Making CRDTs Not So Eventual

Yunhao Mao
University of Toronto
yunhao.mao@mail.utoronto.ca

Gengrui Zhang
University of Toronto
gengrui.zhang@mail.utoronto.ca

Zongxin Liu
University of Toronto
zongxin.liu@mail.utoronto.ca

Pezhman Nasirifard
Technical University of Munich
P.nasirifard@tum.de

Sofia Tijanic
University of Toronto
sofia.tijanic@mail.utoronto.ca

Hans-Arno Jacobsen
University of Toronto
jacobsen@eecg.toronto.edu

ABSTRACT

Conflict-free replicated data types (CRDTs) are designed to deliver high performance and availability in distributed applications through eventual consistency. However, eventual consistency alone is insufficient to guarantee application correctness. This limitation becomes more pronounced in the presence of Byzantine failures. Naively applying consensus and traditional Byzantine fault tolerance (BFT) protocols on CRDT updates for stronger guarantees, while intuitive, negates the performance benefits of CRDTs.

We introduce a novel programming model called *Reliable CRDTs* as an *extension to CRDTs* with additional guarantees: users can query CRDT values that are either strongly or eventually consistent, enforce a total order among selected operations, and define data-type level invariants maintaining correctness in the presence of Byzantine failures. Reliable CRDTs enable the use of CRDTs in scenarios where strong consistency is required, offering both performance benefits and ease of use.

We present a design named *Janus* that implements Reliable CRDTs. *Janus* functions as a middleware that facilitates CRDT communication, thereby enhancing CRDTs with the aforementioned capabilities by utilizing a novel *direct acyclic graph based* BFT consensus protocol. The evaluation of *Janus* demonstrates that it achieves 21× greater throughput than state-of-the-art BFT protocols such as HotStuff achieve, and it remains responsive even under heavy loads.

PVLDB Reference Format:
Yunhao Mao, Gengrui Zhang, Zongxin Liu, Pezhman Nasirifard, Sofia Tijanic, and Hans-Arno Jacobsen. Making CRDTs Not So Eventual. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Romsdal/Janus-CRDT.

1 INTRODUCTION

Conflict-free replicated data types (CRDTs) are abstract data types (ADTs) designed to replicate across multiple nodes with eventual consistency guarantees [39]. When CRDTs are employed, users simply execute ADT interface operations on one replica, and other replicas eventually converge to the same state, as long as they receive the same set of updates, regardless of the order in which the updates are applied. CRDTs eliminate the need for manual conflict resolution and offer distributed system developers an easy-to-use replication solution through well-defined semantics, introducing a more efficient programming model to distributed applications.

Because CRDTs are eventually consistent, they offer high performance and high availability by allowing temporary divergent states to exist. Replicas can update their local states immediately upon receiving updates and then propagating the changes asynchronously [7]. However, this also means that CRDTs cannot meet the correctness requirements of many distributed applications because of the existence of intermittent divergent states [13, 37]. We identify three scenarios where CRDT guarantees are insufficient.

First, users cannot determine whether the system has reached a *stable* state. This is because CRDTs lack a mechanism at the data type level to detect whether an update has been delivered to all replicas or to verify the freshness of a value on a given replica, as eventual consistency does not guarantee a bounded delivery time [7]. If an application needs to perform operations that depend on prior operations or a specific state, such as transactions at the application level, additional consistency checks are required to ensure that the dependent operations have been successfully executed on *all* replicas. For example, in a distributed banking database, if one replica performs a transfer transaction (a withdrawal followed by a deposit), it must first ensure that the withdrawal operation is successful on *all* replicas before performing the deposit operation.

Second, it is difficult to implement correctness checks for CRDT values, such as maintaining invariants, because of concurrent operations [10, 28]. In the banking example, two concurrent withdrawals on two replicas of the same bank account may cause the balance to fall below the minimum balance limit, even if neither operation violates the invariant when executed individually.

Above all, CRDTs cannot operate in the presence of *Byzantine failures* [27]. These are arbitrary failures that a distributed system may experience, such as attacks, software bugs, or hardware malfunctions. Faulty participants can disseminate *conflicting* CRDT messages to different replicas that indefinitely prevent them from converging [51]. For example, a Byzantine replica can mislead some replicas into believing that a deposit has occurred while simultaneously convincing others that there is a withdrawal.

Traditionally, these problems are addressed by using strongly consistent storage and transactional models that enforce a linear order of operations across all replicas [17, 23]. However, these solutions require costly and time-consuming coordination among

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

replicas, such as consensus, which significantly impacts the system's performance [12]. Thus, in this paper, we propose a novel programming model called *Reliable CRDT* that enhances CRDTs with **on-demand strong consistency**, the **selective ordering** of updates, **data-type level invariants**, and the ability to tolerate **Byzantine failures**. Our solution enables the utilization of CRDTs in scenarios where strong consistency was previously required and to expand the scope of existing CRDT-based applications.

With *on-demand strong consistency*, users can either retrieve a *stable value* that is guaranteed to be identical across all correct replicas or retrieve a *prospective value* that is eventually consistent. The *selective ordering* of updates allows users to designate certain updates as *safe* and ensures that they are executed in a total order across all correct replicas. Applications can define *data-type level invariants* as constraints that the stable value must not violate. Finally, Reliable CRDTs tolerate Byzantine failures by preventing *conflicting updates* from violating the guarantees mentioned above.

Reliable CRDTs provide a flexible toolkit for developers to utilize CRDTs to improve performance and ease of development. Stronger guarantees can be employed to meet application-level correctness requirements when necessary without requiring customized consistency protocols.

We implement Reliable CRDTs via a middleware system named *Janus*[1] to handle the communication among CRDT replicas. Our design uses an *underlying consensus protocol* running asynchronously to ensure that all correct replicas observe the same set of valid updates (neither conflicting nor invariant breaking). If updates are found to be invalid by the consensus process, they are *reversed* through the mechanism of *Reversible CRDTs* [31] during the *audit* process. We incorporate a novel *direct acyclic graph (DAG) based* BFT protocol [47] as the underlying consensus protocol because it aligns with the way that CRDTs operate by offering high throughput, the ability to asynchronously order updates, and a method of keeping track of the update history, which are not possible with traditional BFT protocols.

We evaluate *Janus* by incorporating a CRDT-based distributed key-value database with *Janus* as a proof of concept [1]. The performance results are compared with those of plain CRDTs and solutions that apply BFT consensus protocols directly to CRDTs updates.

This paper is organized as follows: Section 2 discusses the background information. Section 3 provides an overview and identifies the applications of Reliable CRDTs. Section 4 introduces the properties of Reliable CRDTs. Section 5 presents the design of *Janus*. Section 6 demonstrates the performance evaluation. Finally, Section 7 discusses related work.

2 BACKGROUND

In this section, we introduce CRDTs and DAG-based BFT protocols as they are essential for understanding our algorithms.

[1]Janus is a Roman god with two faces, which symbolizes changes, transitions, gates, and doorways.

2.1 Conflict-Free Replicated Data Types

First proposed by Shapiro et al. [35, 39], CRDTs are replicated *abstract data types* with deterministic interface operations and predefined *concurrency semantics*; they avoid the need for developers to define application-specific reconciliation mechanisms and provide an easy-to-use replication solution through the interaction with the established interface. A CRDT represents a state (the *data structure* to be replicated), and the interface is a set of operations. Operations with *side effects* (which change the state of the data structure) are referred to as *updates*, and meaningful values can be retrieved through *query* operations without side effects.

When an update is executed on a replica of a CRDT instance, it changes the local state of the executing replica immediately, and then it is propagated asynchronously to other replicas. CRDT replicas adhere to *strong eventual consistency* (SEC), which ensures that updates are eventually delivered to all correct replicas and replicas that have received the same updates have equivalent states. This implies that the convergence of two replicas' states can be determined by checking which updates have been delivered.

There are two main methods of propagating updates that enable CRDTs to achieve SEC, namely, *state-based* (CvRDT) and *operation-based* (CmRDT) protocols. CvRDTs synchronize by sending the entire state, or the delta of the state change [3], of a replica and then *merge* the received states with a *commutative* merge function. This function ensures the same result regardless of the order in which operations are executed. For example, a counter CRDT may use a vector of integers as the state, and merging a new state involves taking the elementwise maximum, similar to a vector clock. CmRDTs propagate the content of the update and execute the same updates on other replicas with the same causal order [26], which requires the operations themselves to be commutative and the propagation of updates to be reliable. For example, a counter may use the addition and subtraction of integers as updates, with receiving replicas performing the same operations to obtain the same result.

The primary characteristic of CRDTs is that the logic behind updating a value is embedded at the data store level through the semantics of CRDTs via fire-and-forget updates, which contrasts with traditional eventual consistency solutions or transactional models. In these systems, updating data requires a read-modify-write transaction on the data store, necessitating an additional consistency protocol to either reconcile conflicting states or to ensure that all replicas perform this transaction in the same order [7]. This distinction offers a unique advantage for CRDTs in terms of ease of use, as the consistency mechanism and date operations are combined. This also leads to a different programming model for data handling in applications that use CRDTs: the separation of data operations from application-level transactions. We discuss this in more detail in Section 3.

Because updating a CRDT object does not require a preceding read, **querying a CRDT is useful only when the value is immediately consumed by the application context**. From a user's perspective, the *effect* of an update on the later queried value is the only factor that determines whether an update is successful. The effect can be viewed as the *delta* between the local values on a replica before and after the update is applied [31]. For example, incrementing a counter by 5 means that the delta is +5. If the delta is 0, the update can be considered nonexistent.

2.2 DAG-based BFT Consensus Protocols

As mentioned, *Janus* utilizes an underlying BFT consensus protocol to achieve the aforementioned guarantees. We choose a novel DAG-based BFT consensus protocol because it allows replicas to concurrently propose and broadcast messages, which enables us to piggyback CRDT updates onto the DAG's message propagation mechanism. In contrast, traditional BFT consensus protocols often require a leader replica to determine the order of messages first, followed by replicas collectively voting to reach an agreement, which prevents preemptive propagation of CRDT updates. In a DAG BFT protocol, replicas construct DAGs based on the causal delivery order of the messages, and the total order of the messages is independently determined by individual replicas by traversing through the DAG.

DAG BFT protocols, such as Hashgraph [9], Aleph [19], and, more recently, DAG-Rider [22], Narwhal & Tusk [18], and Bullshark [40], originally emerged to improve blockchain system throughput by decoupling message propagation and ordering logic [47]. In state-of-the-art protocols such as DAG-Rider and Narwhal & Tusk, each replica maintains a copy of the DAG of *blocks* of messages, as illustrated in Figure 1.

The DAG is organized into *rounds*. In each round, a replica generates a block and reliably broadcasts [15] the block to other replicas asynchronously. Once a replica receives $2f + 1$ blocks from other replicas in a round, it advances to a new round and constructs a new block to broadcast containing the latest messages and references to the blocks in the previous round. Referencing blocks *supports* the referenced blocks, as it indicates that the referenced blocks have been observed by the replica. Different replicas may view different variations of the DAG at a given moment, but the reliable broadcast protocol guarantees that all replicas eventually see the same DAG.

Next, replicas commit blocks to persist the agreed-upon messages; this occurs every few rounds (which constitutes a *wave*). The commit process is triggered independently on each replica once the wave is reached. At each wave, the replicas independently select a common *leader block* among the blocks that are received in the first round of the wave by using a perfect random coin (a random variable that allows the replicas to independently retrieve the same random value at the same wave) [16]. The leader block must also be supported by at least $2f + 1$ successor blocks from the second round; otherwise, no block is committed in the wave. Then, the blocks leading up to the leader are committed by deterministically converting the causal ordering of the blocks to a global total order.

Figure 1 shows an example operation of Narwhal & Tusk. Its consensus process uses three rounds for each wave, with the third round of a wave and the first round of the subsequent wave overlapping to decrease latency. Block $L1$ is the leader of wave $w - 1$ (from round $r - 4$ to $r - 2$), and the red blocks are the predecessor blocks that are committed when $L1$ is committed. Block $L2$ is the leader of wave w (from round $r - 2$ to r), with blue preceding blocks; $L3$ has green preceding blocks. The commit process is conducted based on the local copy of the DAG for each replica; thus, no additional communication overhead is introduced.

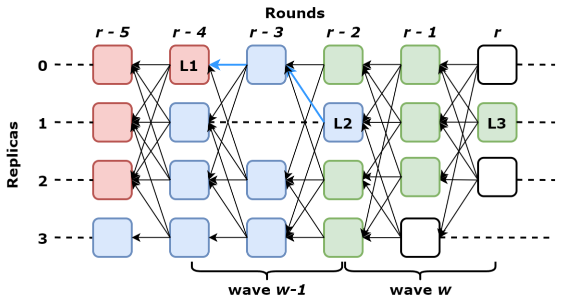


Figure 1: Example of a Narwhal DAG & Tusk consensus process [18] instance from the perspective of one replica currently at round r. Note that $L1$ is not committed until $L2$ is committed because it does not have at least $2f + 1$ supports in round $r - 3$ and must wait for a path connecting it to the next committed leader.

In *Janus*, concurrent block propagation allows for the prompt dissemination of CRDT messages. The greater latency of committing individual messages caused by batching multiple rounds of updates in DAG BFT protocol is mitigated by the fact that CRDT updates are preemptively executed without waiting for consensus as CRDT state convergence does not rely on the order of updates. The DAG also stores the updates as a log, which is useful for reversing invalid updates.

3 RELIABLE CRDT OVERVIEW

Strong consistency solutions, such as transactional databases, are still preferred by many distributed applications, despite the availability and performance advantages of eventual consistency. Therefore, our fundamental design philosophy is to enable applications to use CRDTs as the primary method of replication in scenarios where only strong consistency can meet application-level correctness requirements. Our approach offers stronger guarantees and requires no alteration to the programming model of CRDT data operations while maintaining their performance benefits, as discussed in Section 2.

We define data operations as the interactions of CRDT instances, including *updates* and *queries*. The *application-level correctness requirement* is the set of guarantees that the CRDT object must satisfy to ensure the application's correctness. An *application-level transaction* is a series of data operations that the application defines to achieve its functionality when the correctness requirement is met.

Reliable CRDTs add *on-demand strong consistency* to CRDT values and enforce *selective ordering* for operations by enhancing existing underlying CRDTs with new operations with additional guarantees and Byzantine fault tolerance. The original CRDT operations are still accessible, and their behaviors are unchanged and remain compatible:

(1) Queries can either retrieve strongly consistent *stable* values or eventually consistent *prospective values* from a CRDT instance.
(2) Updates can be designated as *safe*, and safe updates are executed in a total order across all replicas.
(3) Users can define one or more *data-type level invariants* that the CRDT instance honors.

(4) These guarantees hold even in the presence of Byzantine failures that result in conflicting updates.

Before presenting examples to illustrate how our solution enables the use of CRDTs in various scenarios, we categorize three different ways in which applications perform data operations with CRDTs: ① retrieving a CRDT's value without subsequent action; ② updating a CRDT's state without dependency while not breaking invariants; and ③ conducting operations with dependency in *application-level transactions*, requiring mutually exclusive access to an object, or potentially breaking invariants.

Banking Database. In this example, we focus on the consistency and safety guarantees provided by Reliable CRDTs. Furthermore, replicas in the system may be subject to Byzantine failures, which can occur even for internally hosted applications [43]. Consider a replicated banking database whereby clients can check, deposit, withdraw, and transfer amounts of money. These operations can be conducted on any replica, and each account has an overdraft limit as an application-level correctness requirement, which is implemented as a data-type level invariant. Although a plain CRDT counter can ensure that the final balance converges, it cannot guarantee that concurrent withdrawals do not violate the overdraft limit. Faulty replicas can also send conflicting updates to different replicas, causing issues such as one replica believing that an account is overdrawn while another one does not. Using Reliable CRDTs can guarantee the correctness of these operations without using a transactional model.

Assuming a client wishes to check the latest account balance immediately after depositing money, then the "check balance" operation is type-①. Stale or intermediate values are acceptable here since there is no critical follow-up operation. Reliable CRDTs offer the use of a prospective value for type-① operations by providing the latest value without the need to wait for convergence. The latest balance can be retrieved (as long as the update has been received on the replica) even if the deposit may need to be settled and will only be available for use later (after a consensus agreement has been reached on the balance).

Depositing is a type-② operation. There is no causal dependency between this data operation and previous updates that were executed on the CRDT instance (i.e., the ordering of these updates does not matter), and it cannot ever break the overdraft limit. These types of operations are also called *invariant-confluence (I-confluence)* operations; they do not break the invariants of the application and can be executed concurrently [6, 33]. Since plain CRDT operations are designed to be I-confluent, these operations can be executed preemptively.

Withdrawal, however, is potentially invariant-breaking and can be categorized as a type-③ operation. Therefore, the actual account balance must first be checked by using the stable value to ensure that there are sufficient funds. Then, a safe update must be performed to ensure that no two concurrent withdrawals violate the application-level correctness requirement.

Transferring money from one account to another requires two dependent operations, so it is also type-③: withdrawal from one account and deposit into another account is an *application-level transaction.* Safe updates must be used to ensure that the withdrawal is successful before the deposit is made because they are dependent operations that must be executed in a specific order involving updating CRDTs. The application then determines the behavior of subsequent dependent updates based on the status of the first operation. This transaction can also be optimized to allow the deposit to be conducted concurrently without waiting for the withdrawal to succeed if there is a predefined method to undo the deposit if the withdrawal fails.

An example implementation of the banking database is shown in Algorithm 1 with the abstractions provided by Reliable CRDTs.

Algorithm 1 Banking database using Reliable CRDTs

```
function check_invariant($balance$)
    return $true$ if $balance \geq 0$ else $false$ ▷ Balance must be nonnegative

function withdraw($amount$, $account : counter$) ▷ Account is a CRDT counter
    $balance \leftarrow account.query\_stable()$
    if $balance > amount$ then
        return $account.decrement\_safe(amount)$ ▷ Safe decrement, only returning true if the operation is successful

function deposit($amount$, $account$: $counter$)
    $account.increment(amount)$ ▷ Since depositing never breaks the invariant, we can use the regular increment and assume it will eventually be included in the stable state

function display_balance($account : counter$)
    return $account.query\_prospective()$ ▷ Displaying the balance is not critical

function calculate_interest($account : counter$)
    $amount \leftarrow account.query\_stable()$ ▷ Interest is calculated based on the settled cash in the account
    $account.increment(amount \times interest\_rate)$

function transfer($account1, account2$: $counter$)
    $u1 \leftarrow account1.decrement\_safe(amount)$
    if $u1.success$ then
        $u2 \leftarrow account2.increment(amount)$
    return $u1.success \wedge u2.success$
```

Voting System. In this example, we illustrate the direct impact of Byzantine failures on the correctness of the system. Consider a voting system that uses a replicated counter to store the number of votes and allows voters from multiple voting sites to cast ballots concurrently by incrementing the counter. The system should ensure that each participant can only vote for one candidate. There is no invariant in this case, but a Byzantine replica can double-cast votes to two correct replicas, causing the vote count to be inconsistent and unknown to the application.

Here, casting a vote is a type-② scenario since it does not depend on any previous state or operation. Viewing the vote count before voting closes is I-confluent since it does not require any subsequent actions. However, viewing the vote count after voting closes is a type-③ operation since even if no data operations are performed, the application has a dependent subsequent action, which is to determine the winner with an application-level correctness requirement that all voting stations see the number of votes. Thus, the stable value must be read to ensure that the final vote count is correct.

The example implementation of the voting app is shown in Algorithm 2

Algorithm 2 Voting system using Reliable CRDTs

```
1: function cast_vote($candidate : counter$)
2:     $candidate.increment(amount)$
3: function display_vote($candidate : counter$)
4:     return $candidate.query\_prospective()$ ▷ Displaying the vote count before the voting closes is allowed to be inconsistent
5: function decide_winner($all\_candidates : listofcounter$)
6:     $votes : List$
7:     for all $candidate$ in $all\_candidates$ do
8:         $votes.add(candidate.query\_stable())$ ▷ Need to make sure all replicas have received the same vote counts, and any conflicting votes are excluded
9:     return $all\_candidates[votes.index(max(votes))]$ ▷ Winner is decided based on the stable state
```

4 RELIABLE CRDT DESIGN

4.1 System Model and Notation

We consider a distributed system that consists of fully connected nodes that communicate via message passing. The network is asynchronous but reliable, which means that messages will not be lost, duplicated, or reordered. This can be accomplished via reliable communication protocols, such as TCP. However, messages may experience arbitrary delays.

Faulty nodes may exhibit arbitrary Byzantine behavior with at most f faulty nodes, and the total number of nodes needed for the BFT protocol to function is at least $3f + 1$. The system is *permissioned* [4], where the nodes in the system are fully known to one another. Each node is identified by the public key of a private–public key pair using modern cryptographic signature algorithms, such as ECDSA [14]. All of the messages are signed with private keys.

A node stores one replica $\mathcal{R}$ for each CRDT instance in the system. The state of one CRDT instance is denoted as c, and a replica of the CRDT instance on node i is a linear progression of states from the initial state $\mathcal{R}_i = \{c_1, \ldots, c_n\}$ after n updates. Each state is a collection of update operations (denoted as u) that the replica has received at the given time, and the last state is $c_n = \{u_0, u_1, \ldots, u_n\}$. We say that an update u is *executed* on replica $\mathcal{R}$ when u is included in the state of $\mathcal{R}$ and yields a new state, i.e., $c_{k+1} = c_k \cup u$.

According to the definition of CRDTs, the state of a replica is monotonically increasing[2]; that is, $c_i \subseteq c_j$ if $i < j$. Thus, for two updates u_x and u_y, u_x *happened-before* u_y if u_x is included in the state of **all** replicas when u_y is received, denoted as $u_l \prec u_m$; it *happened-after* in the opposite case. u_x and u_y are *concurrent* if neither of them *happened-before* or *happened-after* the other. These definitions constitute the causal relations of CRDT updates [26, 39].

The value of a CRDT is denoted as $Q(c)$, where Q is a query function defined by the CRDT. If two replicas i and j have the same queried value, $Q(c_i) = Q(c_j)$, they are said to be *equivalent*. We define the *effect* of an update u as the delta of the queried value before and after the update is executed, denoted as $E(u) = Q(c_{i+1}) - Q(c_i)$. If there are two updates such that one update inverts the effect of the other, the states before and after the updates can be considered equivalent.

The data type invariant I is defined as a set of constraints $I = \{I_0, I_1, \ldots, I_n\}$, where each constraint can be viewed as a Boolean function $I_i(c)$ that returns true if state c satisfies the constraint.

4.2 Definition of Reliable Operations

In this section, we provide formal definitions of the operations and guarantees of Reliable CRDTs, including the behaviors of prospective and stale values, the guarantees of safe updates, and the concept of conflicting updates.

Q_{stable} and $Q_{prospective}$ are functions that query stable or prospective values. The prospective value is the state of a replica that results from the updates that the replica has already received, which is the same as querying a plain CRDT. However, the state could be inconsistent, stale, or invariant-breaking.

The stable values can be viewed as a log of *stable states* that are consistent across all correct replicas, which is called the *stable history*. All stable states in the stable history satisfy invariants and are guaranteed to be reached by all correct replicas in the same order. Querying a stable value returns the stable history of the replica.

Using the banking database as an example, an integer counter CRDT is used to store account balances. Assuming that there are two replicas with sequences of states $\mathcal{R}_0 = \{5, 3, 4, 2\}$ and $\mathcal{R}_1 = \{5, 1, 4, 7\}$ and that the stable history is $L = \{5, 4\}$, then this means that for both R_0 and R_1, a stable query yields $\{5, 4\}$, while prospective reads may return intermediate values such as 2 on R_0 and 7 on R_1.

Definition 1 (Stable History). *Let the stable history L_i be a subset of a CRDT instance replica $L_i \in \mathcal{R}_i$ on a node i with a list of **stable states** and a set of invariants I: $L_i = \{c_1, c_2, \ldots, c_n\}$ s.t. $\forall c \in L_i, \forall I_j \in I, I_j(c) = true$.*

Let c_k be a stable state in L_i at index k, then for all other nodes j, where $k \leq n$ and $c'_k \in L_j$, $Q(c_k) = Q(c'_k)$.

Definition 2 (Stable Query). $Q_{stable}(\mathcal{R}_i) \leftarrow L_i$

To ensure the freshness of the stable value returned from a replica, a user can perform a *quorum read* [44] to gather the stable history logs from at least $f + 1$ replicas. Since the stable values are consistent among the correct replicas, the union of stable history logs can be used to determine the latest stable value[3].

The update operations behave the same way as in the underlying CRDTs. If the update is designated *safe*, denoted as u^t, it will be returned to the caller as successful only after it has been successfully

[2]Although a monotonically increasing state is only required for state-based CRDTs, all kinds of CRDTs are mathematically equivalent, and they can be constructed to have the same structure [39].

[3]In practice, it is sufficient to return only the latest stable value with $f + 1$ nodes recognized.

included in the stable state; if it is successful, the stable state must also be updated. In contrast, plain CRDT updates are considered successful as long as they are executed locally. A safe update may never bring an instance to an invariant-breaking state, and all safe updates on the system are linearizable.

For example, if the stable account balance is 5 at one point and there are two concurrent safe withdrawals of 3 and 4 on both replicas, the system will order the withdrawals on *all* replicas. Assuming that -3 *happened-before* -4, the second withdrawal returns false only after the first withdrawal returns successful for all replicas.

Definition 3 (Safe Update). *Let an update u^t be safe, then for all histories L_i of a CRDT instance on all replicas, $\exists c \in L_i$ s.t. $u^t \in c$. The set of all safe updates $\{u_0^t, u_1^t, \ldots, u_n^t\}$ of all CRDT instances is totally ordered.*

We offer Byzantine safety by preventing *conflicting updates* [51]. We assume that all updates originating from one correct node are always propagated to all other nodes in the same linear order. If two correct replicas receive updates from the same source at the same "position" but have different content, they are conflicting updates. Faulty replicas are the only cause of conflicting updates, as our network assumptions do not allow for reordered messages.

Definition 4 (Conflicting Updates). *Consider three distinct updates u_1, u_2, and u_3 from $\mathcal{R}_i$ that are delivered on a replica $\mathcal{R}_m$ in the order of $u_1 \prec u_2 \prec u_3$. If another replica $\mathcal{R}_n$ receives $u_1 \prec u_2' \prec u_3$, where $u_1(\mathcal{R}_m) = u_1(\mathcal{R}_i)$ and $u_3(\mathcal{R}_m) = u_3(\mathcal{R}_i)$, but $u_2' \neq u_2$ (in terms of resulting in a different state given the same initial state), then u_2 and u_2' are conflicting updates.*

Definition 5 (Byzantine Safety). *No conflicting updates are included in the stable history of any correct replica.*

Finally, we must consider concurrently executed updates on the prospective value because they may be conflicting or invariant-breaking. *Eventual validity* is a safety guarantee for them to eventually converge to a valid state; that is, querying the CRDT for either prospective values or stable values eventually yields the same result. This also implies that any update that is conflicting or invariant-breaking (an *invalid* update) will eventually be removed.

Definition 6 (Eventual Validity). *For any correct replica $\mathcal{R}$ of a CRDT instance, eventually, there is a state c formed by a sequence of partially ordered updates $c = \{u_1, u_2, \ldots, u_n\}$ s.t. $Q_{prospective}(c_i) = latest(Q_{stable}(\mathcal{R}))$ and $latest()$ returns the last item in the stable history.*

5 JANUS: RELIABLE CRDT MIDDLEWARE

5.1 Overview

Janus consists of three main components, the CRDT instance, the underlying consensus protocol, and the update auditor. Each replica of a Reliable CRDT instance has two state variables for stable and prospective values, namely prospective and stable states, identical to the CRDT's data structure. The consensus protocol asynchronously determines the set of valid updates and the order among updates to uphold the aforementioned Reliable CRDT guarantees. In our implementation, the underlying consensus is a DAG BFT protocol because it offers both strong ordering and BFT capability while serving as the mechanism for propagating prospective updates, as discussed in Section 2. The DAG also serves as the log of the update

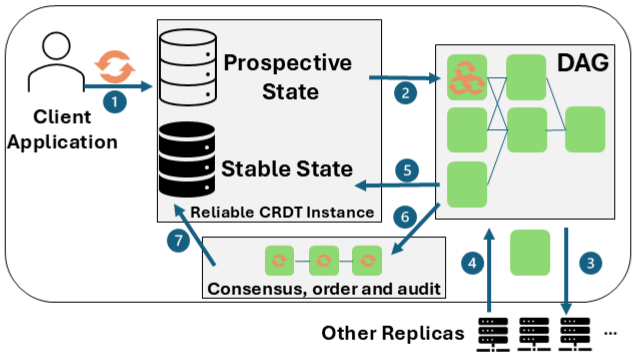


Figure 2: Workflow of *Janus*.

history for the auditing process. Although other consensus protocols (or non-BFT consensus protocols if the network assumption only allows crash failures) can be used here, a separate channel for prospective update propagation and a method of tracking the update history are required, which is less efficient.

An overview of *Janus*'s workflow is shown in Figure 2. For simplicity, we assume that one replica in the system contains all the components, including the client application, and communicates in a peer-to-peer fashion with other replicas. ① A client executes a CRDT update, and it is immediately reflected in the prospective state. ② The executed updates are asynchronously submitted to the DAG BFT protocol, and ③ the DAG is also used as the update propagation channel. ④ When a DAG block is received from another replica, ⑤ the replica executes the updates within the block on the prospective state. ⑥ When the consensus process commits a set of updates, they undergo *auditing* to check whether the updates are valid (i.e., not invariant-breaking), and ⑦ the updates are applied to the stable state. Additionally, the clients who issue the *safe* updates will be notified upon completion. Finally, the prospective state is *reversed* based on rejected invalid updates.

Janus strives for a balance between the performance of the CRDTs and the applicability of strong consistency. Compared with traditional consensus protocols, it allows the preemptive execution of updates on prospective states. In the absence of failures, the SEC properties of the CRDT enable prospective values to converge much more quickly than under strong consistency. Compared with plain CRDTs, it offers the ability to enforce strongly consistent guarantees on certain updates so that it can be utilized in a wider range of applications without requiring full transactional support while incurring only a relatively small computational overhead and enjoying the benefits provided by CRDTs.

5.2 Auditing Prospective States

It is possible for the preemptively executed updates on the prospective state to be invalid. To guarantee eventual validity, we introduce the concept of *auditing*[4], which retroactively undoes invalid updates after consensus is conducted by going through the DAG and checking if any received block is not committed. This process is performed independently on each replica at the time of a consensus commit.

[4]An analogy is that the tax department may collect taxes preemptively and then issue tax refunds after calculating the tax.

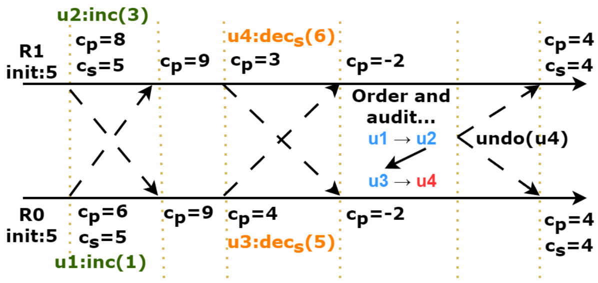


Figure 3: A Reliable CRDT counter under *Janus*: $inc()$ increments the counter by an integer, and $dec_s()$ decrements the counter by an integer and is designated *safe*; $c_p()$ are the values of the prospective state, and $c_s()$ are the values of the stable state at the given time.

We utilize the concept of *compensation* of *Reversible CRDTs* [31] to reverse invalid updates. As mentioned in Section 2.1, if one update's effect cancels out another update, the current state can be considered equivalent to the previous state. Note that the effect of compensation operations is dependent on the specific type, and must be predefined based on the use case. For example, the compensation for the *add* operation in a counter CRDT can be defined as the *subtraction* of the same value.

Figure 3 shows an example of how *Janus* enforces the invariant of a nonnegative counter. Starting with an initial value of 5 for two replicas[5], $\mathcal{R}_0$ increments by 1 and $\mathcal{R}_1$ increments by 3 concurrently. The prospective states are immediately updated and yield values of 8 and 6, respectively, but both stable states remain at 5. After the updates are propagated to both replicas, both prospective values are 9 (the stable state has not changed). Next, both replicas conduct safe decrement operations: $u3$ and $u4$. Although $u3$ and $u4$ do not break the invariant during local execution, their combined effect does, as it causes temporary negative values on the prospective states; therefore, one of the updates is not valid. After the updates are ordered by consensus, $u4$ is rejected, yielding the final stable value of 4. The prospective state consequently reverses the executed $u4$.

5.3 *Janus* Algorithms

The core design of *Janus* is illustrated by Algorithm 3, which follows the workflow in Figure 2. Variables s_state and p_state represent replicas for one or more CRDT instances. We consider a single instance here for simplicity.

Whenever an update is executed on a replica, it is immediately applied to the prospective state and submitted to the DAG instance (Line 4: $NewUpdate()$). If the update is annotated as *safe*, the replica waits until consensus is reached before responding to the client. Otherwise, the result is immediately returned after the update is executed locally.

Updates are then propagated through the DAG blocks. When a replica receives a block from another replica in a round, it immediately executes all updates within the blocks on the prospective states (Line 12: $OnAcceptingBlock()$). We do not explicitly show the process of block generation here since it is part of the DAG algorithm.

[5]It can be assumed that the total number of replicas is sufficient to meet the $3f + 1$ requirement.

Algorithm 3 *Janus* Algorithm

```
1:  Variables: s_state, p_state: CRDT state ▷ Stable and prospective states
2:  Variables: dag                          ▷ DAG BFT protocol instance
3:  Variables: stable_history       ▷ Append-only log of stable values
4:  function NewUpdate(u : Update, isSafe: bool)
5:      exectuion_result ← p_state.execute(u)
6:      if exectuion_result then
7:          consensus_result : awaitable ← dag.propose(u) ▷ Asyn-
            chronously send the update to the DAG instance
8:      if isSafe then
9:          return await consensus_result ▷ Caller can wait on the con-
            sensus result
10:     else
11:         return exectuion_result

12: function OnAcceptingBlock                        ▷ On new DAG blcok
13:     b : Block ← dag.get_latest_block()
14:     for all u ∈ b.updates do         ▷ Execute all updates in the block
15:         p_state.execute(u)

16: function OnCommit              ▷ When the consensus commits updates
17:     to_reverse ← Audit()
18:     p_state.reverse(to_reverse);
19:     stable_history.append(s_state)
20:     Send consensus_result to all waiting safe updates
```

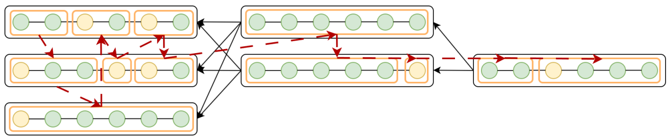


Figure 4: Ordering CRDT updates of committed DAG blocks: the safe updates are marked in red, and the orange boxes are update groups. The order of execution is shown with red arrows.

When the consensus decides to commit on Line 16: $OnCommit()$, the replica uses the $Audit()$ (Line 1) function to determine whether some updates need to be reversed. During the audit process, the replica reverses these updates on the prospective state while updating the stable state, as shown in Algorithm 4.

The audit process can be seen as part of the commit process. Auditing starts by ordering the received updates (Line 9: $Order()$) when the consensus process delivers a series of causally related blocks containing sets of updates. The idea is to replay the update executions on the stable state with the same causal ordering when they are executed on the prospective state so that any application-defined dependency is preserved (the dependent updates are also causally ordered).

To maintain the causal dependencies of the executed updates when ordering updates, we bundle regular I-confluent updates within a block into *update groups* because they can be considered concurrent. Then, we traverse the blocks in a breadth-first manner from the earliest round in the wave and move in a round-robin fashion, as shown in Figure 4. This process is deterministic, so replicas generate the same ordering independently.

Then, updates are executed and checked against invariants to determine if they need to be reversed on the prospective state. If an update violates an invariant, it is added to the *to_reverse* list. The *to_reverse* list also includes the updates that are invalidated by consensus, such as conflicting updates from faulty replicas identified by checking the blocks that are not committed by the consensus process. Finally, these updates are reversed on the prospective state. Note that the compensation operations are conducted independently on each replica. They can be viewed as the complement set of the stable state in all updates on a replica, and reversing them equates the prospective state to the stable state.

Algorithm 4 Order and Audit

```
function Audit
    $to\_reverse$ : $List$;
    while $u$: $Order().Dequeue()$ do
        $result \leftarrow s\_state.try\_execute(u, InvariantCheck)$); ▷ Execute on the stable state; if this breaks the invariant, return false and leave the state unchanged
        if !$result$ then
            $to\_reverse.add((u))$
    $to\_reverse.add(\forall u \in dag.uncommited\_block())$ ▷ Uncommited blocks due to faulty or retransmission
    return $to\_reverse$

function Order
    $execution\_order$: Queue
    for $i \in 0..number\_of\_rounds\_in\_a\_wave$ do ▷ Traverse from the earliest updates while following the causal order
        loop
            $j \leftarrow 0$
            $b$: Block $\leftarrow dag.blocksToCommit[i][j]$
            while $\neg b.updates.empty()$ do
                $execution\_order.enqueue(b.updates.pop())$
                if $b.updates.isSafe$ then
                    $j{+}{+}$ if $dag.blocksToCommit[j+1] \neq null$ else $j = 0$
    return $execution\_order$;
```

With state-based CRDTs, we implement an optimization method called *state compaction* that combines updates of the same CRDT instance in a block into one state as the update message for propagation to reduce the message size of each block. This can be done because later states implicitly encapsulate all previous states when the state is disseminated [39].

5.4 Correctness

In this section, we show that *Janus* provides the guarantees of Reliable CRDTs. Many safety preconditions are already satisfied by the consensus protocol (agreement, validity, and termination) [47] and the SEC guarantees of CRDTs [39], so we do not explicitly prove them here.

Lemma 1. *During the execution of the OnCommit() function, the same set of updates is processed on all correct replicas.*

According to the agreement property of the underlying consensus protocol, the same set of blocks is used for all correct replicas at each wave. Each block must contain the same set of updates (messages) because they cannot be forged since all messages are signed; thus, the same set of updates must be performed on all correct replicas, as this property follows directly from the DAG consensus protocol [18].

Theorem 1. *Querying $stable\_history$ in Janus satisfies the properties of the stable history in Definition 1.*

Proof. The $stable\_history$ consists of a series of states s_state, which are added to every commit in Line 18 of Algorithm 3.

Each s_state is obtained by applying the same set of updates. If the same set of updates is committed on all correct replicas, by Lemma 1 and the *strong convergence* property of CRDTs, the equivalent s_state must be generated for all correct replicas at each commit, resulting in a consistent $stable\_history$.

Since every update executed on s_state is checked against the invariants on Line 4 of Algorithm 4, s_state must satisfy all invariants. Therefore, $stable\_history$ satisfies the stable history property in Definition 1. □

Theorem 2. *If $isSafe$ is set to true for an update, then this update satisfies the properties of safe updates in Definition 3.*

Proof. In Line 9 of Algorithm 3, if the $isSafe$ argument is set to true for an update, it will not return an execution result until the update is committed by the consensus process. If the execution is successful, it must be included in the stable states, and by Theorem 1, it must be included in the stable histories of all correct replicas.

Since the commit process orders all updates deterministically, the set of safe updates must be totally ordered. Therefore, safe updates satisfy the properties of safe updates given in Definition 3. □

Theorem 3. *Janus satisfies Byzantine safety in Definition 5.*

Proof. We show this by proving that the no conflicting update in Definition 4 is included in the $stable\_history$.

Assume that there is a conflicting update that executes on two correct replicas at the same position in the linear order of updates but yields two stable states, namely, s_i and s_i'. According to Theorem 1, the same stable state must be equivalent for all correct replicas. However, according to the SEC, if the conflicting updates have different content, the stable states must be different. Therefore, conflicting updates cannot be included in the stable history by contradiction. □

Lemma 2. *The to_reverse set on a correct replica captures all updates that are not executed on s_state.*

Proof. Since all updates are submitted to a DAG block or received from a block, an update must be either committed or rejected by the consensus process. If an update is committed and it is not executed on s_state, then it must be added to the *to_reverse* set on Line 6 of Algorithm 4.

If the block is not committed, it may never be executed on s_state; it is also added to the $to\_reverse$ set on Line 7. Therefore, the *to_reverse* set captures all updates that are not executed on the s_state. □

Theorem 4. *Janus satisfies the properties of eventual validity with s_state and p_state.*

Proof. p_state executes all received updates that either come from the user or are received by a block, and they are the same for all correct replicas according to Lemma 1. Assuming that all of the updates are delivered eventually, by Lemma 2, $s\_state \cup to\_reverse = p\_state$.

The consistency properties provided by reversible CRDTs can cancel all effects from updates in $to\_reverse$ [31]. Therefore, s_state will eventually be equivalent to p_state because $E(\sum p\_state) - E(\sum to\_reverse) = E(\sum s\_state)$ according to the properties of the compensation operations. □

5.5 Complexity Analysis

The complexity of *Janus* is primarily determined by the underlying BFT protocol (affecting messaging complexity) and the CRDT operations (affecting message size) since they are responsible for communication and updating the data. For Narwhal & Tusk, the messaging complexity given m nodes and n updates is $O(|n|m^3 \log m)$ [18, 47].

Our algorithms impact the processes of applying received updates, ordering and auditing updates. Since there are two copies of a CRDT instance, prospective and stable states, the time complexity of applying n updates for the worst-case (assuming all updates are applied to both states) is $O(2n) = O(n)$, see Line 14 of Algorithm 3 and Line 3 of Algorithm 4.

For ordering updates, the worst-case complexity is $O(n)$, since we are only ordering all the updates that are received at Line 14 of Algorithm 3 (there can be at most n updates in the received blocks) and best-case $O(1)$ if there is nothing committed to order. For reversing invalid updates, the worst-case complexity is $O(n)$, because only at most n originally executed updates need to be reversed and best-case $O(1)$ if there is nothing to reverse.

6 EVALUATION

In this section, we evaluate a proof-of-concept implementation of *Janus*; its source code is publicly available on GitHub [1]. Our implementation includes a distributed key–value database server in which each key–value pair is a CRDT object based on an open-source CRDT library in C# [32], and the *Janus* middleware with a C# port of the core DAG algorithm described in Narwhal & Tusk [18].

The existing implementation of Narwhal is not used because comparing CRDTs and the DAG algorithm with the same code base allows us to better investigate their performance characteristics. Furthermore, Narwhal is developed as part of the Sui blockchain [41], which is not compatible with a CRDT library.

6.1 Experimental Setup

We deploy a *Janus* cluster on Ubuntu 22.04 VMs running .Net 8, hosted on the Compute Canada cloud. Each VM has two 2.4 GHz Intel Xeon vCPUs (Broadwell) and 15 GB of RAM, and the VMs are interconnected by a high-speed LAN with less than $1ms$ latency and 3 Gbps bandwidth. Each VM hosts one server node, and the clients run on a separate VM but in the same cloud region.

We create microbenchmarks with synthetic workloads and implement the banking database application according to the example in Section 5 for a realistic evaluation.

6.2 Microbenchmarks

We use synthetic workloads because existing database benchmarks, such as TPC-C and YCSB do not directly support CRDTs. A workload consists of a series of randomly generated updates (safe and regular) and read requests.

Two common state-based CRDTs are used for evaluation: PN-Counter and OR-Set [38]. PN-Counter is an integer counter that can add or subtract integers. Its value is represented by the difference between the sums of two cluster-size-length vectors (positive and negative). Updating the counter involves adding to the value in the positive vector (for addition) or the negative vector (for subtraction) at the source replica's index. Propagating updates requires sending both vectors, and merging involves an elementwise max operation on each vector. We measure an average size of 380 bytes for PN-Counter's synchronization messages.

OR-Set is a more complex CRDT whose state consists of two sets: an added set and a removed set. When an element is added, it is associated with a unique tag and added to the added set. When an element is removed, the tag of the observed element is added to the removed set, ensuring add-win semantics during concurrent add-removes. Only the element with its tag in the added set and not in the removed set is considered part of the set. Propagating updates requires sending both sets, and merging involves taking the union of both sets. The average message size is $2.4KB$ with our experimental setup[6].

We compare our results to two baseline approaches, plain CRDTs where messages are directly disseminated among nodes and applying BFT consensus protocols on *all* CRDT messages. For the latter, we select two protocols: a traditional partially synchronous BFT protocol HotStuff [45] with message size set to 380 bytes and 2.4KB to mimic conducting consensus on all CRDT messages via HotStuff, and our implementation of Narwhal & Tusk using our KVDB by agreeing on all updates with the consensus before they are applied.

Metrics and Parameters. The experiments are conducted by initializing multiple client threads that continuously send requests to the servers. Each client thread sends a new request only after receiving the response from the previous request. The two main metrics are overall throughput in operations per second (ops/s), which represents the total number of processed operations from clients measured on the servers (excluding any reverse operations), and end-to-end latency in milliseconds (ms) measured on the clients.

We adjust the following parameters in the workloads for different experiments: *Safe ratio* is the percentage of safe updates out of all update operations. *Access pattern* is the ratio of read operations to all update operations. The operations are uniformly distributed among all the objects. Note that all reads are prospective since reading either stable or prospective values involves accessing only the local states, resulting in the same performance impact. *Batch size*, denoted as b, is the number of CRDT updates grouped into a single client message that is supplied to the consensus. The state compaction optimization method mentioned in Section 5.3 is applied during batching. *Sending rate* refers to the target throughput for the clients to send requests. We also vary the *number of objects* and *number of nodes* in the cluster.

Overall Performance. Figure 5 shows the latency of operations when the systems are under different loads (as seen in the throughput); greater loads are represented by increasing the sending rate of the clients. Each sample point is the mean of 5 runs with a fixed

[6]Because of memory constraints, each OR-Set instance is deleted after reaching 50 elements, then a new instance is created.

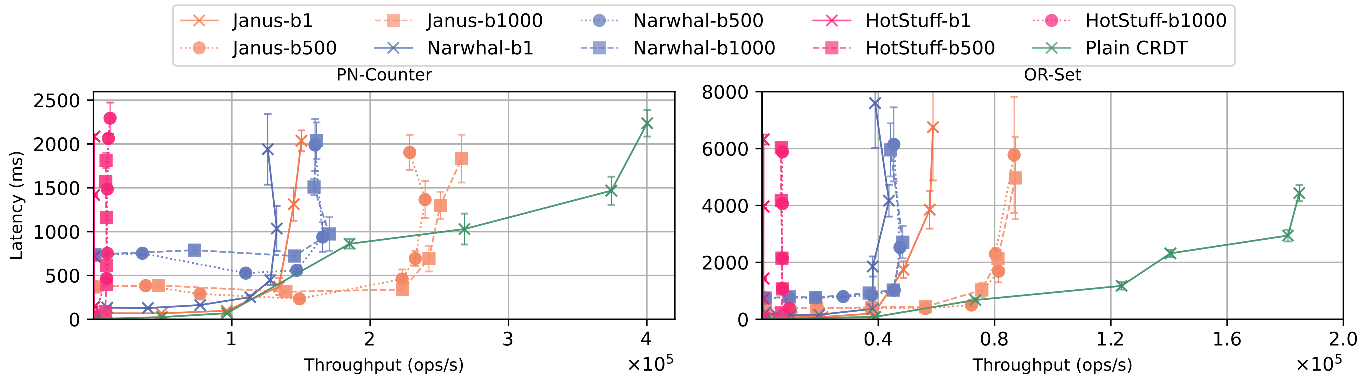


Figure 5: Throughput vs. latency for different batch sizes b, balanced access patterns, 100 objects and 4 nodes. HotStuff is evaluated with message size set to the average message size of PN-Counter and OR-Set.

sending rate, and the error bars represent one standard deviation in average latency.

For PN-Counter, *Janus* has a peak throughput (indicating the system's capacity when saturated) of approximately $260,000$ ops/s at $b = 1,000$, which is $1.5\times$ the peak of the Narwhal at $170,000$ ops/s and $21\times$ the peak of HotStuff at only $12,000$ ops/s. This shows the effectiveness of *Janus* over regular protocols and non-CRDT solutions. However, it is only $0.65\times$ the peak throughput of plain CRDTs, showing the performance trade-off of the Reliable CRDT features. The latency of HotStuff and plain CRDTs is much lower when the load is light, approximately $30ms$ and $3ms$, respectively, but the advantage quickly diminishes.

Increasing the batch size for *Janus* from 1 to 500 increases the peak throughput by $2.3\times$. This shows the effectiveness of the state compaction optimization, but increasing it to $1,000$ does not yield further improvement. In addition, the batch size does not affect the performance of Narwhal as much as that of *Janus*. The low-load latency is also increased $2.3\times$ due to the time spent filling the batch.

For OR-Set, both *Janus* and Narwhal have a lower peak throughput because of the much larger message size and more complex merging process. *Janus* has a peak throughput of about $80,000$ ops/s. The $5\times$ increase in message size from PN-Counter to OR-Set results in a 70% decrease in throughput. However, *Janus* is still $1.6\times$ faster than Narwhal and $11\times$ faster than HotStuff at only $7,000$ ops/s, but has only $0.4\times$ the throughput of a plain OR-Set CRDT.

Number of Objects. Figure 6 illustrates the peak throughput of the systems as the number of objects ranges from 10 to $5,000$. For PN-Counter, there is no significant performance variation based on the number of objects, which is expected because increasing the number of objects does not affect message complexity. However, for OR-Set, the peak throughput decreases when the number of objects exceeds $2,000$. This decline occurs because OR-Set is a significantly more complex data structure than PN-Counter and grows in size when new elements are added. The increase in object counts reduces the frequency of resetting OR-Set instances[6] and exhausts the node's memory, leading to frequent memory swapping to the disk and reduced performance.

Message Size. We evaluate the direct impact of the message size with OR-Set by removing the 50-element-cap[6] for each OR-Set

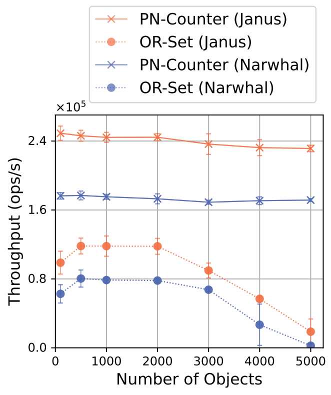


Figure 6: Peak throughput of different number of objects, a balanced access pattern, 50% safe updates, $b = 500$ and 4 nodes.

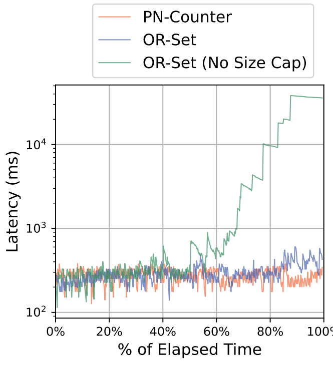


Figure 7: Light-load Latency of safe updates overtime in log scale, a balanced access pattern, 50% safe updates, $b = 500$, 1000 ops/s sending rate, 100 objects and 4 nodes.

instance in this specific experiment. Figure 7 shows the end-to-end latency of safe updates on a single client over a 60-second experiment, with the sending rate set to $1,000$ ops/s, resulting in a lightly loaded system. For PN-Counter and OR-Set instances with a 50-element cap, latency remains stable around $100 - 200$ ms, as the states of the CRDTs do not grow indefinitely, nor the message size.

However, with uncapped OR-Set, we observe that message size increases significantly from 144 bytes to a staggering 196 MB at their largest, as each OR-Set instance contains over $4,000$ elements and causes the latency to rise substantially. Additionally, the latency increase becomes stepped after a certain point, likely due to the longer time required to receive larger messages, resulting in a large number of updates being completed simultaneously, followed by a prolonged wait for the next set of updates.

Access Pattern and Safe Ratio. We vary the ratios from read-heavy (25% update/75% read) to write-heavy (75% update/25% read) access patterns. Then, we measure the peak throughput of each combination.

As shown in Figure 8, the write-heavy workloads are more sensitive to the ratio of safe updates than are the read-heavy workloads,

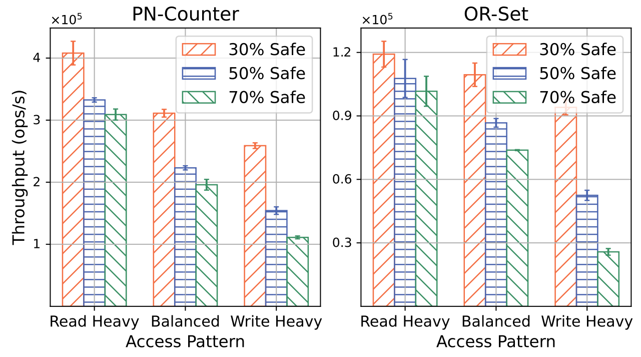


Figure 8: Peak throughput of different access patterns and safe update ratios at $b = 500$, 100 objects and 4 nodes.

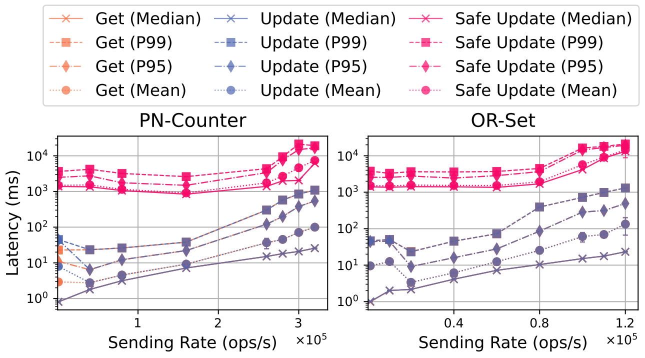


Figure 9: Latency of operations in log scale at $b = 500$ with a balanced access pattern, 50% safe updates, 100 objects and 4 nodes.

since only updates are affected by the consensus process. In addition, the decrease is more significant when the ratio of safe updates increases from 30% to 50% and then from 50% to 70%. The trend is similar for both PN-Counter and OR-Set.

Latency of Operations. Figure 9 depicts the latency changes of different operations when increasing the sending rate of the clients. We measure the mean, median, 95^{th} and 99^{th} percentile latencies for safe updates, regular updates, and reads.

There is a large discrepancy between safe and regular operations for both CRDTs, as expected, due to the heavy consensus process. There is only a negligible difference between regular operations and reads in terms of latency because they are affected only by the local computational efficiency. For both PN-Counter and OR-Set, the latency of reading a value and conducting a local CRDT update increases from a few milliseconds to approximately $100ms$. Safe updates take more than $1,000ms$, increasing to more than 10 seconds when the system becomes saturated, which is still much greater than even the outliers of regular operations.

This shows that even though the consensus process performs poorly when the system is saturated, it can still handle some regular operations within a reasonable time frame, which can be further proven by the increasing gap between the 95^{th} and 99^{th} percentiles and the average latency of regular operations. This is because there are more outliers in terms of latency under a heavy load, but at least half of the operations are still processed promptly, so the median is low.

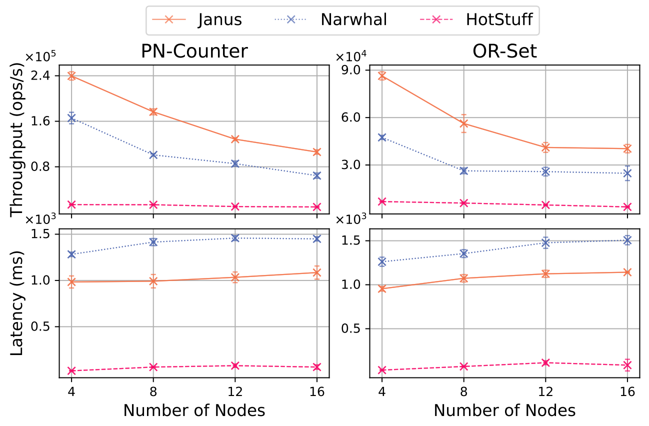


Figure 10: Peak throughput and latency for varying numbers of nodes, $b = 500$, a balanced access pattern, 50% safe updates and 100 objects.

Scalability. The scalability of the systems is shown in Figure 10. We increase the number of replicas from 4 to 16 and measure the peak throughput and the average latency when the systems are lightly loaded.

For PN-Counter, the decrease in peak throughput is almost linear with the number of nodes, from $240,000$ ops/s to $110,000$ ops/s. However, after 8 nodes, the rate of decline slows, especially for Narwhal. This effect is more obvious with OR-Set, where the peak throughput of both *Janus* and Narwhal decreases by approximately 40% when the number of nodes increases from 4 to 8. However, throughput barely changes when the number of nodes increases from 12 to 16. For HotStuff, the throughput with PN-Counter sized messages decreases approximately 30% from $13,000$ ops/s to $9,000$ ops/s from 4 to 16 nodes. For OR-Set sized messages, the decrease is approximately 50%. The throughput scaling of our solution is comparable to HotStuff, despite *Janus* having a much higher absolute value.

The cluster size only marginally affects the latency for *Janus*. For PN-Counter, the latency increases by approximately $150ms$ in moving from 4 to 16 nodes, and for OR-Set, the latency increases by approximately $200ms$. This indicates that the communication overhead is no longer the bottleneck of the system after a certain number of nodes is reached. For HotStuff, latency increases from $20ms$ to $60ms$ with PN-Counter sized messages, and from $30ms$ to $85ms$ for OR-Set sized message. The advantage in latency of HotStuff shows the overhead in the DAG protocols once again.

Performance under Faults. The performance of the systems with a total of 8 nodes under failure is depicted in Figure 11. For crash failures, we stop the processes of random *Janus* nodes at the beginning of each experiment run, and the leader nodes for HotStuff halfway through each run. The throughput of *Janus* improves 50% from approximately $125,000$ ops/s to $190,000$ ops/s when the number of crashed nodes increases from 0 to 2. This is because crashed nodes do not send many messages, thus reducing the messaging complexity. For HotStuff, the performance decreases significantly

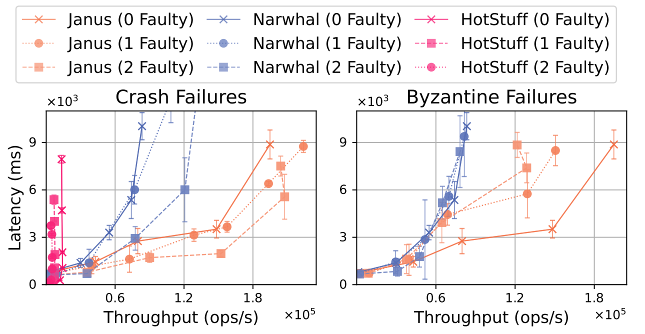
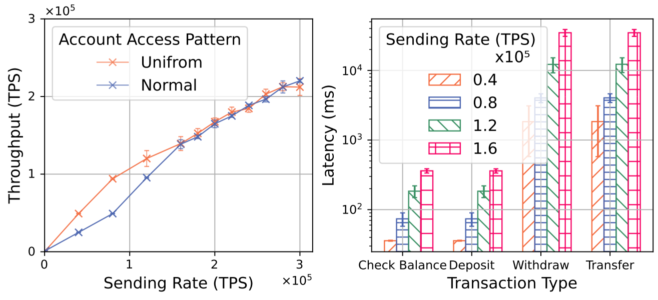


Figure 11: Throughput vs. latency for varying numbers of faulty nodes out of 8 nodes, $b = 500$, a balanced access pattern and 100 PN-Counter objects.



(a) Throughput vs. Request Rate **(b) Transaction Latency**

Figure 12: Throughput and latency of the banking database with 4 nodes and 100 accounts.

by approximately 60% with an already low initial throughput of approximately 13,000 ops/s because stopping the leader introduces a time-consuming view-change process, which is not a problem in DAG BFT protocols because there is no leader. However, if the crashed node is not the leader, then we do expect a performance improvement [48].

To model Byzantine failures, we let 50% of the updates propagated by the faulty replicas to be invalid by not creating certificates for half of the blocks. HotStuff is not measured here because its library does not support the manual injection of conflicting messages, and replicas cannot propose conflicting messages unless the leader is faulty, which would have the same performance impact as a view change caused by a crash failure. Conflicting updates and reverse operations to undo them in the prospective state are not included in the throughput measurement.

The results are shown in Figure 11 (right), where the invalid updates are excluded from the throughput measurement. The peak throughput for *Janus* decreases by around 20%, from 190,000 ops/s to 150,000 ops/s. This performance degradation is expected due to the increased number of updates that need to be reversed. Narwhal's performance is unchanged, as all updates are agreed upon by the consensus before they are applied, so there is no invalid update requiring reversal. However, even with two faulty replicas, *Janus* still outperforms Narwhal.

6.3 Banking Database Performance

We implement the banking database example as a distributed application that utilizes *Janus* with 4 types of transactions: deposit, withdraw, transfer money, and check balance. Each account is represented by a PN-Counter object with an invariant check for non-negative values.

The same setup from previous experiments is used, but a 50ms delay and 10ms jitter are added among the VMs by using *netem* and *tc* tools to emulate a geo-replicated cluster connected by a WAN within North America. The experiments are conducted with 100 accounts, a balanced workload with 50% balance checks and 50% update transactions where 50% of update transactions are withdrawals and the other 50% are transfers.

We also measure the performance for uniform and normally distributed access patterns on accounts. To induce concurrency, we increase the number of transactions sent concurrently by each client per second. Throughput is measured on the client side as the number of completed transactions received by clients per second (TPS).

Figure 12 shows that the throughput of the system increases linearly with the concurrency rate without any significant performance degradation. Only after 250,000 TPS does the trend slightly slow. This is because the consensus process becomes extremely slow under a heavy load, and the system has the opportunity to process more nonsafe updates and read requests. The reading and deposit latency increase approximately 10× when the sending rate increases from 40,000 to 160,000 TPS, but the latency for transactions involving safe updates increases approximately 18×.

In addition, the uniformly distributed access pattern yielded slightly better results than those of the normal distribution at a low sending rate. This finding indicates that contested objects affect performance because of lock contention. However, when the load increases, this effect becomes negligible, as other factors overshadow locking overhead.

6.4 Execution Time Analysis

We profile the system for 60 seconds during an experiment run when the system was saturated with *dotnet-trace* tool. The breakdown for the cost of each operation is shown in Figure 13 for PN-Counter.

We note that the majority (52.3%) of total CPU time was used for applying CRDT updates on the stable and prospective state. Furthermore, the implementation of the CRDT is not optimal where its states are serialized into JSON strings before they are transmitted for updates, resulting in a 24.1% of the total CPU time spent on serializing and deserializing the JSON.

It is worth noting that tracking safe updates took a significant 22.3% of the total CPU time. We use a *ConcurrentDictionary* class to map between the updates and the client that requested them, indicating that the dictionary is not optimized for highly contested operations.

Finally, The communication among nodes is conducted with Protobuf, which consumed a significant 13.2% of the total CPU time. The DAG operations, such as handling blocks and traversing the graph did not consume a significant amount of CPU time, as they were only a part of the remaining 12.3% of the total CPU time.

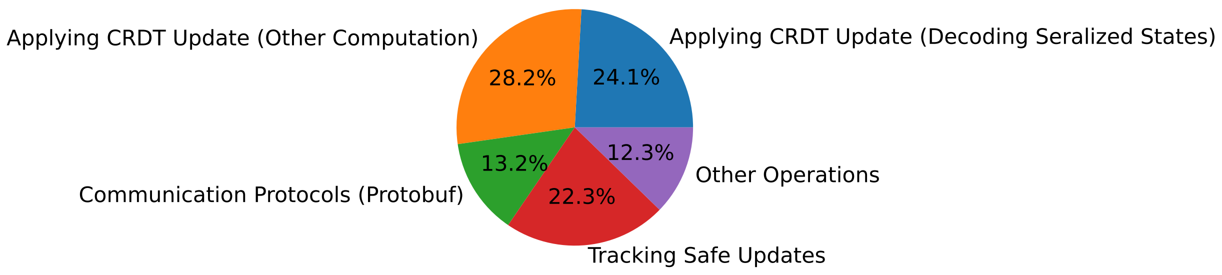


Figure 13: CPU time breakdown for each component during an experiment run with PN-Counter, balanced access patterns, 50% **safe updates,** $b = 500$ 100 **objects and** 4 **nodes.**

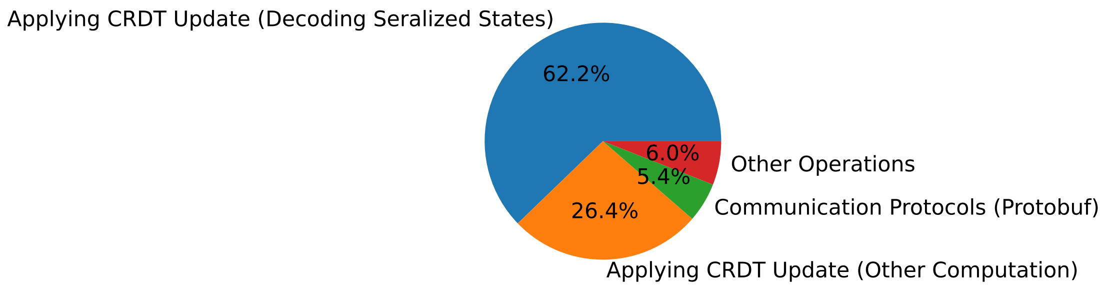


Figure 14: CPU time breakdown for each component during an experiment run with OR-Set, balanced access patterns, 50% **safe updates,** $b = 500$ 100 **objects and** 4 **nodes.**

For OR-Set, the CPU time breakdown is shown in Figure 14. Applying CRDT updates took over 80% of the total CPU time, showing the bottleneck of the current implementation is in the CRDT library.

6.5 Discussion

Our evaluations show that *Janus* has a significant advantage over traditional BFT protocols by allowing the eager execution of operations. Furthermore, the preemptive execution of regular updates and reads can reduce perceived latency and improve availability, as these operations can be executed locally and respond to the client promptly, even under heavy loads.

Since message size is a considerable factor that affects performance, a future direction for optimization could be to further reduce the messages size by using operation-based and delta-based CRDTs [3] or by using consensus algorithms that are more efficient for larger payload sizes.

7 RELATED WORK

Dynamically adjusting consistency levels for various application requirements and network conditions is a common approach that attempts to provide strong consistency without degrading performance [7, 8, 29, 36]. There have also been efforts to combine different consistency levels into one system [2]. For example, RedBlue consistency [28] allows users to mark operations as *red* or *blue*. Red operations are executed in a strongly consistent (serialized) manner, and blue operations are executed in an eventually consistent manner, so the system can be both fast and consistent. Red updates are similar to safe updates in our work. However, CRDTs also consider states, not only the order of operations; therefore, in our work, the ordering of updates is not the goal but rather a way to provide stronger semantics for CRDTs in our programming model.

Rationing consistency permits different consistency levels to coexist in the same system [25], but the granularity is at the level of data objects instead of the whole system. The transactional application protocol for inconsistent replication (TAPIR) [49] defines two kinds of operations: inconsistent and consensus. In this scheme, replicas execute inconsistent operations independently while using consensus operations to determine a value that is agreed upon by all replicas with fault tolerance. AntidoteDB [5, 30] is a distributed database that offers APIs for users to choose which consistency level is most appropriate based on the execution context. However, none of these methods consider Byzantine failures.

For CRDTs, efforts have been made to combine different aspects of CRDTs with strongly consistent systems, such as blockchains. These endeavors aim to either support features commonly found in strongly consistent systems while preserving the performance advantages of eventually consistent systems or to use CRDTs to address the performance drawbacks of strong consistency [34]. For example, OrderlessChain [33] replaces ordering in blockchains with CRDTs that also allow for invariant checks. The Vegvisir blockchain [21] uses a DAG chain instead of a linear chain, and branching chains are merged via CRDTs. Both of these methods allow CRDT-based applications to benefit from the additional safety guarantees offered by blockchains, such as Byzantine fault tolerance and immutability, while improving the performance of blockchains by allowing concurrent operations. However, neither provides new semantics for CRDTs compared with those of Reliable CRDTs.

To combat Byzantine failures in CRDTs, optimistic Byzantine fault tolerance [51] combines CRDTs with BFT consensus by requiring all updates on a replica to be agreed upon via a BFT consensus protocol before they are propagated (while local execution is conducted eagerly) through a checkpoint mechanism. This allows the replicas to detect Byzantine failures and eliminate *conflicting messages*. The same authors (2013, 2016) presented similar approaches for enhancing CRDTs used in collaborative editing tools [52, 53]. However, these works do not consider the possibility of eager execution on all replicas and use traditional leader-based BFT, which may lead to a reduction in performance.

Kleppmann (2022) [24] proposed a novel approach to address Byzantine failures in peer-to-peer (P2P) CRDT systems where the number of untrusted nodes is arbitrary. The focus is on providing a reliable broadcast mechanism for P2P systems. This approach uses a hashgraph (a DAG in which the vertices reference previous vertices using cryptographic hashes) to ensure the causal order and integrity of the updates. Barbosa et al. (2021) [11] and Jannes et al. (2022) [20] suggested methods to combat Byzantine failures in CRDTs by ensuring the privacy and security of CRDT protocols; however, they did not focus on the consistency of the CRDTs.

In addition to DAG-based consensus, several works on BFT protocols endorse concurrency. For example, in Basil [42], instead of using a totally ordered BFT log as in traditional consensus, it optimistically handles concurrent operations on the server side and relies on clients to ensure safety and maintain an illusion of serializability. Similar concepts can be found in Pompe [50] and V-Guard [46], where multiple consensus instances can occur at the same time. Although these consensus protocols could also be applied as the underlying consensus for *Janus*, DAG BFT protocols align better with our design.

8 CONCLUSIONS

In this paper, we introduce the novel concept of Reliable CRDTs, which enhance eventually consistent CRDTs in a Byzantine environment to obtain guarantees typically associated with strongly consistent systems, such as strong ordering of operations and invariant preservation. We discuss how Reliable CRDTs can be used to build fast and reliable distributed applications. Finally, we present *Janus*, an implementation of a Reliable CRDT system that outperforms consensus-only solutions and demonstrates comparable performance to plain CRDTs. It is publicly available at [1].

Future research directions based on this work include further studies on the performance optimization of *Janus* and exploring the broader applicability of Reliable CRDTs. For instance, Reliable CRDTs could serve as a cost-effective method for enhancing reliability in trusted environments, such as data centers, by using similar delayed validation techniques.

REFERENCES

[1] [n.d.]. Janus Prototype. https://github.com/Romsdal/Janus-CRDT.

[2] Hesam Nejati Sharif Aldin, Hossein Deldari, Mohammad Hossein Moattar, and Mostafa Razavi Ghods. 2019. Consistency models in distributed systems: A survey on definitions, disciplines, challenges and applications. *CoRR* abs/1902.03305 (2019). arXiv:1902.03305 http://arxiv.org/abs/1902.03305

[3] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. 2015. Efficient state-based crdts by delta-mutation. In *International Conference on Networked Systems*. Springer, 62–76.

[4] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, Srinivasan Muralidharan, Chet Murthy, Binh Nguyen, Manish Sethi, Gari Singh, Keith Smith, Alessandro Sorniotti, Chrysoula Stathakopoulou, Marko Vukolić, Sharon Weed Cocco, and Jason Yellick. 2018. Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 30, 15 pages. https://doi.org/10.1145/3190508.3190538

[5] AntidoteDB. 2021. Antidote: A planet scale, highly available, transactional database built on CRDT technology. https://github.com/AntidoteDB/antidote.

[6] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014. Coordination Avoidance in Database Systems. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 185–196. https://doi.org/10.14778/2735508.2735509

[7] Peter Bailis and Ali Ghodsi. 2013. Eventual Consistency Today: Limitations, Extensions, and Beyond. *Commun. ACM* 56, 5 (May 2013), 55–63. https://doi.org/10.1145/2447976.2447992

[8] Peter Bailis, Ali Ghodsi, Joseph M Hellerstein, and Ion Stoica. 2013. Bolt-on causal consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 761–772.

[9] Leemon Baird. 2016. The Swirlds Hashgraph Consensus Algorithm: Fair, Fast, Byzantine Fault Tolerance. *Swirlds Tech Reports SWIRLDS-TR-2016-01, Tech. Rep* 34 (2016), 9–11.

[10] Valter Balegas, Diogo Serra, Sérgio Duarte, Carla Ferreira, Marc Shapiro, Rodrigo Rodrigues, and Nuno M. Preguiça. 2015. Extending Eventually Consistent Cloud Databases for Enforcing Numeric Invariants. In *34th IEEE Symposium on Reliable Distributed Systems, SRDS 2015*. IEEE Computer Society, 31–36. https://doi.org/10.1109/SRDS.2015.32

[11] Manuel Barbosa, Bernardo Ferreira, João Marques, Bernardo Portela, and Nuno Preguiça. 2021. Secure Conflict-Free Replicated Data Types. In *International Conference on Distributed Computing and Networking 2021* (Nara, Japan) *(ICDCN '21)*. Association for Computing Machinery, New York, NY, USA, 6–15. https://doi.org/10.1145/3427796.3427831

[12] Eric Brewer. 2012. CAP twelve years later: How the "rules" have changed. *Computer* 45, 2 (Jan. 2012), 23–29. https://doi.org/10.1109/MC.2012.37

[13] Lucas Brutschy, Dimitar Dimitrov, Peter Müller, and Martin Vechev. 2017. Serializability for Eventual Consistency: Criterion, Analysis, and Applications *(POPL '17)*. Association for Computing Machinery, New York, NY, USA, 458–472. https://doi.org/10.1145/3009837.3009895

[14] Johannes Buchmann. 2004. *Introduction to cryptography*. Vol. 335. Springer.

[15] Christian Cachin, Klaus Kursawe, Frank Petzold, and Victor Shoup. 2001. Secure and Efficient Asynchronous Broadcast Protocols. In *Advances in Cryptology — CRYPTO 2001*, Joe Kilian (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 524–541.

[16] Christian Cachin, Klaus Kursawe, and Victor Shoup. 2000. Random Oracles in Constantipole: Practical Asynchronous Byzantine Agreement Using Cryptography (Extended Abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing* (Portland, Oregon, USA) *(PODC '00)*. Association for Computing Machinery, New York, NY, USA, 123–132. https://doi.org/10.1145/343477.343531

[17] George F Coulouris, Jean Dollimore, and Tim Kindberg. 2005. *Distributed systems: concepts and design*. pearson education.

[18] George Danezis, Lefteris Kokoris-Kogias, Alberto Sonnino, and Alexander Spiegelman. 2022. Narwhal and Tusk: A DAG-based Mempool and Efficient BFT Consensus. In *Proceedings of the Seventeenth European Conference on Computer Systems*. ACM, Rennes France, 34–50. https://doi.org/10.1145/3492321.3519594

[19] Adam Gkagol, Damian Leundefinedniak, Damian Straszak, and Michał undefinedwiundefinedtek. 2019. Aleph: Efficient Atomic Broadcast in Asynchronous Networks with Byzantine Nodes. In *Proceedings of the 1st ACM Conference on Advances in Financial Technologies* (Zurich, Switzerland) *(AFT '19)*. Association for Computing Machinery, New York, NY, USA, 214–228. https://doi.org/10.1145/3318041.3355467

[20] Kristof Jannes, Bert Lagaisse, and Wouter Joosen. 2022. Secure Replication for Client-Centric Data Stores. In *Proceedings of the 3rd International Workshop on Distributed Infrastructure for the Common Good* (Quebec, Quebec City, Canada) *(DICG '22)*. Association for Computing Machinery, New York, NY, USA, 31–36. https://doi.org/10.1145/3565383.3566111

[21] Kolbeinn Karlsson, Weitao Jiang, Stephen Wicker, Danny Adams, Edwin Ma, Robbert van Renesse, and Hakim Weatherspoon. 2018. Vegvisir: A Partition-Tolerant Blockchain for the Internet-of-Things. In *38th International Conference on Distributed Computing Systems* (Vienna, Austria) *(ICDCS 2018)*. IEEE Computer Society, 1150–1158. https://doi.org/10.1109/ICDCS.2018.00114

[22] Idit Keidar, Eleftherios Kokoris-Kogias, Oded Naor, and Alexander Spiegelman. 2021. All You Need Is DAG. In *Proceedings of the 2021 ACM Symposium on Principles of Distributed Computing*. ACM, Virtual Event Italy, 165–175. https://doi.org/10.1145/3465084.3467905

[23] Martin Kleppmann. 2016. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems.* O'Reilly.

[24] Martin Kleppmann. 2022. Making CRDTs Byzantine Fault Tolerant. In *Proceedings of the 9th Workshop on Principles and Practice of Consistency for Distributed Data* (Rennes, France) *(PaPoC '22)*. Association for Computing Machinery, New York, NY, USA, 8–15. https://doi.org/10.1145/3517209.3524042

[25] Tim Kraska, Martin Hentschel, Gustavo Alonso, and Donald Kossmann. 2009. Consistency Rationing in the Cloud: Pay Only When It Matters. *Proc. VLDB Endow.* 2, 1 (aug 2009), 253–264. https://doi.org/10.14778/1687627.1687657

[26] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978), 558–565. https://doi.org/10.1145/359545.359563

[27] Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. *The Byzantine Generals Problem.* Association for Computing Machinery, New York, NY, USA, 203–226. https://doi.org/10.1145/3335772.3335936

[28] Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno M. Preguiça, and Rodrigo Rodrigues. 2012. Making Geo-Replicated Systems Fast as Possible, Consistent when Necessary. In *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, Chandu Thekkath and Amin Vahdat (Eds.). USENIX Association, 265–278. https://www.usenix.org/conference/osdi12/technical-sessions/presentation/li

[29] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2013. Stronger Semantics for Low-Latency Geo-Replicated Storage. In *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX Association, Lombard, IL, 313–328. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/lloyd

[30] Pedro Lopes, João Sousa, Valter Balegas, Carla Ferreira, Sérgio Duarte, Annette Bieniusa, Rodrigo Rodrigues, and Nuno M. Preguiça. 2019. Antidote SQL: Relaxed When Possible, Strict When Necessary. *CoRR* abs/1902.03576 (2019). arXiv:1902.03576 http://arxiv.org/abs/1902.03576

[31] Yunhao Mao, Zongxin Liu, and Hans-Arno Jacobsen. 2022. Reversible Conflict-Free Replicated Data Types. In *Proceedings of the 23rd Conference on 23rd ACM/IFIP International Middleware Conference* (Quebec, QC, Canada) *(Middleware '22)*. Association for Computing Machinery, New York, NY, USA, 295–307. https://doi.org/10.1145/3528535.3565252

[32] MergeSharp. 2023. MergeSharp. https://github.com/yunhaom94/MergeSharp.

[33] Pezhman Nasirifard, Ruben Mayer, and Hans-Arno Jacobsen. 2023. OrderlessChain: A CRDT-based BFT Coordination-free Blockchain Without Global Order of Transactions. In *Proceedings of the 24th International Middleware Conference, Middleware 2023, Bologna, Italy, December 11-15, 2023*. ACM, 137–150. https://doi.org/10.1145/3590140.3629111

[34] Pezhman Nasirifard, Ruben Mayer, and Hans-Arno Jacobsen. 2019. FabricCRDT: A Conflict-Free Replicated Datatypes Approach to Permissioned Blockchains. In *Proceedings of the 20th International Middleware Conference* (Davis, CA, USA) *(Middleware '19)*. Association for Computing Machinery, New York, NY, USA, 110–122. https://doi.org/10.1145/3361525.3361540

[35] Nuno M. Preguiça, Carlos Baquero, and Marc Shapiro. 2018. Conflict-free Replicated Data Types (CRDTs). (2018). arXiv:1805.06358 http://arxiv.org/abs/1805.06358

[36] Nuno Preguiça, Marek Zawirski, Annette Bieniusa, Sérgio Duarte, Valter Balegas, Carlos Baquero, and Marc Shapiro. 2014. SwiftCloud: Fault-Tolerant Geo-Replication Integrated all the Way to the Client Machine. In *2014 IEEE 33rd International Symposium on Reliable Distributed Systems Workshops*. 30–33. https://doi.org/10.1109/SRDSW.2014.33

[37] Yasushi Saito and Marc Shapiro. 2005. Optimistic Replication. *ACM Comput. Surv.* 37, 1 (mar 2005), 42–81. https://doi.org/10.1145/1057977.1057980

[38] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A comprehensive study of Convergent and Commutative Replicated Data Types.* Technical Report. Inria Centre Paris-Rocquencourt.

[39] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems*. 386–400.

[40] Alexander Spiegelman, Neil Giridharan, Alberto Sonnino, and Lefteris Kokoris-Kogias. 2022. Bullshark: DAG BFT Protocols Made Practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Los Angeles CA USA, 2705–2718. https://doi.org/10.1145/3548606.3559361

[41] Sui. 2023. Sui. https://github.com/MystenLabs/sui/.

[42] Florian Suri-Payer, Matthew Burke, Zheng Wang, Yunhao Zhang, Lorenzo Alvisi, and Natacha Crooks. 2021. Basil: Breaking up BFT with ACID (Transactions). In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 1–17. https://doi.org/10.1145/3477132.3483552

[43] Guosai Wang, Lifei Zhang, and Wei Xu. 2017. What Can We Learn from Four Years of Data Center Hardware Failures?. In *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 25–36. https://doi.org/10.1109/DSN.2017.26

[44] Michael Whittaker, Aleksey Charapko, Joseph M. Hellerstein, Heidi Howard, and Ion Stoica. 2021. Read-Write Quorum Systems Made Practical. In *Proceedings of the 8th Workshop on Principles and Practice of Consistency for Distributed Data* (Online, United Kingdom) *(PaPoC '21)*. Association for Computing Machinery, New York, NY, USA, Article 7, 8 pages. https://doi.org/10.1145/3447865.3457962

[45] Maofan Yin, Dahlia Malkhi, Michael K. Reiter, Guy Golan Gueta, and Ittai Abraham. 2019. HotStuff: BFT Consensus with Linearity and Responsiveness. In *Proceedings of the 2019 ACM Symposium on Principles of Distributed Computing* (Toronto ON, Canada) *(PODC '19)*. Association for Computing Machinery, New York, NY, USA, 347–356. https://doi.org/10.1145/3293611.3331591

[46] Gengrui Zhang, Yunhao Mao, Shiquan Zhang, Shashank Motepalli, Fei Pan, and Hans-Arno Jacobsen. 2023. V-Guard: An Efficient Permissioned Blockchain for Achieving Consensus under Dynamic Memberships in V2X Networks. https://doi.org/10.48550/ARXIV.2301.06210

[47] Gengrui Zhang, Fei Pan, Yunhao Mao, Sofia Tijanic, Michael Dang'ana, Shashank Motepalli, Shiquan Zhang, and Hans-Arno Jacobsen. 2023. Reaching Consensus in the Byzantine Empire: A Comprehensive Review of BFT Consensus Algorithms. *ACM Comput. Surv.* (dec 2023). https://doi.org/10.1145/3636553 Just Accepted.

[48] Gengrui Zhang, Fei Pan, Sofia Tijanic, and Hans-Arno Jacobsen. 2024. PrestigeBFT: Revolutionizing View Changes in BFT Consensus Algorithms with Reputation Mechanisms. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 1930–1943. https://doi.org/10.1109/ICDE60146.2024.00156

[49] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2018. Building Consistent Transactions with Inconsistent Replication. *ACM Trans. Comput. Syst.* 35, 4, Article 12 (dec 2018), 37 pages. https://doi.org/10.1145/3269981

[50] Yunhao Zhang, Srinath Setty, Qi Chen, Lidong Zhou, and Lorenzo Alvisi. 2020. Byzantine Ordered Consensus without Byzantine Oligarchy. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation (OSDI'20)*. USENIX Association, USA, Article 36, 17 pages.

[51] Wenbing Zhao. 2016. Optimistic Byzantine fault tolerance. *International Journal of Parallel, Emergent and Distributed Systems* 31, 3 (2016), 254–267. https://doi.org/10.1080/17445760.2015.1078802 arXiv:https://doi.org/10.1080/17445760.2015.1078802

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

[53] 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 *2016 IEEE International Conference on Electro Information Technology (EIT)*. 0246–0251. https://doi.org/10.1109/EIT.2016.7535248