On CRDTs in Byzantine Environments

Conflict-Freedom, Equivocation Tolerance, and the Matrix Replicated Data Type

Florian Jacob[1], Saskia Bayreuther[1], Hannes Hartenstein[1]

Abstract: Conflict-free Replicated Data Types (CRDTs) allow updates to be applied to different replicas independently and concurrently, without the need for a remote conflict resolution. Thus, they provide a building block for scalability and performance of fault-tolerant distributed systems. Currently, CRDTs are typically used in a crash fault setting for global scale, partition-tolerant, highly available databases or collaborative applications. In this paper, we explore the use of CRDTs in Byzantine environments. This exploration is inspired by the popular Matrix messaging system: as recently shown, the underlying Matrix Event Graph replicated data type represents a CRDT that can very well deal with Byzantine behavior. This "Byzantine Tolerance" is due to mechanisms inherent in CRDTs and in the hash-based directed acyclic graph (HashDAG) data structure used in Matrix. These mechanisms restrict Byzantine behavior. We, therefore, discuss Byzantine behavior in a context of CRDTs, and how the notion of Byzantine tolerance relates to equivocation. We show that a subclass of CRDTs is equivocation-tolerant, i.e., without equivocation detection, prevention or remediation, this subclass still fulfills the CRDT properties, which leads to Byzantine tolerance. We conjecture that an operation-based Byzantine-tolerant CRDT design supporting non-commutative operations needs to be based on a HashDAG data structure. We close the paper with thoughts on chances and limits of this data type.

Keywords: Dependable Distributed Protocols; Conflict-Free Replicated Data Types; Equivocation Tolerance; Byzantine Fault Model; Matrix Event Graph

1 Introduction

Conflict-free Replicated Data Types (CRDTs) are maintained by replicas that run on multiple processes of a distributed system.[2] To keep all replicas up to date about local changes to the CRDT, replicas repeatedly broadcast updates to all replicas. As the name suggests, CRDTs provide powerful properties: in particular, updates can be applied without further coordination of replicas, and recovery from network partitions can be done with ease. The corresponding property that CRDTs provide, namely Strong Eventual Consistency (SEC), ensures that correct replicas eventually converge to a consistent state, i.e., the same state, regardless of the order in which updates were received. Therefore, CRDTs are popular

[1] Karlsruhe Institute of Technology, KASTEL Institute of Information Security and Dependability, Am Fasanengarten 5, 76131 Karlsruhe, Germany

[2] For simplicity, we typically assume a one-to-one correspondence between replicas and processes.

doi:10.18420/sicherheit2022_07

for global-scale, partition-tolerant, highly-available databases or peer-to-peer collaborative applications like shopping lists and collaborative text editors.

However, previous CRDT-related work mainly focused on crash fault settings. We are interested in an understanding of CRDTs also in a Byzantine environment. In particular, we are interested in the characterization of Byzantine-tolerant CRDTs as well as in the relevance of these CRDTs. This interest is currently shared by various researchers [KH20, Au21] and also inspired by the success of the Matrix messaging system [Th21] that is based on the Matrix Event Graph replicated data type, a CRDT in a Byzantine environment [Ja21].

What does a Byzantine environment mean for CRDTs? Basically, we will call a CRDT Byzantine-tolerant when Byzantine processes cannot induce a violation of the CRDT properties. To achieve Byzantine tolerance, recent work on CRDTs in Byzantine environments have followed different paths. Some previous work makes use of classical assumptions of an honest two-thirds majority [Zh16] or introduce coordination mechanisms (e.g., coordinated Byzantine-tolerant causal-order broadcast [Au21]). Other recent publications study coordination-free, Sybil-resistant CRDTs using broadcast based on the happened-before relation as directed, acyclic graphs [KH20, Ja21].

In this paper, we take up the latter approach that does neither depend on additional coordination mechanisms nor on a particularly strengthened broadcast. As a motivating example, we look at the Matrix Event Graph replicated data type and its use of a hash-based directed acyclic graph data structure. We show that Byzantine behavior targeted to violate SEC essentially reduces to equivocation (and omission as a special case of equivocation) and under which conditions a subclass of crash-tolerant CRDTs is equivocation-tolerant in Byzantine environments. In particular, we show that, due to equivocation tolerance, all state-based and a subclass of operation-based CRDTs in the crash fault model tolerate any number of Byzantine processes. Further, we conjecture that the only non-trivial Byzantine-tolerant CRDT design is a grow-only HashDAG as it is done in the Matrix Event Graph.

Do Byzantine-tolerant CRDTs matter? As “safety does not guard against faulty clients” [Ca99, Section 3], even if Byzantine faults can be tolerated on a CRDT level, the application itself on top of a Byzantine-tolerant CRDT has to cope with the provided SEC guarantee and a potentially Byzantine behavior on application level. While this application-level Byzantine tolerance is out of scope of our paper and cannot be achieved in many cases, in this paper we like to start the discussion on chances and limits of Byzantine-tolerant CRDT deployments.

The structure of the paper is as follows: In Sect. 2, we provide the system model with its terminology and assumptions as well as the relationship between Byzantine tolerance and equivocation tolerance for CRDTs. In Sect. 3, we study the case of the Byzantine-tolerant Matrix Event Graph that powers the Matrix messaging system. In Sect. 4, we provide the general technical characterization of Byzantine-tolerant CRDTs as well as a conjecture. In Sect. 5, we take up the question on the relevance of Byzantine-tolerant CRDTs based on the technical characterization and address open issues.

2 System Model: Terminology and Assumptions

Replicas run on processes of a distributed system and serve data to a (distributed) application. It is, therefore, natural to consider the system under study on three layers: the network layer for the communication between processes, the CRDT layer as the data layer, and the application layer. The focus of this work is on the CRDT layer in the middle, in which replicas execute operations to query or update the state of the CRDT. The replicas rely on the network layer to provide broadcast in order to exchange information on their current state or state changes. In this section, we will first review the guarantees to the application layer that are provided by the replicas as well as their requirements for the network layer, for the case of a crash fault setting. Afterwards, we move to Byzantine environments and the notion of Byzantine-tolerant CRDTs.

Without coordination or conflict resolution between replicas, CRDTs ensure a notion of "conflict-freedom" formalized as *Strong Eventual Consistency* (SEC). SEC consists of the following properties [Sh11]:

Eventual Delivery: If an update is applied at some correct replica, it is eventually applied at every correct replica.
Termination: Every operation that is executed by a correct replica eventually terminates.
Strong Convergence: Correct replicas that applied the same set of updates maintain the same state.

CRDTs are either *state-based* or *operation-based* (see, e.g., [Sh11]). With state-based CRDTs, all states of the CRDT form a *semilattice*. A semilattice is a partially ordered set, and every possible CRDT state is one element of the set. Every pair of states has a least upper bound, which is also called the *join* of two states. An update is *valid* if it is part of the semilattice. If a replica r receives a valid state from another replica p, p's state can be directly applied by merging r's state with p's state using the join operation of the semilattice. To fulfill SEC, replicas repeatedly broadcast the current local state to the other replicas, which merge the received state with their local state. Thus, eventual delivery is required as a property of the underlying network layer.

Operation-based CRDTs differ from state-based CRDTs in the fact that replicas do not send their whole new state as updates but only the operation that lead to the new state. To fulfill SEC, the updates must either be applied in causal order or be commutative. In crash fault environments, CRDTs usually require an underlying causal order broadcast on the network layer to enforce the causal order [Sh11]. We follow a different approach (as it is done in the Matrix Event Graph) to enforce the causal order by making use of the happened-before relation that is used in the CRDT itself. We present the happened-before relation and the corresponding causal order in Sect. 3.

Replicas only apply *valid updates*. The validity of an update is defined by the specific CRDT when viewed on its own, e.g., in case of state-based CRDTs by the semilattice. An invalid update violates locally verifiable properties and cannot be applied to the CRDT. A *conflict* would occur when two concurrent updates that may be individually *valid* but, when viewed together, violate some invariant (cf. Sect. 4 and [Sh11, Footnote 1]). However, a CRDT guarantees that a conflict on the CRDT layer will not occur, typically under the assumption of a crash fault model.

We are now interested to analyze the conditions or restrictions under which a CRDT in the crash fault model can tolerate Byzantine behavior on the CRDT layer. A Byzantine-tolerant CRDT is defined as follows:

Byzantine-tolerant CRDT: A *Byzantine-tolerant CRDT* ensures that on correct processes the SEC guarantee is provided to the application layer by tolerating Byzantine behavior regardless of the number of Byzantine processes as long as the correct processes build a connected component on the network layer.

A CRDT in the crash fault model already provides strong means against Byzantine behavior: an invalid update according to the CRDT definition will not be accepted anyhow. Whether the CRDT guarantees are sufficient for an application is a different aspect on which we comment later. Thus, a Byzantine replica cannot attack the system with invalid updates with respect to the CRDT definition, and *Equivocation* and *Omission* remain as the only options for an attack on CRDT layer. Equivocation is the act of sending different valid updates to different recipients, where a replica should have sent the same update [Ch07]. Omission can be seen as a special case of equivocation where an update is not sent to a subset of replicas or even to all other replica. In contrast to an invalid update, which is detectable when viewed on its own, equivocated updates can only be detected globally or with both equivocated updates. Equivocated updates always originate from a Byzantine replica and do not exist in the crash fault model. Please note that in the scope of this paper, equivocated updates are not necessarily conflicting, i.e., with respect to the invariants of the technical CRDT layer.

We can, therefore, check the Byzantine tolerance of a CRDT by checking whether a CRDT is *equivocation-tolerant* as defined as follows:

Equivocation-tolerant CRDT: A CRDT is *equivocation-tolerant* if it neither needs to detect, prevent, nor remedy equivocation to ensure its provided guarantees beyond what is needed to cope with omission.

Through our assumptions, the relevant Byzantine behavior is restricted to the network and CRDT layer in form of equivocation and omission, as Byzantine behavior on the application layer cannot harm SEC and is thereby out of scope. In Sect. 4, we present a characterization of subclasses of CRDTs that are equivocation-tolerant and, thus, Byzantine tolerant.

For the network layer, we assume an asynchronous network with a static set of processes participating in the system. The processes are connected with authenticated channels and all correct processes form a connected component[1] in the communication graph, so that Byzantine processes can neither forge message senders nor block communication

[1] No fully connected mesh is required.

between two sets of correct processes. Every update that is sent over a channel can be arbitrarily delayed, but is received eventually on the other side of the channel. The requested connected component can, therefore, be built based on a Best-Effort Broadcast as, e.g., defined in [CGR11] (also cf. Appendix A). In the case of state-based CRDTs, one has to require periodic broadcasts of all correct replicas to ensure 'connectedness', i.e., liveness. Correspondingly, in the case of operation-based CRDTs, we will require eventual response to a request when a replica requests information from another replica.

A different question is which applications would work reasonably well with the guarantees provided by Byzantine-tolerant CRDTs — and whether Byzantine behavior can be dealt with on application layer. We will take up this discussion in Sect. 5. In short, one would either stick to grow-only CRDTs or one has to add (policy-based) mechanisms that are based on access control (or both). As grow-only CRDTs only support 'append' operations, they are susceptible to application-layer spamming if not prevented via access control. The Matrix Event Graph, as illustrated in the following section, represents a grow-only CRDT and serves an instant messaging application.

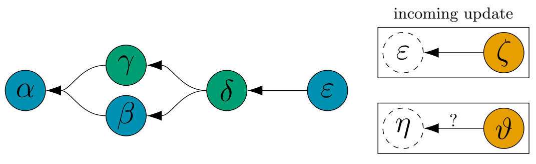
3 Matrix Event Graph

As of today, Matrix is primarily used for decentralized instant messaging, e.g., by the French public sector, by the German military forces, and as upcoming standard in the German healthcare sector [Ho21b]. As of October 2021, the public Matrix federation consists of more than 35 million users and 70 thousand servers [Ho21a].

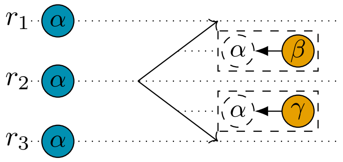
The Matrix Event Graph (MEG) represents the CRDT at the core of Matrix. It provides a grow-only history of messages to the publish/subscribe messaging application layer. The MEG is conflict-free not only in the crash fault model, but also in the Byzantine fault model [Ja21]. Therefore, the MEG serves as our prime example for Byzantine-tolerant operation-based CRDTs in this paper. From the network layer, the MEG only requires a best-effort broadcast and a connected component of all correct processes. We present a short introduction to the MEG concept, and outline why neither Byzantine equivocation nor omission can 'hurt' the MEG.

The data structure of a MEG is a directed, acyclic graph (DAG) [Ja21]. A new message e is appended to a replica of the MEG by an update operation that takes the set L of all currently known forward extremities (intuitively: messages without 'children'), adds a unique identifier w for the operation and sends the tuple (e, L, w) to all replicas (including itself). Each (correct) replica now adds the new vertex (e, w) to the DAG as well as corresponding edges from (e, w) to all 'parents' in L, provided all vertices in L exist at this replica. If not, the replica requests information of missing vertices from other replicas. For a formalization of the CRDT definition please consult Algorithm 1 in Appendix B.

The edges represent a happened-before relation as defined by [La78], and the corresponding potentially causal order provides the causal order for the CRDT. Of course, the potentially



(a) Matrix Happened-Before Relation



(b) Equivocation performed by r_2

Fig. 1: (a) MEG state of one replica with messages from itself (●) and from other replica (●). It receives two updates from a third replica (●): The update ζ can be applied immediately, since its causal parent ϵ is already known. The update ϑ cannot be applied (yet) since η is not part of the replica's state. (b) An example for an equivocation performed by replica r_2.

causal order does not refer to the actual 'real' causality of the messages, but to the potential causal order given by the selection of the set L. In the following, we simply refer to the 'causal order of the CRDT'. Fig. 1a shows a simple example of a causal history of a replica.

To be able to cope with Byzantine behavior, integrity needs to be protected: it should neither be possible to change a message's content in an undetectable manner once the message is added to the MEG, nor should it be possible to change the causal order in the MEG. For integrity protection, *content identifiers* are used based on cryptographic hashes to bind both the identity as well as the causal order of operations in a way that is locally verifiable for correct processes, but unforgeable for Byzantine processes. Let h_1 be a cryptographic hash function that provides for the content identifier $h_1(e)$ of message e. Assume that $w_1, \ldots, w_n$ are the unique identifiers of the operations that added the elements of L to the DAG. Then the content identifier (and unique identifier) for the operation that adds message e is given by $h_2(w_1, \ldots, w_n, h_1(e))$, for a cryptographic hash function h_2. Thus, content and structure of the DAG is protected using cryptographic hashes: when the content identifier for adding message β contains the hash of the content identifier for adding message α, α must have happened before β. Through these hash-based content identifiers, in analogy to a hash chain in blockchains, the MEG builds up a *Hash-based DAG* (HashDAG) data structure.

As depicted in Fig. 1a, the happened-before relation allows replicas to detect faults: If the parents of a new message are not yet known, it is not appended to the graph. Instead, other replicas can be queried for the missing operations. Through the hash-based content identifiers, replicas can verify the integrity of replies to their query even from Byzantine replicas, and will eventually receive the operation if any correct replica received it. Thus, messages with made-up parents will never be added to the graph maintained at a correct replica. As shown in Fig. 1b, a Byzantine replica (r_2) can perform equivocation: It sends an update with message β to r_1 and an update with message γ to r_2 where it should have sent the same to both r_1 and r_3. As different updates have different content identifiers, as soon as correct replicas r_1 and r_3 communicate with each other, they can reliably detect that their graphs differ by comparing content identifiers of received updates. Because the DAG only implies a partial order, the MEG can treat and merge two equivocated updates as two causally independent, concurrent updates, and consistency is restored.

4 CRDTs: Equivocation Tolerance and Byzantine Tolerance

In this section, we analyze under which conditions CRDTs in the crash-fault model are Byzantine-tolerant as defined in and under the assumptions from Sect. 2. As presented in Sect. 2, to check Byzantine tolerance, we have to check equivocation tolerance as well as handling of omissions. Equivocation itself mainly threatens the Strong Convergence property. A Byzantine replica can equivocate using two updates trying to attack the notion of which updates are the same, or the application order of updates of correct replicas. Thus, we have to address these identity and ordering aspects of updates. As we will also see below, omission faults will not threaten the property of Termination.

4.1 State-based CRDTs

Proposition 4.1. *All state-based CRDTs in the crash fault model also trivially provide equivocation tolerance in Byzantine environments.*

Sketch of proof. State-based CRDTs are based on a defined join-semilattice of all valid states. Replicas of a state-based CRDT in the crash fault model only send their current state without metadata, which means that an update is valid if and only if it is part of the semilattice, which is locally verifiable. Due to the commutativity of the join relation and the partial order of the semilattice, any two valid updates cannot conflict with each other, as both can be merged in an arbitrary order with the same result. Thus, in case a replica wants to equivocate, all the replica can do is to send different valid updates that will not conflict, by definition of the state-based CRDT in the crash fault model. Accordingly, an equivocation consisting of d differing updates can be treated as d independent updates for which omission has occurred.

Corollary 4.2. *State-based CRDTs in the crash fault model also ensure Strong Eventual Consistency for all correct replicas and are thereby Byzantine-tolerant.*

Sketch of proof. Due to Proposition 4.1, we can treat equivocation as omission, i.e., an update was not sent to all other replicas. State-based CRDTs broadcast their current state regularly to all other replicas. With the given system model, every update that is sent to another replica is received eventually by this replica. As the replicas' current state indirectly contains all updates they have received and merged before, updates not sent directly to a specific replica will eventually reach that replica indirectly via correct replicas. Thus, also Eventual Delivery and Termination are not violated.

4.2 Operation-based CRDTs

While state-based CRDTs need no further assumptions to be Byzantine-tolerant, only certain operation-based crash-tolerant CRDTs are also Byzantine-tolerant, and they need the following two additional assumptions or invariants.

Operation-based CRDTs require that non-commutative updates are applied in causal order. A CRDT provides an *inherent ordering* when all information that a correct replica needs to apply an update in correct causal order is part of the update and cannot be equivocated. In other words, the updates are either commutative or are integrity protected as it is the case for a HashDAG as outlined in Sect. 3: when inherent ordering is implemented via hash-based content identifiers as in the MEG, Byzantine attackers cannot tamper with the happened-before relation, as hashes verifiably prove the order of the updates.

Hash-based content identifiers also provide an *inherent identity* for update operations: An update has an inherent identity when all information a correct replica needs to distinguish two updates (or to decide that two updates are identical) are part of the update and cannot be equivocated. Thereby, identical updates are not applied twice when received twice.

Proposition 4.3. *Operation-based CRDTs in the crash fault model require inherent identity of updates and inherent ordering of updates to be equivocation-tolerant in any Byzantine environment.*

Sketch of proof. Operation-based CRDTs rely on update metadata, especially on content identifiers of update operations. The uniqueness of content identifiers of update operations represents an invariant whose violation by Byzantine replicas would violate Strong Convergence, and thereby SEC. Without inherent identity, a Byzantine replica can equivocate by sending two different updates with the same identifier to different replicas. Without inherent ordering, a Byzantine replica can equivocate by sending two versions of two non-commutative updates with converse causal order. In both cases, the conflicting updates would be applied and lead to an inconsistent state, without a coordination-free way for the receiving replicas to detect or remedy. With inherent identity and inherent ordering, one of any pair of conflicting updates that would violate identifier uniqueness resp. happened-before correctness is invalid, i.e., can be locally detected and rejected by correct replicas without coordination. It follows that both inherent identity as well as inherent ordering is required for operation-based CRDTs to be equivocation-tolerant.

Without periodic broadcasting of state-based CRDTs, operation-based CRDTs need to be able to handle omissions to ensure Eventual Delivery. If a happened-before relation is included in updates, then *Omission Handling* can rely on the relation to detect missing updates. Using hash-based content identifiers, those missing updates can be requested from other replicas. Alternatively, CRDTs can periodically gossip the set of all received updates. The gossiping approach can be formalized and made more efficient through the happened-before relation and hash chaining [KH20]. We note that this approach essentially uses a state-based set CRDT to synchronize all updates, benefiting from the Byzantine tolerance of all state-based CRDTs shown in Corollary 4.2.

Corollary 4.4. *Omission-handling, equivocation-tolerant operation-based CRDTs ensure Strong Eventual Consistency for all correct replicas and are thereby Byzantine-tolerant.*

Sketch of Proof. Proposition 4.3 allows to treat equivocation as omission. For CRDTs that use an omission handling mechanism (e.g., one of the approaches explained above), Byzantine replicas cannot prevent that updates they sent to at least one correct replica are eventually delivered to all correct replicas, i.e., they cannot harm the Eventual Delivery property through omission.

4.3 Uniqueness Conjecture

In Byzantine environments, a CRDT with non-commutative operations has to record the happened-before relation of updates in the data structure to locally ensure the causal order independently of the broadcast order. Ensuring that some update happened-before some other update in a locally verifiable way in the presence of Byzantine processes directly points to hash-chaining the corresponding updates with a cryptographic hash function guaranteeing preimage resistance. The happened-before relation being a partial order inherently leads to a directed, acyclic graph (DAG) of all updates. To efficiently ensure Eventual Delivery, one can employ the happened-before relationship recorded in the graph by requesting missing parent operations from other replica. In combination, these considerations lead to a grow-only HashDAG. The presented line of thought was independently followed in [KH20] as well as in [Th21, Ja21], which leads us to the following conjecture:

Conjecture. *The only Byzantine-tolerant operation-based CRDT design that supports non-commutative updates is a grow-only HashDAG.*

5 Discussion and Conclusion

We analyzed the reasons why and under which conditions a subclass of CRDTs is not only crash-tolerant, but also Byzantine-tolerant. For the analysis we made use of the notion of equivocation tolerance and its relation to conflict freedom of CRDTs. We showed that regardless of the number of Byzantine processes, this subclass can keep the characteristic traits of CRDTs, like efficiency, low coordination effort, and Strong Eventual Consistency. On the network layer, only a best effort broadcast, i.e., eventual delivery, is required.

We now like to take up the question of "do Byzantine-tolerant CRDTs matter?" by looking to the current practical relevance and limitations of Byzantine-tolerant CRDTs as well as to potential combinations with access control and/or further coordination mechanisms. First of all, the Matrix Event Graph with its grow-only HashDAG design proves the practical relevance of Byzantine-tolerant CRDTs for the use case of instant messaging. But is there a relevance for other use cases than the Matrix one? The Matrix example also points to limitations: while a grow-only data structure avoids the need of handling of deletion events, it brings up the issue of garbage collection and spamming. Furthermore, one has to rethink the applicability to other application scenarios which we will discuss in the following.

State-based CRDTs are easy to deploy in Byzantine environments because of their unconditional equivocation tolerance we showed in Corollary 4.2. However, the identity of updates gets lost since only states are propagated in the system. It is not possible to reconstruct the update that led to a new state, which makes it impossible to prove which replica performed which CRDT updates and whether it was allowed to do so. Hence, access control on the different operations of state-based CRDTs, giving different permissions to different participating replicas, cannot be enforced. Therefore, state-based CRDTs might be suitable for decentralized systems in the spirit of the Newsgroup system where any user can write new articles and reply to old ones.

In contrast to state-based CRDTs, with operation-based CRDTs the original caller of an update operation can be determined and verified with authentication mechanisms like digital signatures. Therefore, operation-based CRDTs provide the necessary prerequisites for access control, which makes them easier to deploy in more demanding systems.

In general, CRDTs forfeit consensus and coordination (cf. the example of SEC but no consensus in Appendix C) in favor of availability and partition tolerance. Without Sybil countermeasures like controlled membership, the space of solvable application-layer problems is the class of invariant-convergent problems, i.e., invariants for which local, uncoordinated replica decisions are sufficient to preserve the invariants globally [KH20]. While this takes cryptocurrencies out of question, the typical CRDT use cases for which Strong Eventual Consistency suffices, e.g., collaborative applications like shopping lists, text editors or whiteboards, are also invariant-convergent and, therefore, uses cases for Byzantine-tolerant CRDTs.

When the application layer requires stronger guarantees, e.g., strong consistency, Byzantine-tolerant CRDTs can obviously only serve as part of a solution and need to be combined with other, stronger mechanisms. While the combination 'technically' inherits the stronger assumptions, from a practical deployment perspective, a hybrid mode of operations might make sense: an application can make use of the Byzantine-tolerant CRDT to collect data only requiring corresponding weak assumptions on the system. Whenever stronger assumptions like synchronous operation are fulfilled at certain phases in time, more demanding tasks like consensus can be performed, which also allows for garbage collection in the CRDT layer. Thus, the resulting hybrid system would fall into the category of partially synchronous system [DLS88], reaching agreement on previously aggregated updates.

We hope the characterization of Byzantine-tolerant CRDTs we presented in this paper provides a basis for further exploration.

A Best-Effort Broadcast

For convenience and to avoid misunderstanding, the definition of best-effort broadcast as presented in [CGR11, Section 3.2] is reproduced here. An example best-effort broadcast algorithm implementation can also be found there.

Abstraction 1 Best-effort Broadcast Interface and Properties

Events:

$\langle Broadcast, m\rangle$: Broadcasts a message m to all processes.

$\langle Deliver, p, m\rangle$: Delivers a message m broadcast by process p.

Properties:

Validity: If a correct process broadcasts a message m, then every correct process eventually delivers m.

No duplication: No message is delivered more than once.

No creation: If a process delivers a message m with sender s, then m was previously broadcast by process s.

B Conflict-free Replicated Data Types

The following formalization and example of Conflict-free Replicated Data Type (CRDT) implementations, i.e., replicated objects that belong to one of the two CRDT families, is based on the original CRDT paper [Sh11] and on the paper [Ja21] that analyzes the Matrix Event Graph. A state-based replicated object is defined as a tuple (S, s^0, q, u, m). S is the space of possible per-replica states; the replica at process p_i has state $s_i \in S$. The initial state of every replica is s^0. The query method q returns the current state of the replica. The update method u modifies the current state. The merge method m merges the state from a remote replica with the current local state. If Eventual Delivery and termination is ensured, an state-based replicated object provides SEC and thereby is a state-based CRDT if the set of possible states S with the merge function m is a join-semilattice.

An operation-based replicated object is defined as a tuple (S, s^0, q, t, u, P). Again, S is the space of possible states and s^0 the initial state. The query method q returns the current state. The update method is composed of a side-effect-free generator step t and an effector step u that performs the state change. The generator step is executed by the source replica and returns an operation that is then broadcast to all replicas. Then, every replica executes the effector step of the update, which applies the operation to their current state. The effector may contain a precondition P, which must be fulfilled before an operation is applied. If causal delivery of updates and termination is ensured, an operation-based replicated object provides SEC and thereby is a operation-based CRDT if all concurrent updates are commutative, and the delivery precondition P is satisfied by causal delivery.

As an example, we provide the definition of the Matrix Event Graph (MEG) grow-only HashDAG as operation-based replicated object in Algorithm 1, which was shown to be a Byzantine-tolerant, operation-based CRDT in [Ja21]. Each vertex is a tuple (e, w) with w

Algorithm 1 Matrix Event Graph Operation-based Replicated Object

```
state set $S = (V, E)$ ▹ vertices are of form $V = (\texttt{event}\ e, \texttt{uid}\ w)$, $E$ represents edge $E \subseteq (V \times V)$
init $(\{e_0, w_0\}, \emptyset)$
query lookup (uid $w$) : boolean
    return $\exists (e', w') \in V : w' == w$
query hasChild (vertex $(e, w)$): boolean
    return $\exists ((e', w') \in V) : ((e', w'), (e, w)) \in E$
query getExtremities () : list of vertices
    return $L = \bigcup_{(e,w) \in V : !\texttt{hasChild}((e,w))} \{(e, w)\}$
query getState () : set
    return $S$
update append
    generator
        let $L$ = getExtremities()
        let $w$ = unique$(L, e)$
        return append, $(e, L, w)$
    effector event $e$, list of vertices $L$, uid w
        pre $\forall (e_p, w_p) \in L$ : lookup$(w_p)$
        $V = V \cup \{(e, w)\}$
        $E = E \cup \bigcup_{(e_p, w_p) \in L} \{((e, w), (e_p, w_p))\}$
```

being a unique identifier and e the actual event. Edges represent the causal relationship between a child and a parent vertex. Then, the state is a DAG which is defined through vertices and edges. The types of methods are indicated with **query** and **update**. The precondition of the effector step is indicated with **pre**. When appending a new event e to the graph, the generator returns operation in the form (e, L, w), where L is the list of current forward extremities, i.e., vertices without children. Let h_1 be a cryptographic hash function that provides for the content identifier $h_1(e)$ of message e. Assume that $w_1, \dots, w_n$ are the unique identifiers of the operations that added the elements of L to the DAG. Then the content identifier (and unique identifier) w for the operation (e, L, w) that adds message e is given by $\texttt{unique}(L, e) = h_2(w_1, \dots, w_n, h_1(e))$, for a cryptographic hash function h_2. After validation, the operation (e, L, w) is applied by the effector which first verifies that the parent vertices from L are already part of the current state, and then adds the new vertex and the edges between the new vertex and the parent vertices from L.

C Example of an Equivocation tolerated by the MEG while being an Application-Layer Conflict

In Sect. 3, we showed that the Matrix Event Graph (MEG) is a Byzantine-tolerant CRDT, and can thereby guarantee the notion of 'conflict-freedom', defined as Strong Eventual Consistency (SEC) in Sect. 2, in face of equivocation. However, this 'conflict-freedom' on the CRDT layer does not mean that the equivocation does not also present a conflict on the application layer that might even require consensus, which is out of scope for our work. As an example, in Fig. 2, we show an equivocation that contains the classical "Attack!" and

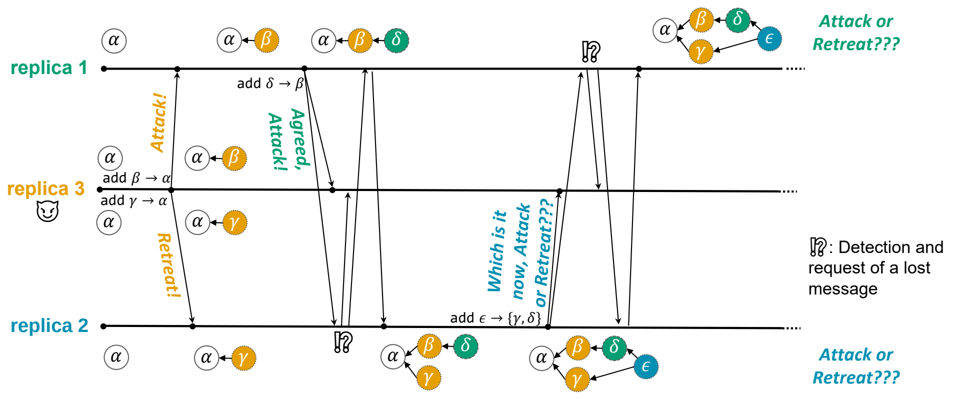


Fig. 2: Evolution of a MEG over time in face of replica 3 running on a Byzantine faulty process performing equivocation on replicas 1 and 2 running on correct processes. Messages associated to operations β and γ are an application-layer conflict, and replica 3 equivocates by sending $\beta \longrightarrow \alpha$ to replica 1 but $\gamma \longrightarrow \alpha$ to replica 2. The MEG, guaranteeing the SEC formalization of 'conflict-freedom' to the application layer, eventually provides converged current states of the causal history at both replica 1 and 2, and contains both equivocated operations β and γ. To solve the remaining application layer conflict based on the synchronized causal history is left to the application.

"Retreat!" conflict of the Byzantine Generals Problem [LSP82] on the application layer, while the MEG still provides SEC to the application layer.

All replicas start with initial state $s^0 = \alpha$. Replica 3 then creates the messages "Attack!" with operation β and "Retreat!" with operation γ that conflict on the application layer, and performs equivocation by sending an update attaching β to α at replica 1, while sending the different update to replica 2 that instead attaches γ to α there. The current state of replica 1 and 2 is thereby momentarily inconsistent. When replica 1 now broadcasts $\delta \longrightarrow \beta$, replica 2 notices that it is missing the operation containing β, and re-requests it from replica 1 and 3. While replica 3 pretends not to know about β, replica 1 returns $\beta \longrightarrow \alpha$ and thereby enables replica 2 to apply both $\delta \longrightarrow \beta$ and $\beta \longrightarrow \alpha$. The respective procedure repeats with $\eta \longrightarrow \gamma, \delta$ broadcast by replica 2. After both replica 1 and 2 added η to their graphs, their current states are consistent again.

The example shows that while the application layer still has to deal with conflicts, it can at least rely on SEC in the way that both α = "Attack!" and β = "Retreat!" end up as parallel events in the current state of both replica 1 and 2, no later than when those current states have eventually reached consistency again.

D Acknowledgements

We like to thank the anonymous reviewers as well as our shepherd Prof. Dr. Joachim Posegga for their instructive comments that helped us a lot to improve the paper. The work of Florian Jacob and Saskia Bayreuther is supported by the JuBot project. The JuBot project is funded by the Carl-Zeiss-Foundation.

Bibliography

[Au21] Auvolat, Alex; Frey, Davide; Raynal, Michel; Taïani, François: Byzantine-tolerant causal broadcast. Theoretical Computer Science, 2021.

[Ca99] Castro, Miguel; Liskov, Barbara et al.: Practical byzantine fault tolerance. In: OSDI. volume 99, pp. 173–186, 1999.

[CGR11] Cachin, Christian; Guerraoui, Rachid; Rodrigues, Luís: Introduction to reliable and secure distributed programming. Springer Science & Business Media, 2011.

[Ch07] Chun, Byung-Gon; Maniatis, Petros; Shenker, Scott; Kubiatowicz, John: Attested append-only memory: Making adversaries stick to their word. ACM SIGOPS Operating Systems Review, 41(6):189–204, 2007.

[DLS88] Dwork, Cynthia; Lynch, Nancy; Stockmeyer, Larry: Consensus in the presence of partial synchrony. Journal of the ACM (JACM), 35(2):288–323, 1988.

[Ho21a] Hodgson, Matthew: , Element raises $30M to boost Matrix. `https://matrix.org/blog/2021/07/27/element-raises-30-m-to-boost-matrix`, 2021. The Matrix.org Foundation C.I.C.

[Ho21b] Hodgson, Matthew: , Germany's national healthcare system adopts Matrix! `https://matrix.org/blog/2021/07/21/germanys-national-healthcare-system-adopts-matrix`, 2021. The Matrix.org Foundation C.I.C.

[Ja21] Jacob, Florian; Beer, Carolin; Henze, Norbert; Hartenstein, Hannes: Analysis of the matrix event graph replicated data type. IEEE Access, 9:28317–28333, 2021.

[KH20] Kleppmann, Martin; Howard, Heidi: Byzantine eventual consistency and the fundamental limits of peer-to-peer databases. arXiv preprint arXiv:2012.00472, 2020.

[La78] Lamport, Leslie: Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM, 21(7):558–565, 1978.

[LSP82] Lamport, Leslie; Shostak, Robert; Pease, Marshall: The byzantine generals problem. ACM Transactions on Programming Languages and Systems, 4(3):382–401, 1982.

[Sh11] Shapiro, Marc; Preguiça, Nuno; Baquero, Carlos; Zawirski, Marek: Conflict-free replicated data types. In: Symposium on Self-Stabilizing Systems. Springer, pp. 386–400, 2011.

[Th21] The Matrix.org Foundation C.I.C.: Matrix specification v1.1. Technical report, 2021. `https://spec.matrix.org/v1.1/`.

[Zh16] Zhao, Wenbing: Optimistic byzantine fault tolerance. International Journal of Parallel, Emergent and Distributed Systems, 31(3):254–267, 2016.