Decentralized Collaborative Version Control

BULAT NASRULIN, Delft University of Technology, Netherlands
JOHAN POUWELSE, Delft University of Technology, Netherlands

Decentralized systems offer alternatives to Big Tech. However, maintaining availability and correctness despite faults and manipulations in decentralized settings is challenging. In this paper, we introduce a collaborative model that is capable of exposing all observable lying, all cheating, and all faults, while only requiring merely unreliable message exchange. Our model is based on conflicting operations on arbitrary data, set reconciliation, and conflict resolution strategies to deal with branches. It is sufficiently general to support applications like Wikipedia, Github, and Datahub in a non-profit, collaborative, and decentralized form. Our protocol guarantees strong convergence despite any Byzantine nodes. We exhibit four conflict resolution strategies that cover the spectrum of possible use cases. A remarkable property of our model is that two honest nodes are guaranteed to converge despite an arbitrary-large number of faulty nodes.

CCS Concepts: • **Computer systems organization → Fault-tolerant network topologies**; • **Information systems → Collaborative and social computing systems and tools**.

Additional Key Words and Phrases: Accountability, Conflict Resolution, Decentralized Collaboration, Byzantine Resilience

1 INTRODUCTION

The total market capitalisation in 2021 of the five biggest tech companies is 8.9 trillion dollar. This concentration of capital and market dominance is unprecedented. Large-scale collaborative applications offer an alternative to the services of Big Tech companies. The collaborative open model has various degrees of decentralization, which stands in stark contrast to Big Tech services which are hosted, governed, and exclusively owned by a central party. The collaborative model is still immature and lacks many theoretical and practical foundations. Nonetheless, this model is a promising direction that provides a defense against monopoly formation in digital markets with intrinsic winner-take-all dynamics.

We present theoretically sound grounding for a single primitive for building large-scale collaborative versioning systems. Our work finds a direct application in existing collaborative applications such as Wikipedia, Github, and Datahub [2]. We designed the primitive around data versioning and conflicts. Cheap storage enables the preservation of all project contributions such as information, code, and data. We also preserve all data versions and conflicting operations. We show that revealing all data versions helps in backtracking, spotlighting errors, and addressing dissensus.

Our method provides accountability, i.e., faulty nodes are eventually identified, and exposed to correct nodes. We formulate three simple requirements to achieve accountability despite Byzantine nodes. Any failure to follow the requirements will expose the faulty node. The exposure serves as irrefutable evidence with guaranteed protection from false accusations.

All operations are preserved in a shared tamper-evident grow-only set. Our model provides strong eventual consistency [16], in which correct nodes converge to the state despite any number of Byzantine failures. Any data conflict is resolved with one of the four proposed conflict resolution functions. Remarkably, two honest nodes can successfully collaborate in a network with otherwise 99% fraudulent participants. In summary, this paper makes the following contributions:

- We present a universal data model for large-scale collaborative systems (Section 2). We list three requirements for accountable collaboration.
- We show a protocol that achieves convergence through set reconciliation despite any Byzantine nodes (Section 4).

- Finally, we propose four conflict resolution policies for different application. (Section 5).

2 PRELIMINARIES AND SYSTEM MODEL

We consider an asynchronous network of nodes. Nodes send messages to each other through pairwise network links. Messages might be delayed or dropped; however, after a sufficient number of retransmissions they are eventually delivered. The network can partition but eventually recovers.

Each node runs the same protocol, which describes how to execute and validate network messages. Each node in the network is either correct or faulty. We later specify three requirements for correct nodes. A faulty node might deviate from these requirements in arbitrary ways. The number of faulty nodes at any time is limited by f.

Each node has a unique private key that it uses for digital signatures. When a node creates a message, it attaches it's public key and signs the message. The corresponding public key uniquely identifies the creator of the received message. This assumption is common in modern decentralized applications, such as Bitcoin, IPFS.

We make an assumption on the node discovery and how the network topology is formed. Without a reliable overlay, malicious nodes might censor any communication in the system. It should guarantee that any correct node will eventually be connected to at least one honest node. This assumption is required to tackle eclipse attacks [17] and communicate in presence of Byzantine nodes. For example, a robust overlay can be achieved with Brahms [4]. We say that each correct node maintains at all times connections to a reasonable threshold of non-faulty nodes. Correct nodes do not maintain connections to observably faulty nodes.

2.1 Consistency Goals

We base our model on strong eventual consistency properties [16]. The key idea of eventual consistency is to guarantee the eventual delivery of all messages and convergence to a common state. The nodes converge to the same state when they have received the same set of messages. The goals of our eventual consistency model are as follows:

Eventual delivery. If a correct node applies an operation, then all correct nodes will eventually apply that operation.

Strong Convergence. Any two correct nodes that have applied the same set of messages are in the same state.

Causal Consistency. If a correct node applies message o_1 before applying message o_2, then all correct nodes apply o_1 before o_2.

2.2 Accountability Goals

We strengthen our model by making it *censorship-evident*. Censorship happens when a faulty node attempts to trick a correct node by hiding messages. We address this challenge by designing a model that forces nodes to reveal the truth or risk being exposed and eventually excluded from the system.

We address two classes of faulty behavior: observable omission and commission failures. Commission failure happens when a node creates a message that a correct node would not produce. However, this failure would be observed only when incorrect messages were received by the correct node. We say a node a is *exposed* as faulty by node b if node b receives an incorrect message (or set of messages) created by node a.

Omission failure happens when a node fails to produce a message when required. For example, a Byzantine node might pretend to crash and refuse to provide service to other nodes. We call such nodes *ignorant*. The difference between an ignorant and crashed node is that the correct node will eventually restore the connection and provide any requested service. We say that node a is *suspected* of being faulty by node b if node a if fails to

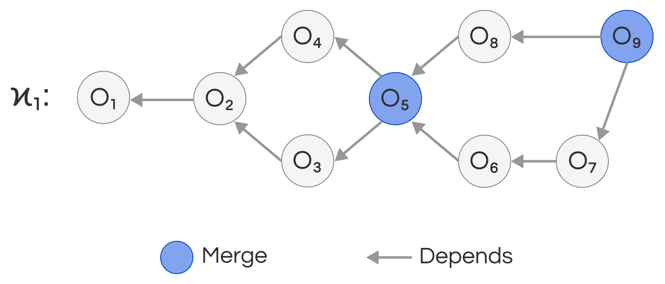

Fig. 1. The example graph of multi-version data with conflicts and merges on one state key κ_1. Nodes in the graph represent operations, and edges are 'depends-on' relationships. o_6 and o_7 operate on a different branch.

send a message when requested by b. As soon as a provides the requested message, it will no longer be suspected. We say that any protocol is accountable if it provides the properties as follows:

- **Correct Suspicion**. No correct node is suspected forever.
- **Correct Exposure**. No correct node is ever exposed.
- **Unanimous Exposure**. If a correct node exposes a faulty node, this faulty node is eventually exposed by every correct node.
- **Unanimous Suspicion**. Ignorant nodes are eventually suspected by every correct node.

3 MUTLI-VERSION COLLABORATION MODEL

In this section we present a model that captures all the collaboration of a decentralized application in one simple data structure. We formulate three requirements that any correct node should follow.

Figure 1 illustrates the model of data versioning used in this paper. We use this figure as a running example throughout the paper, referring to it as the example graph.

Our model is based on nodes, messages, operations, and state. Network nodes collaborate to track any changes in the application data. The application data is captured as a global system state. The global state can be divided into a set of fragments $\mathcal{K}$. One fragment $\kappa \in \mathcal{K}$ is a *state key*. Examples of state keys are pages in Wikipedia or files in a git repository. The state changes through operations. Nodes exchange messages that contain operations. Upon receiving a message, nodes locally execute all operations contained in the message. Operations universally represent any change to the application data, e.g., edits on wiki pages, new source code in Github repositories, or changing a dataset in Datahub.

3.1 Causally Consistent Data

We say that operation $o_i = (\phi_i, \kappa, \breve{O}_i)$ is a function ϕ_i executed in the context of $\breve{O}_i$ and state key κ. The context of operation $\breve{O}_i$ is the set of previous operations that should be execute before o_i. Operations $\{\breve{o} \in \breve{O}_i\}$ are direct *predecessors* to the operation o_i. Operation o_i *depends-on* its predecessors, i.e., $\forall \breve{o} \in \breve{O} \mid o_i -> \breve{o}$, or o_i is a *successor* to $\breve{O}$. Note that the 'depends on' relationship is the reverse of Lamport's 'happened-before'[12]. Depends-on relationships are a strict partial order and are transitive, irreflexive and antisymmetric. All predecessors operate on the same key κ.

Naturally, the operations form a directed acyclic *version graph*, in which a directed edge captures the 'depends-on' relationship, as shown in Figure 1. The first operation, o_1 does not have any predecessors. Any operation mutates the state by applying its corresponding function. Without loss of generality, lets assume that initial state of κ is some string s_0. Then operation o_1 adds new characters to the string, creating a version s_1. The operation o_2 revert changes made by o_1 and creates version s_2. Despite, the fact that s_2 is equal to s_0 we preserve these versions separately. Later, operations o_3 and o_4 concurrently change the string, creating s_3 and s_4 respectively. Finally, the state version s_5 is created when a merge function is applied, combining both s_3 and s_4.

Let O_a be the operations locally known to the node a. Terminal operations are locally known operations such that:

$$terminal(O_a) = \{o_i \in O_a | \nexists o_k \in O_a | o_k - > o_i\}$$

When a node creates an operation, it specifies direct predecessors for the operation. Transitivity property allows capturing complex relationships with only a few direct predecessors. For example, operation o_5 has only two direct predecessors, i.e. $\breve{O}_5 = \{o_3, o_4\}$, but depends on the set of operations $\{o_1, o_2, o_3, o_4\}$. We require that each operation preserves the causal order of its state key. Therefore, any operation created by a correct node must depend on the local terminal operations. In the example graph, o_9 is a terminal operation, and further operations should depend on o_9.

Requirement 1. *Any operation o_i on a key κ created by a correct node a depends on all terminal operations on a key κ known to a.*

Requirement 1 means that the operations should depend on all operations created by the node and the operations received from other nodes. For example, the operation o_5 created by a node a directly depends on the operation o_3 created by the node a and an the operation o_4 created by the node b. A violation of this requirement would indicate that the node operates on a different protocol. For instance, when node a creates operation o_5 and excludes o_4 from $\breve{O}_5$. Another example is when the Byzantine node creates a fork by creating two operations that do not depend on each other.

3.2 Conflicts and Merges

In this subsection, we present a universal semantics that captures any divergence between nodes. A divergence in data can happen in several cases. First, divergence might naturally occur because of weaker consistency and network partitioning [7]. Two correct nodes might simply not know each others operations. Second, data divergence might happen due to Byzantine nodes, that deliberately create divergent view in the network by breaking the requirement 1.

We model all the above-mentioned situations with conflicts in the version graph. A *conflict* in a version graph corresponds to the situation where two operations do not depend on each other, i.e. two operations $o_i = (\phi_i, \kappa, \breve{O}_i), o_j = (\phi_j, \kappa, \breve{O}_j)$ such that $o_i \not\rightsquigarrow o_j \wedge o_j \not\rightsquigarrow o_i$. In the example graph (Figure 1), the operations o_3 and o_4 happen simultaneously without linking each other. Operation o_5 merges two states and resolves the conflict.

We define *merge* as an operation $o_i = (\phi_i, \kappa, \breve{O}_i)$ such that $|\breve{O}_i| > 1$. Merge operations take multiple conflicting operations and output one unambiguous state. We recognize that there is no one-size-fits-all solution. Some conflicts might be automatically resolved while others require manual input. However, a key requirement for strong convergence is a *uniform conflict resolution policy*; i.e., a policy that is recognized by all correct nodes in the network. We list possible approaches to such policies in section 5.

3.3 Validity of Operations

We say the state s_i is *valid* if it satisfies the application-defined validity. The validity is a predicate over the state, i.e., a function $V(s_i)$ that takes a state created after operation o_i and returns either true or false. We say that operation o_i is valid if the produced state s_i is valid. Note that validity is defined in the causal context, i.e., with respect to previous causal operations, not globally.

Requirement 2. *Correct nodes create only valid operations.*

By following the requirement 2, any correct node would reject any invalid operation. Any invalid operation is rejected locally, and no correct node creates and signs the message with an invalid operation. The message with invalid operation serves as an evidence of a fault.

3.4 Data Dissemination

Requirement 1 allows preserving all operations in a grow-only set. We define one more requirement to achieve convergence to a common set of operations. We model the data dissemination as a sequence of *requests*. All nodes periodically send each other requests for all known operations. We require that correct nodes respond fully to all requests.

Requirement 3. *A correct node responds to all requests and reveals all known operations to other nodes.*

4 GUARANTEED FAULT EXPOSURE FOR DATA DISSEMINATION

We now introduce a multi-version data dissemination protocol that is capable of exposing censorship and faults. Using the above-mentioned requirements, we show that the protocol can guarantee detection of tampering for any data type, any data operation, and unbounded group size. Faulty nodes cannot stop the convergence process, and every correct message will eventually progress.

Our protocol is based on the periodical pairwise reconciliation between nodes. A pairwise reconciliation [8] ensures that two nodes exchange missing operations and end up with the same operation set. The reconciliation protocol works as a sequence of requests. Each node sends a request to the other node. If the requestee does not respond before a timeout, the requester suspects the requestee node. In practice, the requester might resend the request multiple times and only later suspect the node. The correct nodes preserve all pending requests. A node might fetch pending requests after a partitioning or a crash.

Let $\bar{O}_a^t$ be the set of operations signed by node a at a time t and shared in the network. For the correct node, the two sets are equal, $O_a^t = \bar{O}_a^t$. However, a faulty node might omit certain operations and reveal different versions to other nodes in the network.

Two operation sets, O_i and O_j are *inconsistent* with each other if both sets have at least one distinct operation:

$$\mathit{inconsistent}(O_i, O_j) = O_i \setminus O_j \neq \emptyset \land O_i \setminus O_j \neq \emptyset \tag{1}$$

A node a is observed and exposed as faulty by correct node b, if correct node b receives messages containing evidence of breaking one of the three requirements. When node b observes two conflicting operations signed by node a (violation of proposition 1), or an invalid operation signed by a (violation of proposition 2), or two responses to requests (or operation and request) that contain inconsistent operation sets (violation of proposition 3):

$$\begin{aligned} \mathit{exposed}(a) = {} & \exists o_i \in O_a | \mathit{invalid}(o_i) \\ & \lor \exists \tau \neq t | \mathit{inconsistent}(\bar{O}_a^t, \bar{O}_a^\tau) \\ & \lor \exists o_i \in O_a \land \exists t | \mathit{inconsistent}(\breve{O}_i, \bar{O}_a^t) \\ & \lor \exists o_i \in O_a \land \exists o_j \in O_a | \mathit{inconsistent}(\breve{O}_i, \breve{O}_j) \end{aligned} \tag{2}$$

We say for two nodes a and b, each having the set of operations O_a and O_b respectively, the difference is a set $D_{a-b} = O_a - O_b$. Node a is *outdated* in the eyes of node b if the last received set signed by a doesn't include all operations known to b:

$$\mathit{outdated}(a, b) = O_b \setminus \bar{O}_a \neq \emptyset. \tag{3}$$

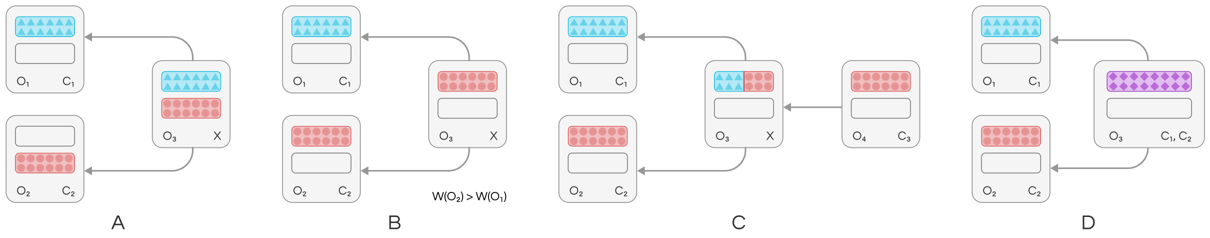

Fig. 2. Conflict resolution policies with two conflicting operations O_1 created by node C_1 and O_2 created by node C_2: A) *Aggregation.* Conflict resolved with a union of two items. B) *Competition.* One branch is selected based on a weight function. C) *Arbitration.* Auto-resolution produces ambivalent state, arbiter node C_3 resolves the conflict. D) *Compromising.* The involved parties sign an operation that resolves the conflict.

Protocol 1 Accountable Reconciliation

(1) **Create operations**. A correct node a creates an operation o_k that depends on known *terminal* (O_a) operations. Update $O_a = O_a \cup \{o_k\}$.
(2) **Request updates.** Node a periodically requests an updated set of operations from the *outdated* (as defined in predicate 3) nodes.
- Upon receiving a set request from a a correct node b sends the set of operations O_b to a.
- Upon receiving $\bar{O}_b^t$, the node a calculates D_{b-a}, and checks if node b is still outdated with the new set.
- Node a send a request for all missing operations $o \in D_{b-a}$.
- Upon receiving an operation, o_k checks validity of the operation and executes it after the operations $\breve{O}_k$.

(3) **Expose faulty nodes**. Nodes check the message creator against the predicate 'exposed' (as defined in predicate 2). The evidence of inconsistent behavior is shared with other nodes.
(4) **Suspect ignorant nodes**. Nodes share the long-pending requests. If possible, other nodes share a response to the request signed by the requestee. The requestee is no longer suspected if the requester has received a response to the request.

The data reconciliation protocol listed in protocol 1 works as follows: (1) All applied operations are stored and included in the local known set; (2) the known set is further disseminated to other nodes via requests; and (3) the missing operations are requested upon detecting inconsistencies between the observed sets.

Faulty nodes can break requirements 1-3 in arbitrary ways. For example, a malicious node x might split O_x into several inconsistent sets and send different versions to other nodes or refuse valid operations and ignore requests. We address the issues by forcing nodes to include all known operations. By continuously polling and requesting updates, nodes converge to a common set. If the node does not respond, it will eventually be suspected by all correct nodes. As a result, faulty nodes have to report consistently; otherwise, they risk exposure from all correct nodes.

Theorem 4.1. *Protocol 1 guarantees eventual exposure or suspicion of malicious nodes for $f < N$.*

Proof. A malicious node $\hat{p_k}$ has two strategies on disrupting the protocol: (a) break consistency of the protocol by sharing different versions of $\bar{O}_k$, or (b) ignore requests to censor operations and break delivery. Let's assume that two correct nodes p_i and p_j received two different versions $\bar{O}_k^i$ and $\bar{O}_k^j$ respectively. Since the protocol guarantees eventual delivery, the nodes p_i and p_j after a reconciliation end up with the operations set $O_i = O_j$, such that $\bar{O}_k^i \cup \bar{O}_k^j \subseteq O_i$. Both nodes p_i and p_j see $\hat{p_k}$ as outdated and request an update from the node $\hat{p_k}$.

If node $\hat{p_k}$ refuses to respond, it will be first suspected as faulty. When node p_i marks node $\hat{p_k}$ suspicious, it will send to p_j last state of node $\hat{p_k}$ ($\bar{O}_k^i$). The node o_j upon receiving $\bar{O}_k^i$ will expose node $\hat{p_k}$ as malicious, since $\bar{O}_k^i \setminus \bar{O}_k^j \neq \emptyset \wedge \bar{O}_k^j \setminus \bar{O}_k^i \neq \emptyset$. □

Theorem 4.2. *Protocol 1 guarantees eventual delivery for correct nodes for $f < N$.*

Proof. Let us assume that the correct node p_i is the first correct node to receive a new operation o_k. Upon receiving o the node p_i updates it's local set O_i. Since node p_i is the first node to receive the operation, all observed sets don't contain operation: $\forall j \neq i o_k \notin \bar{O}_j$, node p_i. As a result, node p_i will enforce all other nodes to update by requesting new operations sets that contain o_k. According to our assumptions, node p_i will eventually stumble upon some correct node p_j and deliver o_k. □

5 CONFLICT RESOLUTION POLICIES

Large-scale collaborative systems produce incompatible operations on the same data and cause a fork. This section introduces four strategies (Figure 2) for conflict resolution, both manual, and fully automated.

Aggregation. Aggregation (Figure 2A) might be appropriate for some collaborative systems; it simply produces a state which includes state changes from both branches. This resolution is possible if two operations work on separable data items, such as different letters, words in a string, or different items in a set. A merge function applies immediately when a conflict is detected. Every correct node creates the same merge operation. This resolution guarantees strong convergence despite Byzantine nodes.

Competition (Figure 2B) is also fully automatic and uses scoring to select only one of the operations. The scoring function determines which operations obtain the highest score, applies these operations and ignores other forks. For instance, within a Wikipedia-like system, the scoring function may be the user's reputation proposing an operation. The 'three revert rule' in Wikipedia gives an abuse penalty to users who abuse the revert operation.

Arbitration. Simple concatenation might result in a semantically ambiguous state (Figure 2C). One natural way to resolve any conflict is to wait for a manual resolution from a dedicated node. This method is commonly seen in big collaborative projects such as Wikipedia or large Github repositories. Repository maintainers or page administrators merge and resolve conflicts. Typically, the arbitration requires manual input. However, one can imagine an automated arbiter that resolves the conflict randomly or according to some procedure. The key benefit that our model provides for this resolution is the transparency of the arbitration process. The biases of any arbitration are visible to all correct nodes.

How do you select an arbiter? In general, any leader selection algorithm might apply. In permissioned settings, the application defines global authorities that act as arbitrary for resolving any conflict. For example, in PeerReview [9] such arbiters might be witness nodes preassigned via a hash function. In federated settings, nodes can predefine their trust set, like in Stellar [14] or Mastadon [15]. The arbiter is a node at the intersection of these sets.

Compromise is a resolution based on the compromised state, to which nodes come via an agreement (Figure 2D). A compromise is achieved through multiple rounds of negotiations between the involved nodes. A conflict is finalized if the resolution achieves some threshold on votes. This strategy is widely used in DAOs [10].

6 RELATED WORK

Our work is one of the first to propose a model which could, in principle, create non-profit versions of Big Tech platforms. Large-scale collaborative systems are studied by both the database community and distributed systems researchers. Most elements of collaborative systems are approached in isolation, such as CRDTs [16], gossip protocols [5], byzantine faults [9, 11], version management [18], tamper-evident storage [1]. We extend their work by showing a strong foundation for any decentralized application.

Eventually, consistent storage is commonly used in collaborative settings. Practical systems [7] priorities availability over strong consistency for performance [13]. The concept later matured into CRDTs [16], which showed that strong convergence can be achieved in collaborative settings. Despite its wide applicability in multi-writer internet services, e.g., Figma, Riak, this concept is rarely discussed in fully decentralized byzantine settings [3, 11].

Versioning is widely discussed in databases, which explores the tradeoffs between space-saving and reconstruction time of different versions of state [6, 18, 19].

7 CONCLUSION

In this paper, we presented a design for decentralized collaborative applications. We show that strong eventual consistency can be applied to decentralised settings despite a supermajority of Byzantine nodes. The key idea that enabled state convergence is accountability. Nodes reveal the full context of the application and isolate any node that attempts to censor it. Our accountability protocol guarantees both correctness and unanimity.

The presented model universally captures all interactions in the network through operations. All operations are kept in an append-only and tamper-evident data structure. Any inconsistency is eventually detected, and the correct nodes converge to a common state via four merge strategies.

REFERENCES

[1] Juan Benet. 2014. Ipfs-content addressed, versioned, p2p file system. *arXiv preprint arXiv:1407.3561* (2014).

[2] Anant Bhardwaj et al. 2014. Datahub: Collaborative data science & dataset version management at scale. *arXiv preprint arXiv:1409.0798* (2014).

[3] Loïck Bonniot et al. 2020. Pnyxdb: a lightweight leaderless democratic byzantine fault tolerant replicated datastore. In *SRDS*. IEEE, 155–164.

[4] Edward Bortnikov et al. 2009. Brahms: Byzantine resilient random membership sampling. *Computer Networks* 53, 13 (2009), 2340–2359.

[5] Stephen Boyd et al. 2006. Randomized gossip algorithms. *IEEE transactions on information theory* 52, 6 (2006), 2508–2530.

[6] Natacha Crooks et al. 2016. Tardis: A branch-and-merge approach to weak consistency. In *ICMD*. 1615–1628.

[7] Giuseppe DeCandia et al. 2007. Dynamo: Amazon's highly available key-value store. *ACM SIGOPS operating systems review* 41, 6 (2007), 205–220.

[8] David Eppstein, Michael T Goodrich, Frank Uyeda, and George Varghese. 2011. What's the difference? Efficient set reconciliation without prior context. *ACM SIGCOMM Computer Communication Review* 41, 4 (2011), 218–229.

[9] Andreas Haeberlen et al. 2007. PeerReview: Practical accountability for distributed systems. *ACM SIGOPS operating systems review* 41, 6 (2007), 175–188.

[10] Christoph Jentzsch. 2016. Decentralized autonomous organization to automate governance. *White paper, November* (2016).

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

[12] Leslie Lamport. 2019. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*. 179–196.

[13] Wyatt Lloyd et al. 2011. Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS. In *SOSP*. 401–416.

[14] Marta Lokhava et al. 2019. Fast and secure global payments with Stellar. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 80–96.

[15] Aravindh Raman et al. 2019. Challenges in the decentralised web: The mastodon case. In *Proceedings of the Internet Measurement Conference*. 217–229.

[16] Marc Shapiro et al. 2011. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems*. Springer, 386–400.

[17] Atul Singh et al. 2006. Eclipse attacks on overlay networks: Threats and defenses. In *In IEEE INFOCOM*. Citeseer.

[18] Sheng Wang et al. 2018. Forkbase: an efficient storage engine for blockchain and forkable applications. *VLDB* 11, 10 (2018), 1137–1150.

[19] Chenggang Wu et al. 2019. Anna: A kvs for any scale. *IEEE Transactions on Knowledge and Data Engineering* (2019).