Logical Clocks and Monotonicity for Byzantine-Tolerant Replicated Data Types

Florian Jacob
florian.jacob@kit.edu
Karlsruhe Institute of Technology
Karlsruhe, Germany

Hannes Hartenstein
hannes.hartenstein@kit.edu
Karlsruhe Institute of Technology
Karlsruhe, Germany

Abstract

Replicated event logbooks are ubiquitous in decentralized systems designed to cope with Byzantine-faulty replicas. Recently, there is a growing subclass that only partially orders its logbooks by hash-linking inscribed events to their causal past. Thereby, these approaches forgo coordination and consensus to gain scalability and availability under partition. We investigate these approaches to explicate their underlying construction by connecting their design to the concept of logical monotonicity and by providing an abstraction as a delta-state conflict-free replicated data type. In particular, we analyze what makes a clock Byzantine-tolerant, and show that these hash-linked causal logbooks represent Byzantine-tolerant clocks. Based on these insight, we model real-world group communication systems as Byzantine monotonic compositions, and analyze their monotonicity properties to understand the guarantees they provide to the application layer.

CCS Concepts: • **Software and its engineering** → **Publish-subscribe / event-based architectures**; **Consistency**; • **Information systems** → *Distributed storage*; • **Computer systems organization** → *Availability*; *Fault-tolerant network topologies*; *Distributed architectures*; • **Security and privacy** → **Distributed systems security**.

Keywords: Autonomous Decentralized Systems, Conflict-Free Replicated Data Types, Byzantine Fault Tolerance, Logical Clocks, Logical Monotonicity, Matrix, Wesh, IPFS

ACM Reference Format:
Florian Jacob and Hannes Hartenstein. 2024. Logical Clocks and Monotonicity for Byzantine-Tolerant Replicated Data Types. In *Principles and Practice of Consistency for Distributed Data (PaPoC '24), April 22, 2024, Athens, Greece.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3642976.3653034

This work is licensed under a Creative Commons Attribution International 4.0 License.
PaPoC '24, April 22, 2024, Athens, Greece
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0544-1/24/04
https://doi.org/10.1145/3642976.3653034

1 Introduction

Decentralized systems for group communication and data storage that support availability under partition in open environments are typically based on replicated data types. Examples include the Matrix protocol [31] used by the Element instant messenger, the Wesh protocol [6, 30] used by the Berty instant messenger, InterPlanetary Filesystem (IPFS)-based conflict-free replicated data types (CRDTs) like OrbitDB [13, 27], the local-first auth library [7], or the Braid state synchronization protocol [11, 32]. Matrix stands out with its comparatively wide adoption: more than 100 000 000 users distributed on more than 100 000 servers are found in the public federation, and nation states like France and Germany operate their own private federations for sovereignty reasons [14]. All these examples have in common that they use partially-ordered causal event logbooks which enable autonomous replicas, i.e., availability under partition.

In this paper, we investigate these autonomous decentralized algorithms to explicate their underlying construction: they all use recursive hash commitments to hash-link events to their causal histories, an approach we call recursive hash histories, to encode their logbooks in a Byzantine fault-tolerant way. The use of recursive hash histories for Byzantine fault-tolerant, coordination-free CRDTs has already been discussed in [16, 20, 27]. We now connect their design to the concept of logical monotonicity [12] and provide an abstraction in the form of a delta-state CRDT. While logically monotonic algorithms are well-known in crash fault and omission fault environments, we are specifically interested in the consequences of logical monotonicity in Byzantine environments, and algorithms to achieve it. In Section 3 we show the need for *Byzantine monotone logical clocks* to be able to cope with equivocation, formulate the *replicated chronicle problem*, and survey solutions to the problem in the Byzantine setup. We then formalize in Section 4 the *recursive hash history* approach as a delta-state replicated data type, which we call a *hash chronicle*, and show that hash chronicles are a class of Byzantine-monotonic logical clocks. In Section 5, we then model various practical systems as hash chronicle based compositions that are logically monotonic in Byzantine environments and thereby show that the analysis of monotonicity builds the foundation for the analysis of application-layer guarantees.

2 Fundamentals and Related Work

In decentralized systems, event logbooks or ledgers are often totally ordered via consensus, so that all replicas agree that all replicas have the same event chain. In this paper, however, we investigate "non-consensus based" systems and refer to them as *autonomous decentralized systems* [26] (ADS). ADS forgo coordination as a prerequisite for state progression. In literature, the ADS class is also called coordination-free, coordination-avoiding, or local-first. To remain available under partition, operation latency must be independent of the network latency [19]. Therefore, replicas cannot coordinate with each other to ensure safety properties, as waiting on other replicas would violate liveness properties. Specifically, availability under partition rules out consensus on a total order of events, as usually implemented in distributed ledgers.

Both in crash-fault and Byzantine environments, the ADS class is characterized by its fundamental limit to confluent invariants [4, 21]. An invariant is confluent if when all replicas locally ensure the invariant, it also holds globally, whereby it can be ensured under partition. For example, the invariant that a counter is grow-only is confluent, while the invariant that the counter is smaller than an upper bound is not.

The CALM (Consistency As Logical Monotonicity) theorem [12] provides a (sufficient) condition for confluent invariance [4] that we build upon in this paper. While the notion of 'logical monotonicity' comes from monotonicity of entailment and the background of Datalog, we simply make use of the order-theoretic definition of monotonicity: A function $f: I \longrightarrow O$ between two partially-ordered sets (posets) I, O is order-preserving, or monotone, if $x \leq_I y \Rightarrow f(x) \leq_O f(y)$. For more than one input or output, I and O can be composed using the Cartesian product $I = I_1 \times I_2$. An endomorphism $f: I \longrightarrow I$ is an inflation if $x \leq_I f(x)$. A semilattice homomorphism, short morphism, is a monotone function that preserves the semilattice structure of inputs and outputs, $m: I \longrightarrow O, m(i_1 \sqcup_I i_2) = m(i_1) \sqcup_O m(i_2)$. Monotone functions and inflations on join-semilattices provide the 'mechanics' of logical clocks and CRDTs [28, 29].

A monotonic distributed algorithm consists of a set of replicas that execute query, join, and mutate functions as follows: Queries are monotonic functions that map the current replica state to an output value. The join function is a monotonic inflation that takes two replica states and derives the least upper bound as joined output, and forms a join-semilattice with the partially-ordered state space [9]. Mutate functions take the current replica state and a user operation, and map them to an element of the state space. To apply a mutation, replicas join the result with their local state and send state to remote replicas as input. On receiving remote state, replicas join it with their local state, and inflate their state with the output.

In crash-/omission-fault systems, vector time logical clocks like vector clocks [24] or version vectors [1] are used heavily as monotonicity mechanisms and for replicating partially-ordered logs [34]. However, they are not robust against Byzantine behavior. Systems like Matrix or IFPS are instead based on hash-linked directed acyclic graphs (HashDAGs), a Byzantine fault-tolerant data structure available under partition [16]. HashDAGs are a recursive variation of the hash history approach [18]. Recursive hash histories have been independently discovered multiple times, in theory [16, 21, 27] as well as in the practical approaches from Section 1. In the IPFS context, HashDAGs are called Merkle DAGs and recursive hash histories are called Merkle clocks [27]. Merkle Search Trees [3] allow efficient reconciliation of HashDAGs.

CRDTs typically fulfill CALM only regarding their internal state, but provide no guarantees on derived query results. Recent papers envision to compose CRDTs from causal event logbooks [5, 33], and to combine CRDT properties of updates and monotonicity of queries on replicated state [8, 22]. To follow this vision we show the usefulness of logical monotonicity in analyzing and synthesizing Byzantine-tolerant ADS and describe protocols as a Byzantine-monotonic composition of lattices, logical clocks, morphisms, and monotone functions, encompassing both state and queries.

Both this and our previous paper [17] are concerned with Byzantine-tolerant extend-only partially-ordered sets. While we were agnostic of the partial order in the previous paper, we now focus on causal event sets and analyze their connection to logical clocks and monotonicity, as well as demonstrate their use in modeling practical systems.

3 Analysis: Byzantine Monotonicity

3.1 Challenge

To the best of our knowledge, CALM analysis has not yet been applied in Byzantine environments. However, as a mathematical property of functions, the definition of monotonicity does not depend on a fault model, and the CALM theorem also applies to Byzantine environments. We show that monotonicity is a strong defense against Byzantine behavior. However, monotonicity of a distributed algorithm might depend on invariants that the sender is trusted not to violate, as receivers cannot verify them without coordination. For example, logical clocks are monotone functions c on causally-ordered events: $e_1 \leq e_2 \implies c(e_1) \leq c(e_2)$. When using a classical logical clock like vector clocks or Lamport clocks, monotonicity rests on the assumption that replicas assign a total order on outgoing events: sequence numbers are a semilattice and should be only modified by strict inflations. This assumption cannot be verified without coordination: it does not hold globally when replicas check for it locally, and thereby, classical clocks are not Byzantine-monotonic.

We now analyze what it means to be monotonic under Byzantine faults. Under the assumption of a connected component of correct replicas in an asynchronous, unreliable

network formed by authentic and integrity protected communication channels, e.g., using digital signatures, Byzantine replicas have three attack vectors: malformation, omission, and equivocation of messages [15].

Malformation means a message from a Byzantine replica violates a confluent invariant, i.e., an invariant that can be verified by a replica on its own, like syntactical correctness or signature validity. Malformed messages can be filtered using a monotonic function like intersecting the set of received messages with the set of valid messages. Thus, malformation is not an issue.

Omission on a message means that a Byzantine replica adds the message to the input sets of a strict subset of all replicas that should have received it. In face of non-Byzantine but unreliable replicas connected via unreliable links, monotonic algorithms typically use gossip/epidemic broadcast to ensure that replica states converge via their join semilattice. In comparison, Byzantine omission is no additional challenge.

The attack vector that poses a specific challenge in Byzantine environments is **equivocation**, which means that a Byzantine replica creates two different valid inputs and adds them to non-overlapping strict subsets of the set of replicas it should have sent the input to. With the goal of exploiting an order dependence in the algorithm, the inputs do not violate invariants on their own, but may do so in conjunction. This is the case in the introductory example of classical clocks: The binding between logical timestamp and event ought to be unequivocal, but a Byzantine replica can equivocate by skipping the inflation of logical time to assign the same logical timestamp to different events. Remote replicas cannot locally verify that a given logical timestamp is unique and bound to the presented event. Which version of an event ends up in the output of a replica for the given timestamp then depends on the order in which the two messages arrive at a replica, and thereby is not monotonic. Thus, the challenge of Byzantine monotonicity is to ensure local verifiability of invariants which are required for the monotonicity of functions whose outputs are shared with other replicas.

3.2 Replicated Chronicle Problem

We define the problem of replicating chronicles as a common abstraction for approaches already present in literature and practice. Chronicles are partially-ordered event logbooks: on inscribing an event, they ensure that not only the event itself, but also its predecessors and the causal relation among them are immutable and included in the logbook. The replicated chronicle problem is to maintain replica state so that their local chronicle is a lower bound of the conceptual global chronicle of all occurred events, and evolves monotonically towards the global chronicle. A distributed algorithm for the problem gets as input the local chronicle state, new events that occurred at the local replica, and replication state received from remote replicas, to output the inflated local chronicle and replication state for other replicas.

The replicated chronicle problem is monotonic, i.e., solvable by a monotonic algorithm: Conceptually, a chronicle only requires a grow-only set of events, a grow-only partial order relation among events, and replication by joining replica states via set union. Thereby, the core of the problem is finding a logical clock function that enables *efficient and fault tolerant* replication. While the problem itself is independent of fault models, we are specifically interested in studying the problem under the lens of Byzantine-tolerant monotonicity. We now state notation on events and causality to formally define chronicles and the replicated chronicle problem. Solutions are studied in Section 3.3.

An event $e \in \mathbb{E}$ from the set of valid events $\mathbb{E}$ in a distributed system describes the execution of a discrete action in spatial and causal context, e.g., via a replica identifier $r \in R$ and logical timestamps $t \in \mathbb{T}$ from a logical clock function $c\colon \mathbb{E} \longrightarrow \mathbb{T}$. Causal precedence $\preceq$ is a partial order on events [28], an event $\hat{e}$ may only influence an event e if and only if $\hat{e} \preceq e$. The set of events that causally precede an event e is its causal $past(e) = \{\hat{e} \in \mathbb{E} \mid \hat{e} \preceq e\}$, the set of events that causally succeed e is its causal $future(e) = \{\check{e} \in \mathbb{E} \mid \check{e} \succeq e\}$. While the order in which events occur in distributed systems is observer-relative, the causal past and future of an event is observer-invariant. The causal concurrency relation is defined as $e_\parallel \parallel e \Leftrightarrow e \npreceq e_\parallel \wedge e_\parallel \npreceq e$. The order of concurrent events is observer-relative, but due to being causally concurrent, those cannot influence each other.

We call a set of events partially-ordered by their causality relation a *causal event set*. A causal event set is downward-closed if the causal past of every event is also part of the set. We assume that the causal past of every event e contains a minimum event $e_\bot \in \mathbb{E}$ that is older than every other event in its causal past. We call $e_\bot$ their *genesis event*. A downward-closed causal event set where all events share the same genesis event is downward-directed at the genesis event [17], and we call such an event set a *chronicle*. Chronicles are partially-ordered, i.e., chronicle C' is a superset of C only if it includes all events of C and their immutable causal past: $C \subseteq C' \Leftrightarrow \forall e \in C\colon e \in C' \wedge past(e \in C) = past(e \in C')$.

The chronicle C_r of a correct replica $r \in R$ contains its current knowledge on occurred events and their causal order in the system. Event ordering is defined by the replica where an event originated: The causal past of an event is always a lower bound on the origin replica's knowledge. The replicated chronicle problem is the problem of spreading and merging knowledge on events and their order among correct replicas so that their local chronicles converge to a global chronicle $C = \bigcup_{r \in R} C_r$. A distributed algorithm solves the problem if given locally-occurred events and incoming replication messages as input, it outputs local replica state and outgoing replication messages, so that any correct replica from the set of correct replicas $r \in R$ fulfill *Chronicality* and *Monotonicity* as safety conditions, and *Eventual Delivery* as liveness condition:

Chronicality The local state C_r is always a chronicle, i.e., a causal event set that is downward-closed and directed at the minimal event $e_\perp$ of global state C.

$$\forall r \in R, \forall e \in C_r\colon \quad \forall \bar{e} \in C\colon \quad e \leq \bar{e} \lor \bar{e} \leq e \lor e \parallel \bar{e} \qquad (1)$$

$$\forall \bar{e} \in C\colon \quad \bar{e} \leq e \implies \bar{e} \in C_r \qquad (2)$$

$$\exists e_\perp \in C\colon \quad \forall \bar{e} \in C\colon e_\perp \leq \bar{e} \qquad (3)$$

Monotonicity The next local chronicle C'_r is an inflation of the current C_r towards global C, i.e., the sequence of replica states is monotonic.

$$\forall r \in R\colon C_r \subseteq C'_r \subseteq C \qquad (4)$$

Eventual Delivery If an event is included in one local state, it is eventually included in all local states.

$$\forall e \in \mathbb{E}\colon (\exists r \in R\colon e \in C_r \implies \Diamond \forall a \in R\colon e \in C_a) \qquad (5)$$

Monotonic algorithms are sufficient to fulfill these conditions and thereby to solve the replicated chronicle problem, as the chronicle state space is a semilattice: The chronicle state space is the set of downward-closed causal event sets $\mathcal{D}(\mathbb{E})$, and the set of downward-closed subsets $\mathcal{D}(X)$ of any partially-ordered set X is a semilattice. Due to the semilattice state space, a trivial solution of the replicated chronicle problem is a state-based CRDT. At the core of more efficient solutions lies a logical clock c that encodes the causal order of an event in way that is not only order-preserving, i.e., monotone, but also order-reflecting, i.e., $c(e_1) \leq c(e_2) \Rightarrow e_1 \leq e_2$, to enable decoding the order. An order-preserving and order-reflecting function is order-embedding, $e_1 \leq e_2 \Leftrightarrow c(e_1) \leq c(e_2)$. From the perspective of monotonic algorithms, logical clocks are just a morphism from the causal event semilattice to a logical timestamp semilattice [28].

3.3 Byzantine Monotonic Solutions

Schwarz' and Mattern's 1994 seminal paper on causality analysis [28] includes a logical clock that maps an event to its causal past, $c_p(e) = past(e)$. We exemplify this clock with a chronicle in Fig. 1. While being featured as "[...] only of theoretical interest, because the size of the causal history sets is of the order of the total number of events [...]", the clock allows a simple solution to the Byzantine replicated chronicle problem, as this clock is Byzantine-monotonic [17]: by defining events as equal only if their causal past is equal, equivocation leads to separate, concurrent events whose order, by definition, is observer-relative.

The timestamp size of logical clock c_p can be improved using a form of hash commitments: a replica sends a hash digest as commitment on the hash's preimage, and later sends the preimage as reveal. Such hash commitments are computationally binding when used with a collision-resistant hash function, but intentionally not hiding, so that a replica which already knows the preimage from another source does not need the reveal. Hash commitments for improving efficiency of chronicle replication were first seen in [18], albeit in a

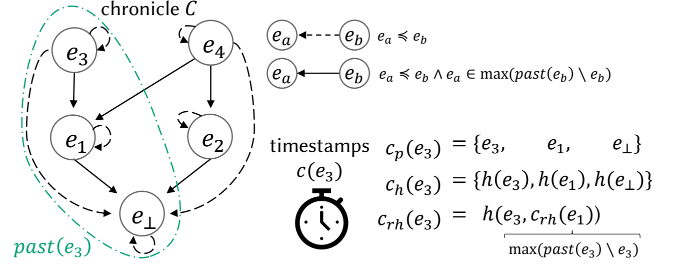

Figure 1. Example chronicle C and logical timestamps assigned to event e_3 by causal past c_p, hash history c_h, and recursive hash history c_{rh} Byzantine-monotonic logical clocks.

non-Byzantine environment. Kang et al. call this the "hash history approach for reconciling mutual inconsistency": they synchronize chronicles by encoding each event with a hash commitment on the event, and encode the causality relation using parent-child-pairs of hashes to link events.

To further reduce timestamp size compared to hash histories, downward-closed sets can be represented compactly by their maximal elements. Hence, links between an event and its direct predecessors are sufficient to encode chronicles: Given a poset X, the function $\max(X) = \{x \in X \mid \forall y \in X\colon x \leq y \Rightarrow y \leq x\}$ returns the set of maximal elements, and set $\mathcal{M}(X) = \{\max(S) \mid S \in \mathcal{P}(X)\}$ consists of the sets of maximal elements of all subsets of X. The set $\mathcal{M}(X)$ is also a semilattice, and isomorphic to the semilattice of downward-closed subsets [5]. The isomorphism can be utilized to compress timestamps of hash histories: Instead of having parent-child-pairs separate from the event hash set, an event hash is recursively constructed by hashing the event together with the set of hashes of the maximal events in its strict causal past. This way, every event hash is a hash commitment on both the event itself as well as a recursive hash commitment on its strict causal past, hence we name this the *recursive hash history* approach. A chronicle can then be compressed to the recursive history hashes of of its maximal events, while receiving replicas can locally verify the binding between event hash, event, and its ordering. As in authenticated data structures [25], recursive hash commitments transferably ensure authenticity and integrity of the committed-to chronicle in Byzantine environments. In addition to Byzantine monotonicity, the timestamp size of (recursive) hash histories is independent of the number of replicas, which is, compared to vector time, favorable in open systems with a high number and churn of replicas [3]. However, in contrast to vector time, hash commitments compress events and chronicles just for efficient synchronization, they need to be revealed later in the protocol to reconstruct the causal order. Without actually synchronizing and revealing the full history, we cannot say whether one recursive hash history logical timestamp, i.e., event hash, is in the causal past or concurrent to another timestamp, unless they are direct predecessors or the same.

4 Synthesis: Delta-State Hash Chronicles

Like regular state-based CRDTs, delta-state CRDTs have a state space lattice. But instead of gossiping their full state, delta-state CRDTs only gossip a lattice elements that describe state change deltas [10]. In general, delta-state CRDTs are only crash fault tolerant. For example, causal δ-CRDTs [2] are version vector-based chronicles. We now formalize *hash chronicles* as *Byzantine fault-tolerant delta-state CRDT*. Hash chronicles are based on using recursive history hashing $c_{rh}(e) = h(e, \{c_{rh}(\hat{e}) \mid \hat{e} \in \max(past(e) \setminus e)\})$ as logical clock: The recursive history hash acts as event timestamp that allows comparison as soon as its causal past is revealed.

As shown in Algorithm 1, we replicate a local causal event set E with set union as join and subset inclusion as partial order, and map that set to its downward-closed subset, i.e., its embedded *chronicle*(). For every event e, there is a record of the set of recursive history hashes $t(e) = \{c_{rh}(\hat{e}) \mid \hat{e} \in \max(past(e) \setminus e)\}$ of e's direct predecessors. The chronicle is monotonically derived by *close_chronicle*() based on $t(e)$: starting from the bottom event $e_\perp$, we iteratively follow $t(e)$ by adding events whose past is included in the current chronicle C, until only fragmented events whose connection to $e_\perp$ is yet unknown are left. The monotonic queries can then be applied on the chronicle, e.g., to query the causal relation among events, or get the chronicle's minimal genesis event $bot(C)$. The set of maximal events $now(C)$ represent the chronicle's causal presence – applying the clock function c_{rh} on every element returns the current logical timestamp. In δ-CRDTs, mutate functions return a δ-update, i.e., an element of the state semilattice that, when merged with local state, cause the desired state inflation. The *inscribe* function is for adding a new event e to the chronicle: It adds the replica's current logical timestamp as $t(e)$ to the event e, and encloses the event in a set. On *inscribe* operations, the δ-update is joined with the current causal event set, and gossiped. Implementations may batch multiple events together in a δ-update to increase efficiency. On receiving δ-updates, replicas verify via *assert* that events contain at least one recursive history hash before joining, with the genesis event being the only exception, and otherwise interrupt the function with an error. On potential fragmentation of the causal event set, a receiving replica cannot ensure whether the casual history of a received event is directed at $e_\perp$, or whether the causal past exists at all. To combat fragmentation, replicas periodically gossip the set of maximal events $now(C)$, whose recursive history hashes act as chronicle compression. Replicas request missing events needed to connect fragmented events, but otherwise ignore fragmented events as long as the commitments on their pasts are not revealed. Dangling fragmented events decrease efficiency, but do not hurt Byzantine monotonicity due to the transitivity of causality. Periodic gossiping of the set $now(C)$ ensures eventual delivery in absence of updates, and may be optimized using logical timestamps.

Algorithm 1 Delta-State Hash Chronicle (run by each replica). Given are the universe of events $\mathbb{E}$, the genesis event $e_\perp$, and the recursive history hash function c_{rh}.

state causal event set $E \in \mathcal{P}(\mathbb{E})$ ▹ $\mathcal{P}(E)$ is a semilattice
 $\sqsubseteq (E_1, E_2 \subseteq \mathbb{E}) : E_1 \sqsubseteq E_2 :\Leftrightarrow E_1 \subseteq E_2$
 $\sqcup (E_1, E_2 \subseteq \mathbb{E}) : E_1 \sqcup E_2 :\Leftrightarrow E_1 \cup E_2$
initial $E \leftarrow \{e_\perp\}$
query *chronicle* $(E) : C = close_chronicle(\{e_\perp\}) \in \mathcal{D}(E)$
 ▹ return largest downward-closed subset directed at $e_\perp$
 ▹ $\mathcal{D}(E)$ denotes downward closed subsets of E.
function *close_chronicle* $(C \subseteq E) : C' \in \mathcal{D}(E)$
 $C^\dagger \leftarrow \{e \in E \setminus C \mid t(e) \subseteq \{c_{rh}(e) \mid e \in C\}\}$
 $C' \leftarrow C \cup C^\dagger$
 if $C^\dagger \neq \emptyset$ **then**
  $C' \leftarrow close_chronicle(C')$
query *past* $(e \in C) : \hat{C}_e \in \mathcal{D}(C)$
 $\hat{C}_e \leftarrow \{\hat{e} \in C \mid c_{rh}(\hat{e}) \in t(e)\}$
 $\hat{C}_e \leftarrow \{e\} \cup \hat{C}_e \cup \bigcup\{past(\hat{e}) \in \hat{C}_e\}$
query *future* $(e \in C) : \check{C}_e = (C \setminus past(e)) \cup \{e\}$
query $\preceq (e_1, e_2 \in C) : e_1 \preceq e_2 :\Leftrightarrow e_1 \in past(e_2)$
query $\parallel (e_1, e_2 \in C) : e_1 \parallel e_2 :\Leftrightarrow \neg(e_1 \preceq e_2) \land \neg(e_2 \preceq e_1)$
query *bot* $(C \in \mathcal{D}(\mathbb{E})) : e_\perp = \perp(C) \in C$
query *now* $(C \in \mathcal{D}(\mathbb{E})) : C_{\max} = \max(C) \in \mathcal{M}(C)$
▹ $\mathcal{M}(C)$ denotes sets of maximal elements of subsets of C

mutate *inscribe* $(e \in \mathbb{E}) : \delta \subseteq \mathbb{E}$
 $t(e) \leftarrow \{c_{rh}(\hat{e}) \mid \hat{e} \in now(chronicle(E))\}$
 $\delta \leftarrow \{e\}$
on operation(*inscribe*(e))
 $\delta = inscribe(e)$
 $E' \leftarrow E \sqcup \delta$
 gossip(δ)
on receive($\delta \subseteq \mathbb{E}$)
 assert $\forall e \in \delta\colon t(e) \neq \emptyset \lor e = e_\perp$
 $E' \leftarrow E \sqcup \delta$
periodically
 gossip(*now*(*chronicle*(E)))
 request($\bigcup\{t(e) \mid e \in E'\} \setminus \{c_{rh}(e) \mid e \in E\}$)

5 Case Study

To practically demonstrate our claim that Byzantine monotonicity is a suitable formalism, we model a number of practical systems as a decomposition into semilattices and monotonic functions, and verify their Byzantine monotonicity. The decomposition highlights hash chronicles as common denominator. Chronicles are used to organize group communication and data storage, thus, an event is a change to replicated communication history or other shared data, like group name, membership, or permissions. All cases studied have in common that a chronicle is topologically

sorted to a causality-preserving event chain. However, in an autonomous setting, the chain cannot grow monotonically, as that would require coordination on chain positions. In contrast, monotonic queries return the sets of predecessors and successors of any event in the chain. To define a unique topological sorting, the chronicle is composed with other semilattices using the lexicographical product. The lexicographic product $\boxtimes$ composes two partial orders by giving the first component a higher priority in comparison, and only if equal, the second component is considered [5]: $(x_1, y_1) \sqsubseteq (x_2, y_2) = x_1 \sqsubset x_2 \lor (x_1 = x_2 \land y_1 \sqsubseteq y_2)$.

OrbitDB [13] is a peer-to-peer CRDT database library. As a practical implementation of the concepts of Sanjuan et al. [27], it is based on IPFS for data storage and gossiping. OrbitDB's foundation is their OpLog, a hash chronicle of CRDT operations. The OpLog acts as foundation to compose complex CRDTs from the included operations. For topological ordering, OpLogs use extended Lamport clocks [23], which assign a natural number to an event e which correspond to the maximal length of a chain of events between $e_\perp$ and e, in addition to a replica identifier from a totally-ordered set $R_\leq$. The composition of causal event sets with Lamport timestamps $\mathbb{E}_\leq \boxtimes (\mathbb{N}_\leq \boxtimes R_\leq)$ inherits the chain property from the timestamps only if the clock function $c\colon \mathbb{E} \longrightarrow (\mathbb{N} \boxtimes R)$ is injective. Byzantine replicas can break injectivity through equivocation of two events with the same Lamport timestamp, whereby Byzantine replicas can force an order dependency in the sense that which events come first in the event chain depends on their arrival order at a specific replica. Depending on the application, e.g., when concurrent operations are commutative, this inconsistency might be tolerable.

Wesh is a decentralized peer-to-peer communication protocol [6], used by the Berty instant messenger [30]. While there are plans to independently implement hash chronicles, Wesh is currently based on OrbitDB's OpLog. Per communication group, Wesh uses two chronicles: one for data like chat messages, one for metadata like permissions. The intention is to be able to employ different retention policies for the chronicles, maximizing privacy for the data chronicle and security for the metadata chronicle. However, administrative events and regular events need a common, shared causality relation [35], i.e., need to share a chronicle for secure enforcement. In conjunction with equivocation on the Lamport clock, this has the potential of Byzantine non-monotonicity.

Local-first Auth [7] is a library for autonomous decentralized authentication, authorization, and group management, intended for collaborative applications. Authorizations are stored in hash chronicles. Concurrent events are treated by application-specific sort and filter functions. It is up to domain specific logic to ensure Byzantine monotonicity of topological sorting, i.e., avoid equivocation opportunities to ensure that the set of predecessors and successors in the chain is monotonic. Filters may introduce non-monotonicity in form of order dependencies as well.

Matrix is a federated protocol for communication and data storage [31]. Matrix stores administrative events and regular events in the same hash chronicle. Matrix employs a multi-level topological sorting algorithm, that we now decompose to analyze Byzantine monotonicity. The event set $\mathbb{E}_r \subsetneq \mathbb{E}$ consists of all administrative events that may revoke privileges. The first sorting criterion is the permission level of the creator at the time of creating the event, $l\colon \mathbb{E} \longrightarrow \mathbb{Z}$. Sorting concurrent events descending by permission level if they are in $\mathbb{E}_r$, i.e., $e \in \mathbb{E}_r \mapsto l(e), e \notin \mathbb{E}_r \mapsto \perp$, leads to composition $\mathbb{E}_\leq \boxtimes \mathbb{Z}_\geq$. For any event, there is an administrative event that authorized it. Concurrent non-revocation events are ordered increasingly by the length of the chain of authorizing administrative events up to the genesis event, which leads to $\mathbb{E}_\leq \boxtimes \mathbb{Z}_\geq \boxtimes \mathbb{N}_\leq$. All events are then ordered increasingly by the wall-clock timestamp $w\colon \mathbb{E} \longrightarrow \mathbb{T}$ of the event's origin replica, and finally by the recursive history hash $c_{rh}\colon \mathbb{E} \longrightarrow \mathbb{N}$, so we end up with $\mathbb{E}_\leq \boxtimes \mathbb{Z}_\geq \boxtimes \mathbb{N}_\leq \boxtimes \mathbb{T}_\leq \boxtimes \mathbb{N}_\leq$.

As receiving replicas cannot verify the wall-clock timestamp, from a theoretical point of view, it is of no significant difference to Lamport clocks in Byzantine environments in the other cases (while potentially being more meaningful to show to users in practice). However, under the assumption that $c_{rh}\colon \mathbb{E} \longrightarrow \mathbb{N}$ is "practically" injective due to the collision resistance of the hash function, the chain property of the hash images is inherited to the composition. Thereby, in contrast to the other approaches, the consistency of the event chain is order-independent, i.e., the set of predecessors and successors in the chain is Byzantine-monotonic.

Summary: Under the lens of the Byzantine monotonicity concept, not all of the analyzed systems can be considered as sufficiently Byzantine monotone. Thus, the case study demonstrated the suitability and value of the proposed concept, and forms a basis for further analysis and system design.

6 Conclusion

Partially-ordered append-only event logbooks are a common foundation in autonomous decentralized systems. In this paper, we formalized event logbooks as "hash chronicle" replicated data type that achieves state and query monotonicity under Byzantine faults. The foundation of this data type is a Byzantine-tolerant logical clock. We showed that the concept of monotonicity facilitates the description and analysis of deployed Byzantine fault tolerance systems by modeling them as composition of lattices and monotone functions. Our analysis case study also revealed 'subtle' differences that might significantly affect application layer guarantees.

Acknowledgments

This work was funded by the Helmholtz Pilot Program Core Informatics. We like to thank Marc Shapiro for his suggestion to study the monotonicity of hash chronicles, and Matthew Weidner for his thoughts on chronicle-based composition.

References

[1] Paulo Sérgio Almeida, Carlos Baquero, Ricardo Gonçalves, Nuno Preguiça, and Victor Fonte. 2014. Scalable and Accurate Causality Tracking for Eventually Consistent Stores. In *Distributed Applications and Interoperable Systems*, Vol. 8460. Springer, Heidelberg. https://doi.org/10.1007/978-3-662-43352-2_6

[2] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. 2018. Delta State Replicated Data Types. *J. Parallel and Distrib. Comput.* 111 (Jan. 2018). https://doi.org/10.1016/j.jpdc.2017.08.003

[3] Alex Auvolat and François Taïani. 2019. Merkle Search Trees: Efficient State-Based CRDTs in Open Networks. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*. https://doi.org/10.1109/SRDS47363.2019.00032

[4] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014. Coordination Avoidance in Database Systems. *Proceedings of the VLDB Endowment* 8, 3 (Nov. 2014). https://doi.org/10.14778/2735508.2735509

[5] Carlos Baquero, Paulo Sérgio Almeida, Alcino Cunha, and Carla Ferreira. 2017. Composition in State-based Replicated Data Types. *Bulletin of the European Association for Theoretical Computer Science* 123 (Oct. 2017). https://run.unl.pt/handle/10362/93093

[6] Berty Technologies. 2023. *Wesh Protocol.* Technical Report. https://berty.tech/docs/protocol

[7] Herb Caudill. 2024. LF-Auth. https://github.com/local-first-web/auth

[8] Kevin Clancy and Heather Miller. 2017. Monotonicity Types for Distributed Dataflow. In *Proceedings of the Programming Models and Languages for Distributed Computing (PMLDC '17)*. ACM, New York. https://doi.org/10.1145/3166089.3166090

[9] Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. 2012. Logic and Lattices for Distributed Programming. In *Proceedings of the Third ACM Symposium on Cloud Computing* (San Jose, California) *(SoCC '12)*. ACM, Article 1. https://doi.org/10.1145/2391229.2391230

[10] Vitor Enes, Paulo Sérgio Almeida, Carlos Baquero, and João Leitão. 2019. Efficient Synchronization of State-Based CRDTs. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. https://doi.org/10.1109/ICDE.2019.00022

[11] Braid Working Group. 2024. Braid: Interoperable State Synchronization. https://braid.org/

[12] Joseph M. Hellerstein and Peter Alvaro. 2020. Keeping CALM: When Distributed Consistency Is Easy. *Commun. ACM* 63, 9 (Aug. 2020). https://doi.org/10.1145/3369736

[13] Mark Robert Henderson, Samuli Pöyhtäri, Vesa-Ville Piiroinen, Juuso Räsänen, Shams Methnani, and Richard Littauer. 2022. The OrbitDB Field Manual. (Aug. 2022). https://github.com/orbitdb/field-manual/blob/main/dist/Book.pdf

[14] Matthew Hodgson. 2023. Matrix 2.0. https://archive.fosdem.org/2023/schedule/event/matrix20/

[15] Florian Jacob, Saskia Bayreuther, and Hannes Hartenstein. 2022. On CRDTs in Byzantine Environments. In *Sicherheit 2022 : Sicherheit, Schutz und Zuverlässigkeit* (Karlsruhe, Germany). Gesellschaft für Informatik, Bonn. https://doi.org/10.18420/sicherheit2022_07

[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 (SACMAT '20)*. ACM, New York. https://doi.org/10.1145/3381991.3395399

[17] Florian Jacob and Hannes Hartenstein. 2023. On Extend-Only Directed Posets and Derived Byzantine-Tolerant Replicated Data Types. In *Proceedings of the 10th Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC '23)*. ACM, New York. https://doi.org/10.1145/3578358.3591333

[18] Brent Byunghoon Kang, R. Wilensky, and J. Kubiatowicz. 2003. The Hash History Approach for Reconciling Mutual Inconsistency. In *23rd International Conference on Distributed Computing Systems, 2003. Proceedings.* https://doi.org/10.1109/ICDCS.2003.1203518

[19] Martin Kleppmann. 2015. A Critique of the CAP Theorem. https://doi.org/10.48550/arXiv.1509.05393 arXiv:1509.05393 [cs]

[20] Martin Kleppmann. 2022. Making CRDTs Byzantine Fault Tolerant. In *Proceedings of the 9th Workshop on Principles and Practice of Consistency for Distributed Data* (Rennes, France) *(PaPoC '22)*. ACM, New York. https://doi.org/10.1145/3517209.3524042

[21] Martin Kleppmann and Heidi Howard. 2020. Byzantine Eventual Consistency and the Fundamental Limits of Peer-to-Peer Databases. *arXiv:2012.00472 [cs]* (Dec. 2020). arXiv:2012.00472 [cs]

[22] Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, Natacha Crooks, and Joseph M. Hellerstein. 2022. Keep CALM and CRDT On. *Proceedings of the VLDB Endowment* 16, 4 (Dec. 2022). https://doi.org/10.14778/3574245.3574268

[23] Leslie Lamport. 2019. *Time, Clocks, and the Ordering of Events in a Distributed System.* ACM. https://doi.org/10.1145/3335772.3335934

[24] Friedemann Mattern. 1988. Virtual Time and Global States of Distributed Systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms* (Chateau de Bonas, France). Elsevier Science Publishers B. V. (North-Holland). http://vs.inf.ethz.ch/publ/papers/VirtTimeGlobStates.pdf

[25] Andrew Miller, Michael Hicks, Jonathan Katz, and Elaine Shi. 2014. Authenticated Data Structures, Generically. *ACM SIGPLAN Notices* 49, 1 (Jan. 2014). https://doi.org/10.1145/2578855.2535851

[26] K. Mori. 1993. Autonomous Decentralized Systems: Concept, Data Field Architecture and Future Trends. In *Proceedings ISAD 93: International Symposium on Autonomous Decentralized Systems.* https://doi.org/10.1109/ISADS.1993.262725

[27] Hector Sanjuan, Samuli Poyhtari, Pedro Teixeira, and Ioannis Psaras. 2020. Merkle-CRDTs: Merkle-DAGs Meet CRDTs. https://doi.org/10.48550/arXiv.2004.00107 arXiv:2004.00107 [cs]

[28] Reinhard Schwarz and Friedemann Mattern. 1994. Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. *Distributed Computing* 7, 3 (March 1994). https://doi.org/10.1007/BF02277859

[29] 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.), Vol. 6976. Springer, Heidelberg. https://doi.org/10.1007/978-3-642-24550-3_29

[30] Berty Technologies. 2023. Wesh Network. https://wesh.network/

[31] The Matrix.org Foundation CIC. 2023. *Matrix Specification v1.9.* Technical Report. https://spec.matrix.org/v1.9/

[32] Michael Toomim, Greg Little, Rafie Walker, Bryn Bellomy, and Seph Gentle. 2023. *Braid-HTTP: Synchronization for HTTP.* Internet Draft draft-toomim-httpbis-braid-http-04. Internet Engineering Task Force. https://datatracker.ietf.org/doc/draft-toomim-httpbis-braid-http

[33] Matthew Weidner, Huairui Qi, Maxime Kjaer, Ria Pradeep, Benito Geordie, Yicheng Zhang, Gregory Schare, Xuan Tang, Sicheng Xing, and Heather Miller. 2023. Collabs: A Flexible and Performant CRDT Collaboration Framework. https://doi.org/10.48550/arXiv.2212.02618 arXiv:2212.02618 [cs]

[34] Gene T.J. Wuu and Arthur J. Bernstein. 1984. Efficient Solutions to the Replicated Log and Dictionary Problems. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing (PODC '84)*. ACM, New York. https://doi.org/10.1145/800222.806750

[35] Elena Yanakieva, Michael Youssef, Ahmad Hussein Rezae, and Annette Bieniusa. 2021. On the Impossibility of Confidentiality, Integrity and Accessibility in Highly-Available File Systems. In *Networked Systems (Lecture Notes in Computer Science)*, Karima Echihabi and Roland Meyer (Eds.). Springer International Publishing, Cham. https://doi.org/10.1007/978-3-030-91014-3_1