallel m'$, we say m' *rolls back* m when m's presence has no effect on the materialised view:

$$\mathsf{Rollback}(m', m) \overset{\text{def}}{\iff} \mathrm{MV}(G \in \{m, m'\}) \equiv \mathrm{MV}(G \in \{m'\}) \tag{1}$$

Finality is then the absence of any concurrent event that could cause a rollback:

$$\mathsf{Final}(m) \overset{\text{def}}{\iff} \forall m' : \ m' \parallel m \ \Rightarrow \ \neg\, \mathsf{Rollback}(m', m) \tag{2}$$

Intuitively, whether Alice is joined to the group cannot be affected by Bob saying "hello" in the group, even if it were concurrent. Thus, knowledge of Bob's "hello" is not required for finality.

2.2 Backdating

Backdating attacks occur when a Byzantine node sends an event to intentionally cause an event to be rolled back. Backdating attacks fall into two categories: *detectable* and *undetectable. Detectable backdating* occurs when a node sends an event concurrent with itself. Let $n \in N$ be a node, and let E_n be the set of events sent by n:

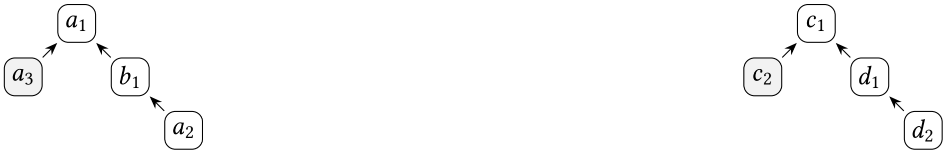
$$\exists e_1, e_2 \in E_n : \quad e_1 \parallel e_2 \tag{3}$$

To detect this, the protocol must forbid this behaviour, meaning any concurrent event sent by the same node is an attempt to backdate:

$$\forall n \in N,\ \forall e_1, e_2 \in E_n : \quad \neg(e_1 \parallel e_2) \tag{4}$$

Undetectable backdating occurs when a node sends an event concurrent to the latest received event, instead of causally after it. This is undetectable because the node is not violating the concurrent events rule: it is indistinguishable from a correct node under poor network conditions who may genuinely not have received some events.

Both forms are shown in Figure 1.



(a) Detectable backdating: Node a backdated event a_3 which is detectable due to it being concurrent with a_2.

(b) Undetectable backdating: it is impossible to know if node c backdated c_2 or if it is just a late arriving event.

Figure 1. DAGs of causal histories demonstrating detectable vs undetectable backdating. Each event points to their causal predecessor. The highlighted event is an attempt to backdate.

3 Problem Statement

Consider a hash DAG of membership events. Each member can have one of the following roles:

1. **Reader:** The member can read events in the DAG. This is the lowest level role.
2. **Writer:** The member can write events into the DAG, in addition to the Reader role.
3. **Admin:** The member can change the role of other members, in addition to the Writer role.

The valid operations are as follows:

1. **join(a):** a joins the group. *Any user* can do this, without prior permission. The first user to join is an Admin. Subsequent users are a Reader.
2. **promote(a, b, role):** a increases the role of b to role. To do this, a must be an Admin.
3. **demote(a, b, role):** a decreases the role of b to role. To do this, a must be an Admin. b may be any role, and it is valid for $a == b$: a *self-demotion.*
4. **write(a):** a writes a message into the DAG. To do this, a must be a Writer or Admin.

The following safety and liveness properties are required from the perspective of a single node:

P1: **Authorization Safety.** No user may successfully execute an operation requiring permissions they do not possess.

P2: **Availability (Liveness).** Authorised users are able to send events even when they cannot communicate with *any* other user.

The protocol allows for the existence of a *causal broadcast* algorithm in order to ensure events are delivered in causal order, such as [21].

Consider the duelling admins problem in Figure 2. What strategy should be used to comply with the safety and liveness properties and ensure a fair outcome?

3.1 Diagram conventions

All diagrams resemble a DAG, where nodes denote events and edges denote hash DAG causal ordering. The position of each node denotes its position in the *execution order* when read from top to bottom, indicating how concurrent events are sorted. Events that are *unauthorised* according to this order are indicated with ×. Annotations to the right of each event show the effective role assignments visible *after* the event is applied at that point in the execution order. Consequently, the *materialised view* is the bottom-most annotation in each diagram. **Bold text** represent events that are backdated.

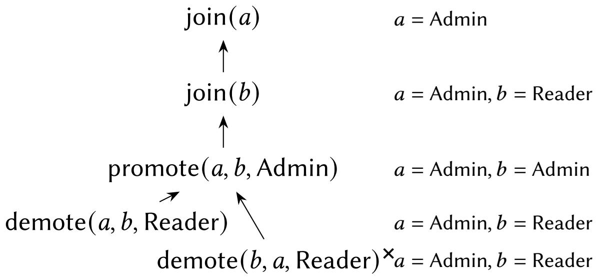


Figure 2. The duelling admins problem. Which user remains an Admin depends on how demote events are sorted. Sorting by user centralises authority. Sorting by event ID causes repeated rollbacks.

3.2 Design Considerations

Firstly, the protocol cannot forbid equal-permissioned users from demoting each other because the user who first gained that permission can backdate, as shown in Figure 3.

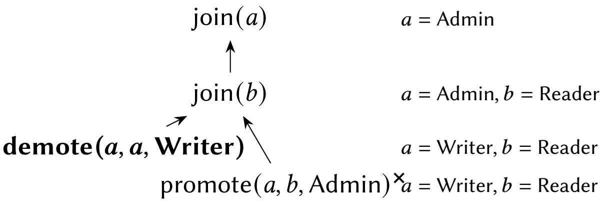

join(a)
a = Admin
join(b)
a = Admin, b = Reader
demote(a, a, Writer)
a = Writer, b = Reader
promote(a, b, Admin)×
a = Writer, b = Reader

Figure 3. Even if equal-permissioned roles are forbidden from demoting each other, backdating can subvert this. demote(a, a, Writer) executes first, meaning a is no longer authorised to promote(a, b, Admin).

Seniority ranking [19] formalises this and is defined as:

> Group creator has rank 1, user added by rank-i user has rank $i+1$, break ties by lexicographical order on hashes of operations that added the users. [20]

Should two concurrent events involve the same user (and thus the same seniority), these events are tie-broken on data within the event e.g. timestamp or event hash. Initially, this appears to enforce the safety property P1, as shown in Figure 4. However, *self-demotions* break the invariant because a malicious user controls all data within the event. This means self-demoted users can backdate events to re-use permissions they gave up, as shown in Figure 5.

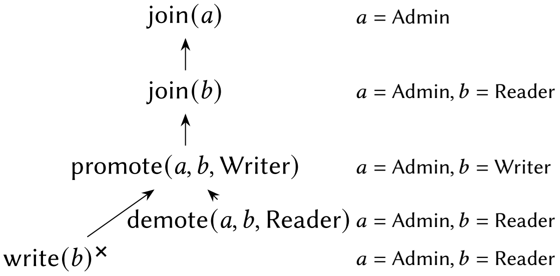

join(a)
a = Admin
join(b)
a = Admin, b = Reader
promote(a, b, Writer)
a = Admin, b = Writer
demote(a, b, Reader)
a = Admin, b = Reader
write(b)×
a = Admin, b = Reader

Figure 4. Admin operations are ordered before Writer operations. As a result, write(b) is not authorised because b is now a Reader.

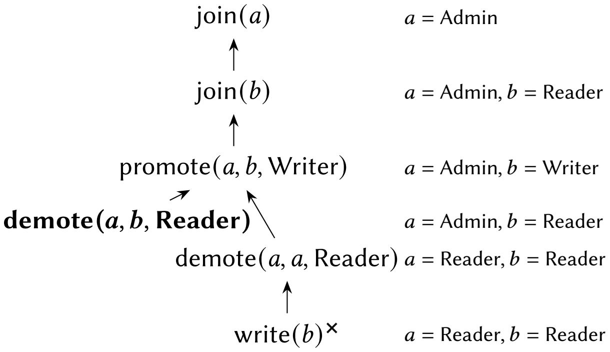

join(a)
a = Admin
join(b)
a = Admin, b = Reader
promote(a, b, Writer)
a = Admin, b = Writer
demote(a, b, Reader)
a = Admin, b = Reader
demote(a, a, Reader)
a = Reader, b = Reader
write(b)×
a = Reader, b = Reader

Figure 5. User a backdated event demote(a, b, Reader) despite not being authorised to demote anymore. Whether it executes before or after demote(a, a, Reader) depends solely on attacker-controlled data. This breaks safety property P1.

The self-demotion and duelling admins problem highlight a tension between the two possible ways events sent by the same permission level can be ordered without consensus.

User-based ordering: This means the protocol arbitrarily and consistently decides which equal-permissioned user will win tie-breaks. Seniority ranking is one such example, where the user who first (according to the lowest graph depth) gained their role will always win. By sorting all users into a total order, this centralises authority as there is always one user who is guaranteed to win any conflicts, even if the roles indicate shared authority. If this most senior user is demoted, they can simply backdate a retaliatory demotion to cancel out their own demotion, breaking safety property P1. This ordering is incomplete when handling concurrent events sent by the same user, resulting in the inability to enforce self-demotions.

Event-based ordering: This means the protocol chooses based on the revoking event's timestamps or event IDs. Matrix's state resolution algorithm [5] is one example of this. This allows equal-permissioned users to both win tie-breaks equally. Malicious users may game this system to consistently win e.g by always using a lower/higher event ID. This can create constant rollbacks as the two admins fight each other.

Options that involve consensus or Proof of X aren't much better. Every Admin could announce the order they saw demotions. The majority would decide the final execution order. This is vulnerable to Sybil attacks [14] because existing Admins can backdate sockpuppet Admin accounts to gain a majority. Proof of Work systems are not appropriate because, as Nakamoto puts it: "Proof-of-work is essentially one-CPU-one-vote" [23]. This essentially approximates user-based ordering as CPUs are not fairly distributed across nodes. Proof of Stake systems are similarly inappropriate because CRDTs lack any inherent economic incentive which would discourage slashing [11]. Even if such an incentive existed, it too would not be fairly distributed [12] among Admins, approximating user-based ordering.

The severity of these problems are exacerbated by backdating, which allows the demoted Admin to retaliate. Is it possible to detect and perform some remediation upon detecting backdating? Almeida and Shapiro [8] propose that Byzantine nodes are excluded from participating upon detecting equivocation (backdating). However, as nodes will not agree *which event* was backdated, some amount of backdating is accepted, making this solution incomplete.

4 Concept: Finality nodes

The concept presented in this section approaches the problem differently by acknowledging that a neutral bystander, a "finality node", is required in order to arbitrate an ordering between the duelling admins. This neutral bystander can also prevent Byzantine nodes from "going back on their word" by backdating events. A finality node is a CRDT participant which periodically announces the event IDs of the current *sources* it has in the DAG. Other CRDT participants use this information when ordering events into an execution order for calculating the materialised view. As such, both duelling admins must trust the finality node is emitting sources honestly.

4.1 Epoch events

Finality nodes announce their current sources via *epoch events*. Assuming finality nodes do not backdate epoch events, this forms onion-like layers of events in each epoch, as shown in Figure 6. Events are then ordered by their epoch *first*, thus finalising the *execution order* of the CRDT, as shown in Figure 7. For example, d_1 is not part of the first epoch, despite it pointing to an event a_2 within the epoch. In the execution order, this means $epoch_1 \rightarrow d_1$. Events not in any epoch are in a *pending epoch* which executes after all other epochs, regardless of what the causal predecessors are in the event itself. Epochs are triggered based on the number of events in the pending epoch, but could also be triggered on-demand for non-monotonic events such as demotions.

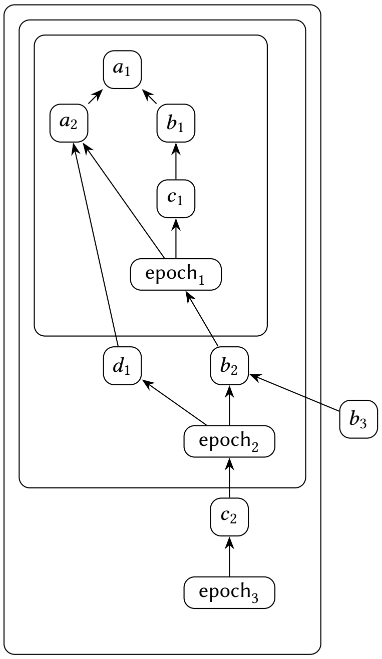


Figure 6. Epochs define a *closed past* [7] to prevent backdating. Event b_3 is in a *pending epoch*, despite it pointing to an earlier event in a previous epoch.

5 Discussion

The motivation behind epochs is to achieve finality for CRDT state in order to meet safety property P1. The duelling admins problem is a manifestation of the lack of finality in CRDT state. This means epochs only need to be sent in response to non-monotonic operations such as the demote operation, which have the possibility to roll back events. This resolves the duelling admins problem without centralising authority (as seniority ranking does) or repeatedly rolling back state (as Matrix's state resolution algorithm does).

Finality nodes also have applications beyond group management CRDTs. Any situation where the materialised view is used to perform sensitive operations at the application level may benefit from finality. For example, an application may use a Positive Negative Counter (PN-Counter) to track outstanding references to a file. When this counter reaches 0, every node deletes the file. It is not safe to delete the file until all concurrent operations are known: this requires coordination. In this circumstance, the application may decide to only delete the file once the counter reaches 0 *and* the counter has been finalised with that value.

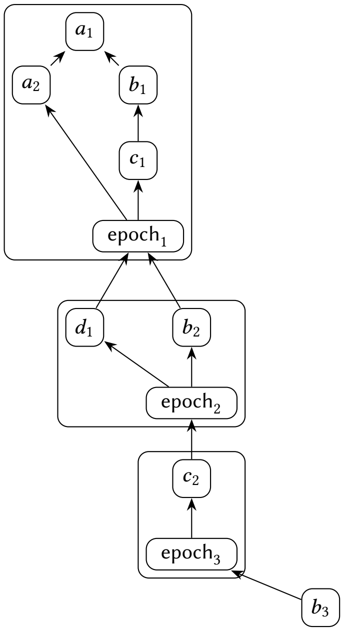


Figure 7. The partial execution order for events in Figure 6 after ordering by epoch. After sorting by epoch, events can then be sorted by revocation, timestamp, event hash, etc.

Traditional consensus requires coordination *before* operations commit [10], but this design enables local execution followed by asynchronous coordination via epochs. This is similar to how finality works in blockchains, where miners do not coordinate to mine blocks but instead gradually converge on a single fork via Proof of X systems. The consistency of finalised events relies on the monotonic creation of epochs by finality nodes. This results in two consequences outlined below.

5.1 Trust

The finality node is trusted to monotonically generate epochs. It is possible to "rewrite history" if a finality node sends concurrent epoch events, as shown in Figure 8.

This is a form of *detectable backdating*, which enables the protocol to take remediation steps. For example, the protocol may define an ordered list of finality nodes, where each node independently issues epoch events but the protocol only uses the first node. Upon detecting backdating, the second node is used, and so on. This does not prevent history from being

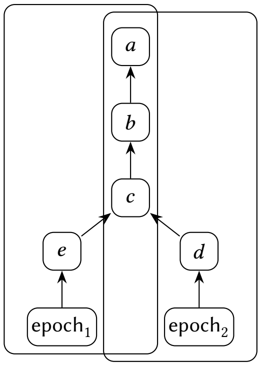


Figure 8. Concurrent epochs can cause history to deterministically re-order events d and e. If epoch$_1$ < epoch$_2$ then $\{a, b, c, e\} \rightarrow d$. If epoch$_2$ < epoch$_1$ then $\{a, b, c, d\} \rightarrow e$.

rewritten as there is no guarantee that each node will agree on which events are in which epoch, but it does prevent a malicious finality node from manipulating CRDT state to its advantage. An additional approach is to encrypt the underlying CRDT state from the finality node. This implies the finality node is a *mutually trusted* third party and not part of the group nor colluding with anyone inside the group, as otherwise they would be able to decrypt the CRDT state. This would not prevent history from being rewritten, but it would make it much more difficult for a malicious finality node to intentionally rewrite history in a way which is beneficial as it would only be able to view ciphertexts. Protocols like Keyhive already encrypt CRDT events and so can more freely delegate responsibility for ordering events to third parties. For CRDTs which operate on the public internet, lists of publicly available finality nodes could be published along with uptime percentages and the number of times that the node has been caught backdating. This would provide a mechanism for users to find and select reliable third party finality nodes. Trust in finality nodes does *not* extend to other CRDT group participants. Group participants, including the creator, may be Byzantine. This is possible because the group creator cannot override the ordering provided by the finality node. This requires participants to trust the finality node that the group creator has selected, which further reinforces the need for public finality node lists. The idea of security critical infrastructure depending upon neutral services is not new: Roughtime [13] provides a secure alternative to NTP and TLS certificates have Certificate Transparency Logs [1].

5.2 Reliability

The finality node must reliably receive events, generate epoch events and transmit them to other nodes. If this does not happen, finality will not be reached on a growing set of events. Despite this, nodes are still able to communicate with each other, preserving liveness property P2. The absence of finality would enable backdating attacks outlined in this article on events in the *pending epoch* only.

If all nodes operate on the public internet, a node with high uptime should be chosen as the finality node. If nodes are not connected to the public internet (e.g. airgapped environments, mobile ad hoc networks (MANETs)), then the finality node should be positioned on the network to maximise both reachability and uptime. This ensures that the finality node has the best chance to be able to receive events and transmit the resulting epoch events to other nodes.

6 Conclusion and Future Work

Finality nodes provide a technique to solve the duelling admins problem in a fair way which guarantees revocation cycles terminate. They guard against undesirable rollbacks and backdating attacks by finalising the execution order of events. If the finality node is unreachable, finalised events are safe from being rewritten, but no other events will be finalised. This provides application designers a choice: security sensitive operations may wait for finality for some operations to enforce safety properties, whilst other operations may occur immediately with weak consistency to enforce liveness properties. This allows applications to choose which operations require consistency (and thus may block) versus which tolerate eventual consistency.

A drawback to this approach is that the finality node is trusted to order events. Finality nodes may intentionally delay ordering events or may intentionally re-order events, the latter being a form of detectable backdating. Two main techniques can be used to disincentivise malicious behaviour. The first is to encrypt CRDT event data from the finality node such that a non-colluding finality node cannot intentionally re-order events. The second is to attach a public reputation to the finality node such that malicious behaviour is socially penalised. Which technique is best depends on the ultimate aims and use cases of the CRDT.

This design separates finalised and unfinalised events. Future research may take advantage of the event separation by having novel algorithms for finalised events vs temporarily unfinalised events. It may also be possible to use finality to gradually mutate the list of finality nodes over time to provide a mechanism to remove Byzantine finality nodes.

Acknowledgments

I thank Martin Kleppmann for helpful discussions and Florian Jacob for detailed feedback on earlier drafts.

References

[1] [n. d.]. *Certificate Transparency: The list of current, usable logs.* https://certificate.transparency.dev/logs/

[2] [n. d.]. *Developing with Riak KV Data Types.* https://docs.riak.com/riak/kv/2.2.3/developing/data-types/

[3] [n. d.]. *Distributed Data - Akka Documentation.* https://doc.akka.io/libraries/akka-core/2.5/distributed-data.html?language=scala

[4] 2025. Matrix Specification v1.17. https://spec.matrix.org/v1.17/

[5] 2026. Matrix Specification v1.17: Room Version 12. https://spec.matrix.org/v1.17/rooms/v12/#state-resolution

[6] Paulo Sérgio Almeida. 2024. Approaches to Conflict-free Replicated Data Types. *ACM Comput. Surv.* 57, 2, Article 51 (Nov. 2024), 36 pages. doi:10.1145/3695249

[7] Paulo Sérgio Almeida. 2024. A Framework for Consistency Models in Distributed Systems. arXiv:2411.16355 [cs.DC] https://arxiv.org/abs/2411.16355

[8] Paulo Sérgio Almeida and Ehud Shapiro. 2025. The Blocklace: A Byzantine-repelling and Universal Conflict-free Replicated Data Type. arXiv:2402.08068 [cs.DC] https://arxiv.org/abs/2402.08068

[9] Keyhive Authors. [n. d.]. *Keyhive Group Membership Design.* https://github.com/inkandswitch/keyhive/blob/keyhive-wasm/0.0.0-alpha.54/design/group_membership.md

[10] Eric Brewer. 2012. CAP Twelve years later: How the "Rules" have Changed. *Computer* 45 (02 2012), 23–29. doi:10.1109/MC.2012.37

[11] Vitalik Buterin, Daniël Reijsbergen, Stefanos Leonardos, and Georgios Piliouras. 2019. Incentives in Ethereum's Hybrid Casper Protocol. In *2019 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*. 236–244. doi:10.1109/BLOC.2019.8751241

[12] Tom Celig, Tim Ockenga, and Detlef Schoder. 2025. Distributional equality in Ethereum? On-chain analysis of Ether supply distribution and supply dynamics. *Humanities and Social Sciences Communications* 12 (03 2025). doi:10.1057/s41599-025-04728-9

[13] Marcus Dansarie. [n. d.]. *Roughtime — datatracker.ietf.org.* https://datatracker.ietf.org/doc/draft-ietf-ntp-roughtime/

[14] John R. Douceur. 2002. The Sybil Attack. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems (IPTPS '01)*. Springer-Verlag, Berlin, Heidelberg, 251–260.

[15] Florian Jacob, Saskia Bayreuther, and Hannes Hartenstein. 2021. On Conflict-Free Replicated Data Types and Equivocation in Byzantine Setups. arXiv:2109.10554 [cs.DC] https://arxiv.org/abs/2109.10554

[16] Florian Jacob, Luca Becker, Jan Grashöfer, and Hannes Hartenstein. 2020. Matrix Decomposition: Analysis of an Access Control Approach on Transaction-based DAGs without Finality. In *Proceedings of the 25th ACM Symposium on Access Control Models and Technologies* (Barcelona, Spain) *(SACMAT '20)*. Association for Computing Machinery, New York, NY, USA, 81–92. doi:10.1145/3381991.3395399

[17] Florian Jacob and Hannes Hartenstein. 2024. Logical Clocks and Monotonicity for Byzantine-Tolerant Replicated Data Types. In *Proceedings of the 11th Workshop on Principles and Practice of Consistency for Distributed Data* (Athens, Greece) *(PaPoC '24)*. Association for Computing Machinery, New York, NY, USA, 37–43. doi:10.1145/3642976.3653034

[18] Florian Jacob and Hannes Hartenstein. 2025. To the Best of Knowledge and Belief: On Eventually Consistent Access Control. In *Proceedings of the Fifteenth ACM Conference on Data and Application Security and Privacy* (Pittsburgh, PA, USA) *(CODASPY '25)*. Association for Computing Machinery, New York, NY, USA, 107–118. doi:10.1145/3714393.3726520

[19] Martin Kleppmann. [n. d.]. *Keynote: Byzantine Eventual Consistency and Local-First Access Control*. https://martin.kleppmann.com/2025/03/31/papoc-keynote-byzantine.html

[20] Martin Kleppmann. 2025. Byzantine eventual consistency and local-first access control. https://speakerdeck.com/ept/byzantine-eventual-consistency-and-local-first-access-control?slide=69

[21] Martin Kleppmann and Heidi Howard. 2020. Byzantine Eventual Consistency and the Fundamental Limits of Peer-to-Peer Databases. arXiv:2012.00472 [cs.DC] https://arxiv.org/abs/2012.00472

[22] Martin Kleppmann, Adam Wiggins, Peter Van Hardenberg, and Mark McGranaghan. 2019. Local-first software: you own your data, in spite of the cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. ACM, Athens Greece, 154–178. doi:10.1145/3359591.3359737

[23] Satoshi Nakamoto. 2009. Bitcoin: A Peer-to-Peer Electronic Cash System. (May 2009). http://www.bitcoin.org/bitcoin.pdf

[24] Nuno Preguiça, Carlos Baquero, Paulo Sérgio Almeida, Victor Fonte, and Ricardo Gonçalves. 2010. Dotted Version Vectors: Logical Clocks for Optimistic Replication. arXiv:1011.5808 [cs.DC] https://arxiv.org/abs/1011.5808

[25] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems*, Xavier Défago, Franck Petit, and Vincent Villain (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 386–400.

[26] Brooklyn Zelenka and Alex Good. [n. d.]. *Keyhive: Local-first Access Control.* https://www.inkandswitch.com/keyhive/notebook/