Making CRDTs Byzantine Fault Tolerant

Martin Kleppmann
martin@kleppmann.com
University of Cambridge
Cambridge, UK

Abstract

It is often claimed that Conflict-free Replicated Data Types (CRDTs) ensure consistency of replicated data in peer-to-peer systems. However, peer-to-peer systems usually consist of untrusted nodes that may deviate from the specified protocol (i.e. exhibit Byzantine faults), and most existing CRDT algorithms cannot guarantee consistency in the presence of such faults. This paper shows how to adapt existing non-Byzantine CRDT algorithms and make them Byzantine fault-tolerant. The proposed scheme can tolerate any number of Byzantine nodes (making it immune to Sybil attacks), guarantees Strong Eventual Consistency, and requires only modest changes to existing CRDT algorithms.

CCS Concepts: • **Computer systems organization** → **Peer-to-peer architectures**; • **Security and privacy** → *Distributed systems security*.

Keywords: Byzantine fault tolerance, CRDTs, optimistic replication, eventual consistency

1 Introduction

A key characteristic of many peer-to-peer (P2P) systems is that peers are not under the control of a single authority [9]; indeed, many P2P applications are *open, trustless* systems in which anybody on the Internet can run a node and join the system. As the operators of nodes cannot be trusted, it must be assumed that some nodes may not correctly follow the specified protocol of the network: be it due to bugs in implementations, hardware glitches [15], deliberate attempts to gain an advantage over other nodes, or sheer vandalism, a P2P system cannot rely on nodes always behaving the way that the designers of the system intended. Nodes may send false, malformed, or contradictory messages to other nodes, and they may try to actively undermine any guarantees the system aims to provide. As P2P systems grow to larger scale, the probability of such problematic behaviour increases.

When a node deviates from the specified protocol, we call it *Byzantine-faulty* (or simply *Byzantine*), regardless of whether the deviation is by accident or by malice [20]. Byzantine behaviour is not always detectable by other nodes, since a Byzantine node may attempt to hide its protocol violations. One approach would be to try to identify Byzantine nodes and exclude them from the system, but this is unlikely to be effective if the banned node can simply rejoin the system under a different identity, and the system would still have to somehow repair the damage already done by the Byzantine node. A more robust approach is to *tolerate* Byzantine faults: that is, to ensure that the system can meet its advertised guarantees even when some of its nodes are Byzantine-faulty.

Conflict-free Replicated Data Types (CRDTs) [34] are often presented as an approach for providing consistency of replicated data in peer-to-peer systems [27, 36, 38], because they do not require a central server for concurrency control. However, despite ostensibly targeting P2P systems, the vast majority of CRDT algorithms actually do not tolerate Byzantine faults, since they assume that all participating nodes correctly follow the protocol. If such algorithms are deployed in a system with Byzantine nodes, the algorithm cannot guarantee the consistency properties that CRDTs are supposed to provide, and nodes may end up with permanently inconsistent replicas of the shared data.

In some circumstances, the lack of Byzantine fault tolerance can be justified by restricting CRDT-based collaboration to small, trusted groups of nodes: for example, in a collaborative editor, the set of users who are authorised to edit a document may be limited to immediate colleagues, who may trust each other to run the CRDT algorithm correctly. However, as the range of collaborators is widened to public settings such as wikis [26, 37, 38], and as CRDT designers seek to enable collaboration at “massive scale” [3, 23, 38], blindly trusting collaborators to do everything correctly seems like an increasingly dangerous assumption.

How difficult is it then to provide Byzantine fault tolerance (BFT) for CRDTs? We might take inspiration from Byzantine consensus systems, which have received much attention due to their use in blockchains. Experience has shown that providing BFT requires consensus algorithms that differ fundamentally from their non-Byzantine counterparts and are significantly more expensive [6]. Moreover, Byzantine consensus is only possible if we assume that less than one third of the nodes in the system are faulty [13]. This assumption is problematic for P2P systems: to prevent many adversary-controlled nodes joining the network and overwhelming it (a Sybil attack [12]), they must either exercise centralised control over which nodes are allowed to join the network, or employ expensive Sybil countermeasures such as proof-of-work [25].

This paper shows that the situation with BFT CRDTs is very different from BFT consensus: it is possible to guarantee the standard CRDT consistency properties even in systems in which *arbitrarily many* nodes are Byzantine, e.g. where the Byzantine nodes outnumber the correct nodes. This makes the algorithms immune to Sybil attacks, allowing them to be deployed in open P2P systems that anybody can join, without requiring proof-of-work or proof of any other resource.

Moreover, making CRDTs Byzantine fault tolerant does not require a redesign of the algorithms: it is possible to *retrofit* BFT to existing CRDT algorithms with some modest tweaks, without changing the fundamental way how they work. This paper introduces the principles that are needed to make operation-based CRDT algorithms robust against Byzantine faults.

2 Background and system model

This section highlights some of the problems that CRDT algorithms encounter when faced with Byzantine faults.

2.1 System model: nodes and network links

This paper assumes a peer-to-peer system in which all nodes are equal peers. The set of nodes need not be known, and nodes may join and leave at any time. Nodes may fail by crashing, and maybe recover again after a crash; and nodes may arbitrarily deviate from the protocol (i.e. exhibit Byzantine behaviour). One node does not know whether another node is Byzantine. Nodes that are neither crashed nor Byzantine are called *correct*. There is no limit on the number of crashed or Byzantine nodes in the system.

Nodes can communicate by exchanging messages over pairwise network links. The network is not necessarily complete, i.e. it is not required for every node to be able to communicate with every other node. Nodes and network links are assumed to be asynchronous and unreliable with a fair-loss assumption, i.e. messages may be lost but are eventually received after a finite number of retries. Byzantine nodes may behave arbitrarily, including sending corrupted, contradictory, or malicious messages. However, we assume that when two correct nodes communicate with each other directly, the messages they exchange are not corrupted; this can be ensured by cryptographically authenticating messages.

2.2 Correctness of CRDTs

The standard model of correctness for CRDTs is Strong Eventual Consistency [14, 34], which requires:

Eventual delivery: An update delivered at a correct replica is eventually delivered at all correct replicas.
Convergence: Correct replicas that have delivered the same set of updates have equivalent state.
Termination: All method executions terminate.

Termination is generally easy to ensure, so we focus on the other two properties. Systems usually ensure eventual delivery by using a reliable broadcast protocol, such as a gossip protocol [21], to disseminate updates. To ensure convergence, operation-based CRDTs construct updates such that they are commutative: delivering them in any order results in the same replica state. In some cases, a causal delivery algorithm additionally ensures that when one update has a dependency on an earlier update, the earlier update is delivered before the later update on all replicas.

2.3 Eventual delivery with Byzantine nodes

In a non-Byzantine system, eventual delivery can obviously only be achieved if there is some way for every pair of correct nodes to communicate, either directly or indirectly via other correct nodes. Two nodes that are permanently partitioned from each other can never communicate and hence cannot ensure they deliver the same updates.

This idea generalises to Byzantine systems: if p and q are correct nodes, and if all of the communication paths between p and q go via Byzantine nodes, then eventual delivery cannot be guaranteed, since the Byzantine nodes may choose to block communication between p and q. To ensure eventual delivery, we therefore have to assume that any two correct nodes can communicate either directly, or indirectly via other correct (non-Byzantine) nodes.

We focus on the case where two correct nodes p and q can communicate with each other directly; the case of indirect communication via other correct nodes then consists simply of several instances of such pairwise direct communication, chained together. Eventual delivery then requires that if p has delivered some update u, then q must also eventually deliver u.

To ensure that p and q eventually deliver the same set of updates, a simple but inefficient algorithm would be: periodically, p sends q every update that it has not already received from q, and vice versa. This algorithm has the downside that many updates may be sent to nodes that have already received them from another node, wasting network bandwidth. The advantage of this algorithm is that it is robust against Byzantine nodes: regardless of whether an update originated from a Byzantine or non-Byzantine node, correct nodes will propagate it to other correct nodes.

2.3.1 Version vectors.

It would be desirable to have a more efficient protocol for nodes p and q to determine which updates they need to exchange so that, at the end, both have delivered the same set of updates (an *anti-entropy* or *reconciliation* protocol). A common algorithm for this purpose is to use *version vectors* [29]: each node sequentially numbers the updates that it generates, and so the set of updates that a node has delivered can be summarised by remembering just the highest sequence number from each node.

Unfortunately, version vectors are not safe in the presence of Byzantine nodes, as shown in Figure 1. This is because a Byzantine node may generate several distinct updates with the same sequence number, and send them to different nodes. Subsequently, when correct nodes p and q exchange version vectors, they may believe that they have delivered the same

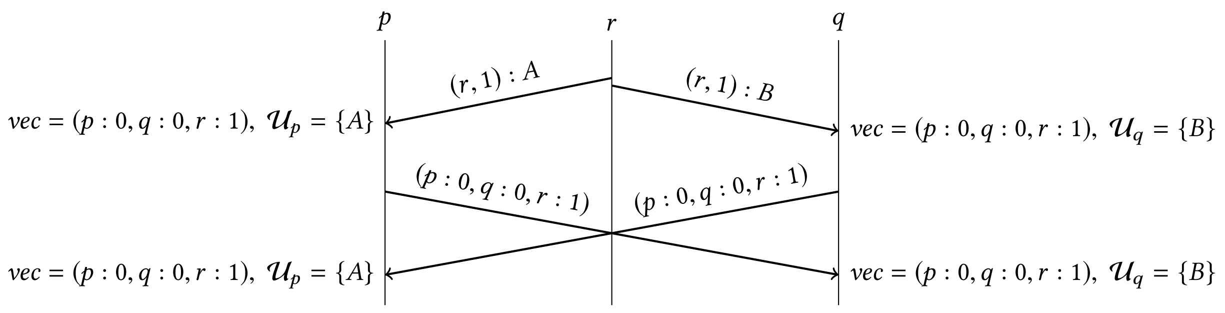
p
r
q
$(r, 1) : A$
$(r, 1) : B$
$vec = (p : 0, q : 0, r : 1),\ \mathcal{U}_p = \{A\}$
$vec = (p : 0, q : 0, r : 1),\ \mathcal{U}_q = \{B\}$
$(p : 0, q : 0, r : 1)$
$(p : 0, q : 0, r : 1)$
$vec = (p : 0, q : 0, r : 1),\ \mathcal{U}_p = \{A\}$
$vec = (p : 0, q : 0, r : 1),\ \mathcal{U}_q = \{B\}$

Figure 1. Byzantine node r sends two different updates (A and B) with the same ID $(r, 1)$ to correct nodes p and q. Now, p and q have identical version vectors, even though they have delivered different sets of updates $\mathcal{U}_p \neq \mathcal{U}_q$.

set of updates because their version vectors are identical, even though they have in fact delivered different updates.

Even if updates are signed, this type of misbehaviour by Byzantine nodes cannot be ruled out. The reuse of the same sequence number might later be detected, but this would require an additional protocol on top of version vectors.

2.4 Convergence with Byzantine nodes

Achieving the convergence property of Strong Eventual Consistency depends on the state update logic that a CRDT algorithm executes when an update is delivered. We can assume that correct nodes all execute the same update logic, but Byzantine nodes may send arbitrarily malformed updates to the correct nodes.

Some types of updates can easily be identified as malformed and rejected, for example because they do not follow the message structure expected by the CRDT algorithm. If the decision whether to reject an update is a deterministic function of the update itself, and does not depend on the state of the delivering node, then it is easy to ensure convergence, since all correct nodes will make the same decision on whether to reject a given update.

More problematic are updates that may or may not be valid, depending on the replica state of the node delivering the update. To give a few examples:

- Many CRDTs, such as Logoot [38] and Treedoc [31], assign a unique identifier to each item (e.g. each element in a sequence); the data structure does not allow multiple items with the same ID, since then an ID would be ambiguous. Say a Byzantine node generates two different updates u_1 and u_2 that create two different items with the same ID. If a node has already delivered u_1 and then subsequently delivers u_2, the update u_2 will be rejected, but a node that has not previously delivered u_1 may accept u_2. Since one node accepted u_2 and the other rejected it, those nodes fail to converge, even if we have eventual delivery.
- In many CRDTs, one operation depends on a prior one: for example, in RGA [33], an element in a sequence can only be deleted if that element was previously inserted, but a nonexistent element cannot be deleted. With non-Byzantine nodes, the metadata on updates can ensure that causally prior updates are delivered before causally later ones, but Byzantine nodes may not set this metadata correctly. Consequently, if update u_2 depends on update u_1, it could happen that some nodes deliver u_1 before u_2 and hence process both updates correctly, but other nodes may try to deliver u_2 and fail because they have not yet delivered u_1. This situation also leads to divergence.
- Some CRDTs include values in an update that must be computed in a certain way. For example, in WOOT [28] and YATA [27], an insertion operation must reference the IDs of the predecessor and successor elements, and the algorithms depend on the predecessor appearing before the successor in the sequence. The order of these elements is not apparent from the IDs alone, so the algorithm must inspect the CRDT state to check that the predecessor and successor references are valid.

3 Tolerating Byzantine faults

To achieve Byzantine fault tolerance, CRDT implementations must be able to ensure eventual delivery and convergence even in the presence of Byzantine nodes, overcoming the problems listed in Sections 2.3 and 2.4. This section lays out how CRDTs can tolerate Byzantine faults.

3.1 Constructing a hash graph

Let u be any update, encoded as a byte string. We then identify u by its hash $H(u)$, where H is a cryptographic hash function such as SHA-256 or SHA-3. We assume that H is collision-resistant, i.e. that it is computationally infeasible to find distinct x and y such that $H(x) = H(y)$. This is a standard assumption in cryptographic protocols.

Every update u contains a set of *predecessor hashes*, which are hashes of updates that causally precede u (in other words, the *causal dependencies* of u). An update may have an empty set of predecessors if it is the first update to happen on a particular node. In a system with no concurrency, each update after the first contains one predecessor, namely the hash of the update that occurred immediately before. When updates are generated concurrently and then merged, the next update contains multiple predecessors.

Dependencies that can be reached transitively via other dependencies are not included in the set of predecessor hashes, which ensures that this set remains small, even in systems with a large number of updates. We define the *heads* to be the set of updates that are currently not a dependency of any other update.

Updates and predecessor hashes form a directed acyclic graph (DAG). This graph resembles a Git commit history, with one difference: because merges of concurrent updates happen automatically in a CRDT, there is no concept of a "merge commit". Instead, after concurrent updates have occurred, we simply have a DAG with multiple heads; the next update that causally depends on those heads has multiple predecessor hashes. The graph is essentially the Hasse diagram of the partial order representing the causality relation among the updates.

3.2 Ensuring eventual delivery

Using cryptographic hashes of updates has several appealing properties. One is that if two nodes p and q exchange the hashes of their current heads, and find them to be identical, then they can be sure that the set of updates they have observed is also identical, because the hashes of the heads indirectly cover all updates. If the heads of p and q are mismatched, the nodes can run a graph traversal algorithm to determine which parts of the graph they have in common, and send each other those parts of the graph that the other node is lacking.

Our prior work [18] presents optimised algorithms for reconciling two nodes' sets of updates via hash graphs. This process can be very efficient because the number of heads is typically very small: for example, with up to three heads (i.e. three concurrently generated updates) and 256-bit hashes, the entire set of updates delivered by a node can be summarised in less than 100 bytes, regardless of how many updates have occurred or how many nodes have been involved.

Moreover, causally ordered delivery is easy to achieve with this hash graph: when a node receives new updates from another node, it simply ensures that dependencies (identified by predecessor hashes) are delivered before the updates that depend on them. A Byzantine node may generate an update with arbitrary predecessor hashes, including hashes that do not resolve to any update. This is not a problem: it simply results in the update with the missing dependency never being delivered by correct nodes.

Byzantine nodes may add arbitrary vertices and edges to the hash graph, including, for example, updates that seem to have occurred far in the past. However – assuming they cannot generate hash collisions – it is not possible for Byzantine nodes to do anything that would prevent correct nodes from delivering the same set of updates as they communicate. A proof of this property appears in prior work [18]. Byzantine nodes may attempt to cause a performance degradation by generating a large number of concurrent updates, and hence a large number of heads, or updates with a large number of predecessor hashes, but these updates will not affect the correctness of the algorithm.

Hence, we can guarantee eventual delivery, requiring only the unavoidable (Section 2.3) assumption that every correct node can eventually communicate with every other correct node, either directly or indirectly via other correct nodes.

3.3 Unique IDs

As highlighted in Section 2.4, many CRDTs require operations to have unique IDs (sometimes known as *dots* [1]). Most CRDTs generate such IDs by giving each node or replica a unique identifier, and combining that identifier with a per-node counter or sequence number. This method of generating IDs only works with trusted nodes, since a Byzantine node can easily generate duplicate IDs.

However, building upon the hash graph of updates, we have a simple way of generating unique IDs that is not susceptible to Byzantine misbehaviour: the ID of an operation is the hash of the update containing that operation. The ID of an operation is therefore only known after the update has been encoded as a byte string, but that is not a problem, since operation-based CRDTs prepare an operation in a side-effect-free manner and then apply it using an effector function [34]; the byte string is known once the preparation is complete, and the IDs only need to be known by the effector.

If an update contains multiple operations or requires multiple IDs, the IDs within an update can be an integer $1, 2, \ldots$ concatenated with the update hash. A Byzantine node cannot change this numbering without also changing the hash, making it impossible to generate duplicate IDs. If one operation needs to reference the ID of another operation within the same update, it can use this same integer (it cannot use the hash since the hash is not yet known at the time when the update is encoded as a byte string).

In CRDTs where the IDs need not satisfy any further properties besides uniqueness, it should be easy to switch to this scheme. If there are further requirements on IDs, such as ordering properties, further checks are needed as described in Section 3.4.

The downside of using hashes as IDs is that they require more space than other schemes. To avoid the storage cost of explicitly materialising all of the hashes, it would be possible to design a compression scheme that recomputes hashes when necessary, and store only the information necessary to perform this recomputation.

3.4 Checking validity of updates

If every possible update that a Byzantine node could generate is valid, we are done. However, if it is possible for an update to be rejected because it is invalid, ensuring convergence also requires that all correct nodes agree on whether a given update is valid or not, as discussed in Section 2.4.

The validity of an update u may depend on which updates happened previously. Let $\mathcal{U}_p$ be the set of updates that have been delivered at node p immediately before delivering u, and let $\mathcal{U}_q$ be the similar set at node q. We then need to ensure that p and q make the same decision about whether u is valid, even if $\mathcal{U}_p \neq \mathcal{U}_q$.

One way of guaranteeing a consistent validity decision is as follows: let $\mathit{before}(u) \subseteq \mathcal{U}_p$ be the set of updates that can be transitively reached through the predecessor hashes of u, and its predecessors, etc. Even if nodes p and q have delivered different sets of updates $\mathcal{U}_p \neq \mathcal{U}_q$, the subset $\mathit{before}(u)$ they compute for some update u is guaranteed to be the same.

Therefore, we can safely decide whether update u is valid by basing the decision only on the updates in $\mathit{before}(u)$, and not taking into account any updates that are not in $\mathit{before}(u)$. For example, say u references the ID of an element created in a previous update, and u is only valid if that element exists. Then we check whether the update that created this ID exists within $\mathit{before}(u)$; if so, u is valid. If the update that created this ID is not in $\mathit{before}(u)$, we reject u: even though the element may exist at the local replica, we cannot be sure that it will also exist at other replicas, and therefore it is safest to reject u. Updates generated by correct nodes are always valid under this scheme; only updates generated by Byzantine nodes may be rejected.

When the validity criterion is the existence of some element, rather than rejecting updates that reference a nonexistent element, an alternative approach is to delay delivering an update until some later time when the referenced element comes into existence. As long as the existence of elements is logically monotonic [2], i.e. elements can only be created but not destroyed (e.g. by retaining tombstones of deleted elements), this approach can also guarantee convergence. However, there is a risk that if an update by a correct node depends on an update by a Byzantine node, and the delivery of the Byzantine's update is delayed, then the correct node's update will also be delayed, so eventual delivery is no longer guaranteed.

As another example of a validity check, RGA [33] requires that insertions have an ID that is not only unique, but also obeys a total order that is consistent with causality (a Lamport timestamp [19] can be used). To check whether the ID contained in an update u is valid, we take the maximum ID appearing in any of the causal predecessors of u, and then require the ID in u to be one greater than that maximum.

3.5 Adapting existing CRDT algorithms

I conjecture that the techniques listed here – generating unique IDs based on the hashes of updates, and ensuring that all correct nodes agree on whether an update is valid – are sufficient to guarantee convergence in a Byzantine setting: that is, any update that a Byzantine node might produce will either be rejected by all correct nodes, or result in a legitimate CRDT state update on all correct nodes (in the sense that a non-Byzantine node may have generated the same update). Moreover, most operation-based CRDTs can be made Byzantine fault tolerant in this way. I leave a detailed proof of this conjecture in the context of particular CRDT algorithms for future work.

If authentication of updates is desired, i.e. if it is important to know which node generated which update, then updates can additionally be signed. However, this is not necessary for achieving Strong Eventual Consistency.

4 Related Work

The idea of referring to some data item using a cryptographic hash of its content, also known as *content addressing*, is used in many systems, including Git [10], BitTorrent [30], and IPFS [7]. Similarly, hash graphs are widely used: in Git [10], Merkle trees [24], blockchains [5], IPLD [32], and others [17].

However, there are fairly few attempts to provide Byzantine fault tolerance for CRDTs, and to my knowledge none that have the characteristics of the approach in this paper. Zhao et al. [11, 40, 41] propose a scheme that requires $3f + 1$ replicas to tolerate f Byzantine nodes (both among the servers and among the users). ASPAS [35, 39] relies on a Byzantine fault tolerant state machine replication protocol such as BFT-SMaRt [8], again requiring $3f + 1$ servers to tolerate f Byzantine faults. The $3f + 1$ assumption means these protocols cannot be deployed in open peer-to-peer systems, since they would be vulnerable to Sybil attacks [12]. In contrast, my approach makes no assumption about the number of Byzantine nodes.

Van der Linde et al. [22] have different goals in their CRDT-based system with untrusted nodes: this work is concerned, for example, with ensuring that nodes correctly report their causal dependencies, whereas my approach permits nodes to generate arbitrary causal dependencies (since they do not affect the eventual delivery or convergence properties). This approach relies on trusted servers and primarily focuses on *rational* clients (assuming clients will deviate from the protocol only if this cannot be detected). If a Byzantine replica behaves in a way that leaves cryptographic evidence of faulty behaviour, that replica can be excluded from the system, but it is unclear how the system recovers from misbehaviour that has already occurred by the time that replica is excluded.

Jacob et al. [16] suggest that it is possible for CRDTs to tolerate an arbitrary number of Byzantine nodes using hash graphs, but do not propose any particular algorithm.

Merkle Search Tree [4] is a state-based CRDT for sets that is able to tolerate any number of Byzantine nodes. It can be regarded as a state-based counterpart to the operation-based approach in this paper.

5 Conclusions and future work

This paper has shown how, in principle, operation-based CRDT algorithms can be adapted to tolerate any number of Byzantine faults while still guaranteeing Strong Eventual Consistency. I hope that it will inspire further research into Byzantine fault tolerant CRDTs.

Further work is required to demonstrate whether the techniques presented here are indeed effective in the context of particular CRDT algorithms, to prove their correctness in the face of Byzantine nodes, and to measure the performance impact of Byzantine fault tolerance.

Acknowledgments

Thank you to Heidi Howard for feedback on a draft of this paper. Martin Kleppmann is supported by a Leverhulme Trust Early Career Fellowship, the Isaac Newton Trust, Nokia Bell Labs, and crowdfunding supporters including Ably, Adrià Arcarons, Chet Corcos, Macrometa, Mintter, David Pollak, Prisma, RelationalAI, SoftwareMill, Talent Formation Network, and Adam Wiggins.

References

[1] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. 2017. Delta state replicated data types. *J. Parallel and Distrib. Comput.* 111 (Aug. 2017), 162–173. https://doi.org/10.1016/j.jpdc.2017.08.003

[2] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and William R. Marczak. 2011. Consistency Analysis in Bloom: a CALM and Collected Approach. In *5th Biennial Conference on Innovative Data Systems Research (CIDR 2011)*.

[3] Luc André, Stéphane Martin, Gérald Oster, and Claudia-Lavinia Ignat. 2013. Supporting adaptable granularity of changes for massive-scale collaborative editing. In *9th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2013)*. ICST, 50–59. https://doi.org/10.4108/icst.collaboratecom.2013.254123

[4] Alex Auvolat and François Taïani. 2019. Merkle Search Trees: Efficient State-Based CRDTs in Open Networks. In *38th Symposium on Reliable Distributed Systems (SRDS 2019)*. IEEE, 221–230. https://doi.org/10.1109/srds47363.2019.00032

[5] Leemon Baird. 2016. *The Swirlds hashgraph consensus algorithm: Fair, fast, Byzantine fault tolerance.* Technical Report TR-2016-01. Swirlds. https://www.swirlds.com/downloads/SWIRLDS-TR-2016-01.pdf

[6] Shehar Bano, Alberto Sonnino, Mustafa Al-Bassam, Sarah Azouvi, Patrick McCorry, Sarah Meiklejohn, and George Danezis. 2019. SoK: Consensus in the Age of Blockchains. In *1st ACM Conference on Advances in Financial Technologies (AFT 2019)*. ACM, 183–198. https://doi.org/10.1145/3318041.3355458

[7] Juan Benet. 2014. IPFS – Content Addressed, Versioned, P2P File System. arXiv:1407.3561 [cs.NI] https://arxiv.org/abs/1407.3561

[8] Alysson Bessani, João Sousa, and Eduardo E. P. Alchieri. 2014. State Machine Replication for the Masses with BFT-SMART. In *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2014)*. IEEE, 355–362. https://doi.org/10.1109/DSN.2014.43

[9] John F. Buford and Heather Yu. 2010. Peer-to-Peer Networking and Applications: Synopsis and Research Directions. In *Handbook of Peer-to-Peer Networking*, Xuemin Shen, Heather Yu, John Buford, and Mursalin Akon (Eds.). Springer, 3–45. https://doi.org/10.1007/978-0-387-09751-0_1

[10] Scott Chacon and Ben Straub. 2014. *Pro Git* (second ed.). Apress. https://git-scm.com/book/en/v2

[11] Hua Chai and Wenbing Zhao. 2014. Byzantine Fault Tolerance for Services with Commutative Operations. In *2014 IEEE International Conference on Services Computing (SCC 2014)*. IEEE, 219–226. https://doi.org/10.1109/SCC.2014.37

[12] John R. Douceur. 2002. The Sybil Attack. In *International Workshop on Peer-to-Peer Systems (IPTPS 2002)*. Springer, 251–260. https://doi.org/10.1007/3-540-45748-8_24

[13] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. 1988. Consensus in the Presence of Partial Synchrony. *J. ACM* 35, 2 (April 1988), 288–323. https://doi.org/10.1145/42282.42283

[14] Victor B.F. Gomes, Martin Kleppmann, Dominic P. Mulligan, and Alastair R. Beresford. 2017. Verifying strong eventual consistency in distributed systems. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (Oct. 2017). https://doi.org/10.1145/3133933

[15] Peter H. Hochschild, Paul Turner, Jeffrey C. Mogul, Rama Govindaraju, Parthasarathy Ranganathan, David E. Culler, and Amin Vahdat. 2021. Cores that don't count. In *Workshop on Hot Topics in Operating Systems (HotOS 2021)*. ACM, 9–16. https://doi.org/10.1145/3458336.3465297

[16] 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

[17] Brent Byunghoon Kang, Robert Wilensky, and John Kubiatowicz. 2003. The hash history approach for reconciling mutual inconsistency. In *23rd International Conference on Distributed Computing Systems (ICDCS 2003)*. IEEE, 670–677. https://doi.org/10.1109/ICDCS.2003.1203518

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

[19] Leslie Lamport. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

[20] Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems* 4, 3 (July 1982), 382–401. https://doi.org/10.1145/357172.357176

[21] João Leitão, José Pereira, and Luís Rodrigues. 2009. Gossip-Based Broadcast. In *Handbook of Peer-to-Peer Networking*. Springer, 831–860. https://doi.org/10.1007/978-0-387-09751-0_29

[22] Albert van der Linde, João Leitão, and Nuno Preguiça. 2020. Practical Client-side Replication: Weak Consistency Semantics for Insecure Settings. *Proceedings of the VLDB Endowment* 13, 11 (July 2020), 2590–2605. https://doi.org/10.14778/3407790.3407847

[23] Xiao Lv, Fazhi He, Weiwei Cai, and Yuan Cheng. 2016. An efficient collaborative editing algorithm supporting string-based operations. In *20th IEEE International Conference on Computer Supported Cooperative Work in Design (CSCWD 2016)*. IEEE, 45–50. https://doi.org/10.1109/cscwd.2016.7565961

[24] Ralph C. Merkle. 1987. A Digital Signature Based on a Conventional Encryption Function. In *A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology (CRYPTO 1987)*. Springer, 369–378. https://doi.org/10.1007/3-540-48184-2_32

[25] Satoshi Nakamoto. 2008. Bitcoin: A Peer-to-Peer Electronic Cash System. https://bitcoin.org/bitcoin.pdf

[26] Brice Nédelec, Pascal Molli, and Achour Mostefaoui. 2016. CRATE: Writing Stories Together with our Browsers. In *25th International World Wide Web Conference (WWW 2016)*. ACM, 231–234. https://doi.org/10.1145/2872518.2890539

[27] Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. 2016. Near Real-Time Peer-to-Peer Shared Editing on Extensible Data Types. In *19th International Conference on Supporting Group Work (GROUP 2016)*. ACM, 39–49. https://doi.org/10.1145/2957276.2957310

[28] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. 2006. Data consistency for P2P collaborative editing. In *ACM Conference on Computer Supported Cooperative Work (CSCW 2006)*. ACM, 259–268. https://doi.org/10.1145/1180875.1180916

[29] D. Stott Parker Jr., Gerald J Popek, Gerard Rudisin, Allen Stoughton, Bruce J. Walker, Evelyn Walton, Johanna M. Chow, David Edwards, Stephen Kiser, and Charles Kline. 1983. Detection of Mutual Inconsistency in Distributed Systems. *IEEE Transactions on Software Engineering* SE-9, 3 (May 1983), 240–247. https://doi.org/10.1109/tse.1983.236733

[30] Johan Pouwelse, Paweł Garbacki, Dick Epema, and Henk Sips. 2005. The BitTorrent P2P File-Sharing System: Measurements and Analysis. In *4th International Workshop on Peer-to-Peer Systems (IPTPS 2005)*. 205–216. https://doi.org/10.1007/11558989_19

[31] Nuno Preguiça, Joan Manuel Marques, Marc Shapiro, and Mihai Letia. 2009. A Commutative Replicated Data Type for Cooperative Editing. In *29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009)*. IEEE, 395–403. https://doi.org/10.1109/icdcs.2009.20

[32] Protocol Labs. [n. d.]. IPLD. https://ipld.io/

[33] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. 2011. Replicated abstract data types: Building blocks for collaborative applications. *J. Parallel and Distrib. Comput.* 71, 3 (March 2011), 354–368. https://doi.org/10.1016/j.jpdc.2010.12.006

[34] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *13th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2011)*. Springer, 386–400. https://doi.org/10.1007/978-3-642-24550-3_29

[35] Ali Shoker, Houssam Yactine, and Carlos Baquero. 2017. As Secure as Possible Eventual Consistency: Work in Progress. In *3rd International Workshop on Principles and Practice of Consistency for Distributed Data (PaPoC 2017)*. ACM, Article 5. https://doi.org/10.1145/3064889.3064895

[36] Albert van der Linde, Pedro Fouto, João Leitão, Nuno Preguiça, Santiago Castiñeira, and Annette Bieniusa. 2017. Legion: Enriching Internet Services with Peer-to-Peer Interactions. In *26th International Conference on World Wide Web (WWW 2017)*. ACM, 283–292. https://doi.org/10.1145/3038912.3052673

[37] Stéphane Weiss, Pascal Urso, and Pascal Molli. 2007. Wooki: A P2P Wiki-Based Collaborative Writing Tool. In *8th International Conference on Web Information Systems Engineering (WISE 2007)*. Springer, 503–512. https://doi.org/10.1007/978-3-540-76993-4_42

[38] Stéphane Weiss, Pascal Urso, and Pascal Molli. 2009. Logoot: A Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks. In *29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009)*. IEEE, 404–412. https://doi.org/10.1109/ICDCS.2009.75

[39] Houssam Yactine, Ali Shoker, and Georges Younes. 2021. ASPAS: As Secure as Possible Available Systems. In *21st IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems (DAIS 2021)*. Springer, 57–73. https://doi.org/10.1007/978-3-030-78198-9_4

[40] Wenbing Zhao and Mamdouh Babi. 2013. Byzantine fault tolerant collaborative editing. In *IET International Conference on Information and Communications Technologies (IETICT 2013)*. Institution of Engineering and Technology, 233–240. https://doi.org/10.1049/cp.2013.0057

[41] Wenbing Zhao, Mamdouh Babi, William Yang, Xiong Luo, Yueqin Zhu, Jack Yang, Chaomin Luo, and Mary Yang. 2016. Byzantine Fault Tolerance for Collaborative Editing with Commutative Operations. In *IEEE International Conference on Electro Information Technology (EIT 2016)*. IEEE, 246–251. https://doi.org/10.1109/eit.2016.7535248