Making CRDTs Not So Eventual

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

Gengrui Zhang
Concordia University
gengrui.zhang@concordia.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 highly available and performant data replication solutions for distributed applications. However, their eventual consistency guarantees are often insufficient for ensuring application correctness, especially in the presence of Byzantine failures. Naively applying traditional consensus and Byzantine fault tolerance (BFT) protocols to CRDT updates for stronger guarantees, while intuitive, negates the performance benefits of CRDTs.

We introduce a novel programming model called *reliable CRDTs* that expands CRDTs with additional guarantees: users can query strongly or eventually consistent values, enforce a total order among selected operations, and define data-type level invariants while remaining operational in the presence of Byzantine failures. Reliable CRDTs enable the use of CRDTs in scenarios where strong consistency is needed while maintaining their performance advantages.

We present an implementation of reliable CRDTs named *Janus*. It enhances CRDTs with the aforementioned features by functioning as a middleware that facilitates CRDT communication and asynchronously runs a BFT consensus protocol. Our evaluation demonstrates that *Janus* achieves 21× higher throughput than naively applying state-of-the-art BFT protocols such as HotStuff achieves, 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, 18(2): 349 - 362, 2024.
doi:10.14778/3705829.3705850

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

1 INTRODUCTION

Conflict-free replicated data types (CRDTs) are *abstract data types* (ADTs) that are designed to replicate across multiple nodes with eventual consistency guarantees [40]. Users simply execute ADT interface operations on one CRDT 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 offer developers an easy-to-use replication solution through well-defined semantics and eliminate the need for manual conflict resolution, introducing an efficient programming model for replication in distributed applications.

Because they are eventually consistent, CRDTs are fast and highly available by allowing temporary divergent states to exist. Replicas can update their local states immediately upon receiving updates and propagate the changes asynchronously [8]. However, this also means that CRDTs may not meet the correctness requirements of many distributed applications because of the existence of intermittent divergent states [15, 38]. We identify three scenarios where CRDT guarantees are insufficient.

First, applications cannot determine whether CRDT replicas have reached a *stable* state. This is because one replica cannot detect whether an update has been delivered to all replicas or to verify the freshness of a value, as eventual consistency does not guarantee a bounded delivery time [8]. If an application needs to perform operations that depend on prior operations or a specific state, it must perform additional consistency checks to ensure that the previous 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 [13].

Second, it is difficult to implement correctness checks for CRDT values, such as maintaining invariants, because of the concurrent operations [11, 30]. 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* [29]. 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.

These problems are typically addressed via strongly consistent storage, which enforces a linear order of operations across all replicas following the ACID transaction model [19, 25]. However, these solutions require costly and time-consuming coordination among replicas, such as consensus, which significantly impacts the system's performance [14]. Thus, in this paper, we propose a novel programming model called *reliable CRDT* that enhances CRDTs with *on-demand*

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 18, No. 2 ISSN 2150-8097.
doi:10.14778/3705829.3705850

strong consistency, *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 expands the scope of existing CRDT-based applications without the need to employ ACID transactions.

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 enables users to designate certain updates as *safe* and ensures that they are executed in a total order across all correct replicas. Applications can declare *data-type level invariants* as data value constraints. Finally, reliable CRDTs tolerate Byzantine failures by preventing *conflicting updates* from violating the guarantees mentioned above.

Reliable CRDTs offer a flexible toolkit for developers to utilize CRDTs in distributed applications. Stronger guarantees can be employed to meet application-level correctness requirements when necessary without customized consistency protocols. For example, a distributed database can use reliable CRDTs as the replicated storage layer to support CRDT objects and provide atomicity properties as needed.

We implement reliable CRDTs via a middleware system named *Janus* that handles the communication among CRDT replicas. It uses an *underlying consensus protocol* operating 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 after the consensus process, they are *reversed* through the mechanism of *reversible CRDTs* [33] during the *audit* process. We incorporate a novel *directed acyclic graph (DAG)-based* BFT consensus protocol [47] because it offers high throughput, asynchronously ordering of updates, and a method of recording the causal update history, which is 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. The performance results are compared with those of plain CRDTs and implementations that apply BFT consensus protocols directly to CRDT updates.

The remainder of 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 presents 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 design.

2.1 Conflict-Free Replicated Data Types

First proposed by Shapiro et al. [36, 40], CRDTs replicate through deterministic interface operations and predefined *concurrency semantics*, thus avoiding the need for application-specific reconciliation mechanisms. A CRDT represents a state (the *data structure* to be replicated), and the interface provides 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 replica immediately, and then it is propagated asynchronously to other replicas. CRDTs adhere to *strong eventual consistency* (SEC), which ensures that updates are eventually delivered to all correct replicas, and the replicas that have received the same updates have equivalent states. This implies that the convergence of two replicas can be determined by comparing the updates that have been delivered to them.

There are two main methods for propagating updates that enable CRDTs to achieve SEC: *state-based* (CvRDT) and *operation-based* (CmRDT) data types. CvRDTs synchronize by sending the entire state, or the delta of the state change [5], of a replica and then *merge* the received states with a *commutative* merge function. This function ensures the same end state regardless of the order in which operations are executed. For example, a CRDT counter 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 each update and execute the same updates on other replicas with the same causal order [28], which requires the operations 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 type level through the semantics of CRDTs, enabling fire-and-forget updates, which contrasts with traditional eventually consistent or transactional systems. In these systems, updating data requires a read-modify-write transaction on the data store, necessitating consistency protocols to either reconcile conflicting states or to ensure that all replicas perform this transaction in the same order [8].

This distinction leads to a unique programming model when CRDTs are used: 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 [33]. 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

Janus utilizes an underlying BFT consensus protocol to achieve the aforementioned guarantees. We choose a DAG-based BFT consensus protocol because it enables replicas to concurrently propose and broadcast messages by decoupling message propagation and ordering logic [47], which allows 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 the 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 determined independently by individual replicas traversing the DAG.

DAG-based BFT protocols include Hashgraph [10], Aleph [21], and, more recently, DAG-Rider [24], Narwhal & Tusk [20], and Bullshark [41]. In recent 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 blocks are organized into *rounds*. At each round, a replica generates a block and reliably broadcasts the block to other replicas asynchronously [17]. Once a replica receives $2f + 1$ blocks from other replicas in a round, it advances to a new round and constructs a new block that contains the latest messages and references to the blocks in the previous round. Referencing blocks would *support* 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) [18]. 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. The consensus process uses three rounds for each wave, with the third round of a wave and the first round of the subsequent wave being overlapped to reduce 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. All colored blocks are committed at the wave when $L3$ is selected as the leader. 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. This process is conducted based on the local copy of the DAG of each replica; thus, no additional communication overhead is introduced.

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 the 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 causal log, which is useful for reversing invalid updates.

3 RELIABLE CRDT OVERVIEW

Strongly consistent 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

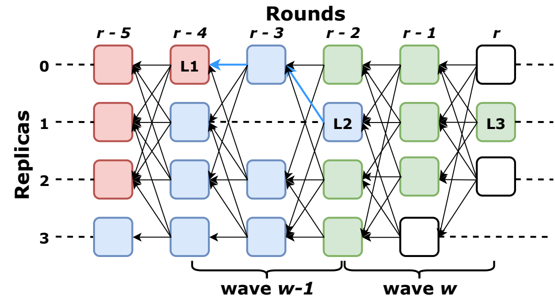


Figure 1: Example of a Narwhal DAG & Tusk consensus instance from the perspective of one replica at round r

strong consistency can meet application-level correctness requirements. Our approach offers stronger guarantees and requires no alteration of the programming model of CRDT data operations, as discussed in Section 2, while maintaining their performance benefits.

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 uphold 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 *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 conflicting updates caused by Byzantine failures.

Before presenting examples to illustrate how our solution enables the use of CRDTs in various scenarios, we categorize three different ways in which applications may perform CRDT data operations: 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. In addition, replicas in the system may be subject to Byzantine failures, which can occur even for internally hosted applications [43]. We consider a replicated banking database whereby clients can check balances, deposit, withdraw, and transfer 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 divergence, such as one replica believing that an account is overdrawn while another one does not. By using reliable CRDTs, the correctness of these operations can be guaranteed without the use of a transactional model.

Assuming that a client wishes to simply check the latest account balance immediately after depositing money, the "check balance" operation is of 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 returning 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 [7, 35]. Since plain CRDT operations are designed to be I-confluent, these operations can be executed preemptively.

Withdrawal, however, is a potentially invariant-breaking operation and can be categorized as type-③. Therefore, the stable account balance must be checked 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 is an *application-level transaction* that requires two dependent operations: withdrawal from one account and depositing into another account, so it is also type-③. Safe updates must be used to ensure that the withdrawal is successful before the deposit is made because they are dependent updates that must be executed in a specific order. The application determines the behavior of subsequent dependent updates based on the status of the first operation. This transaction can also be optimized to enable the deposit to be conducted concurrently without waiting for the withdrawal to succeed if there is a predefined method for undoing the deposit if the withdrawal fails.

Voting System. This example illustrates the direct impact of Byzantine failure on the correctness of the system. We 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 vote for only one candidate. There is no invariant in this case, but a Byzantine replica can double-cast votes to two correct replicas, resulting in the vote count being inconsistent and invalid for the application.

Here, casting a vote is a type-② operation 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.

4 RELIABLE CRDT DESIGN

4.1 System Model and Notation

We consider a distributed system that consists of fully connected nodes and communicates by passing messages. The network is asynchronous but reliable, which means that messages are not 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 is at least $3f+1$ for the BFT protocol to function. The system is *permissioned* [6], 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 via modern cryptographic signature algorithms, such as ECDSA [16]. All of the messages are signed with private keys.

A node stores one replica, denoted as $\mathcal{R}$, for each CRDT instance in the system. The state of one CRDT instance is denoted as c, and a replica $\mathcal{R}=[c_1,c_2,...,c_n]$ is a linear sequence of states starting from an initial state to a state after n updates are executed. Each state is a set of updates (denoted as u) that the replica has received at the given time, and the last state is $c_n=\{u_1,u_2,...,u_n\}$. 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[1]; that is, $c_x\subseteq c_y$ if $x<y\leq n$. Thus, for two updates u_i and u_j, $1\leq i,j\leq n$, u_i *happened-before* u_j if u_i is included in the state of *all* replicas when u_j is received, denoted as $u_i\prec u_j$; and *happened-after* in the opposite case. Updates u_i and u_j are *concurrent* if neither *happened-before* nor *happened-after* the other. These definitions constitute the causal relationships of the CRDT updates [28, 40].

The value of a CRDT is denoted as $Q(c)$, where Q is a query function defined by the CRDT. If two states from two replicas c_i and c_j, $1\leq i,j\leq n$ have the same queried value, namely, $Q(c_i)=Q(c_j)$, 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 $\mathcal{I}$ is defined as a set of m constraints $\mathcal{I}=\{inv_1,inv_2,...,inv_m\}$, where each constraint can be viewed as a Boolean function $inv_i(c)$, $1\leq i\leq m$ that returns true if state c satisfies the constraint.

4.2 Definition of Reliable Operations

In this section, we formally define the operations and guarantees of reliable CRDTs, including the behaviors of prospective and stable values, the guarantees of safe updates, and the concept of conflicting updates.

[1] Although a monotonically increasing state is required only for state-based CRDTs, all kinds of CRDTs are mathematically equivalent, and they can be constructed to have the same structure [40].

Q_{stable} and $Q_{prospective}$ are functions that query stable and prospective values, respectively. A 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, it may be inconsistent, stale, or invariant-breaking.

The stable value can be viewed as the history of *stable states* that are consistent across all correct replicas, which is called the *stable history* denoted as $\mathcal{L}$. 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 corresponding 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 $\mathcal{L} = [5,4]$, then for both R_0 and R_1, a stable query yields $[5,4]$, whereas prospective queries may return intermediate values such as 2 for $\mathcal{R}_0$ and 7 on $\mathcal{R}_1$.

Definition 1 (Stable History and Stable State). *The stable history $\mathcal{L}$ is a subsequence of a CRDT replica $\mathcal{R}$ that always satisfies the constraints $\mathcal{I}: \forall c_i \in \mathcal{L} = [c_1,c_2,...,c_n], \forall inv_j \in \mathcal{I}, inv_j(c_i) = true$.*

States in L are called the stable states. Given a stable state c_k in $\mathcal{L}$ on one correct replica at index k where $k \leq n$, and for the stable state c'_k on all other correct replicas $Q(c_k) = Q(c'_k)$.

Definition 2 (Stable Query). $Q_{stable}(\mathcal{R}) = \mathcal{L}$

To ensure the freshness of the stable value returned from a replica, a user can perform a *quorum read* [44] to gather the stable histories from at least $f+1$ replicas. Since the stable values are consistent among the correct replicas, the union of the stable histories can be used to determine the latest stable value[2].

The update operations behave identically to those of the underlying CRDTs. If an update is designated *safe*, denoted as u^t, it will be returned to the caller as successful only after it has been successfully included in the stable state; if it is successful, the stable state must also be updated. In contrast, plain CRDT updates are considered successful if 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 serializable.

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 successfully on all replicas.

Definition 3 (Safe Update). *Safe updates are specially designated updates that are guaranteed to be executed in a total order across all correct replicas. Let $u^t \in \mathcal{L}$ be a safe update of one correct replica, then for $\mathcal{L}'$ of all correct replicas, $\exists c \in \mathcal{L}'$ such that $u^t \in c$.*

The Byzantine safety property prevents *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 and are 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 disallow message reordering.

[2] In practice, it is sufficient to gather only the latest stable values from $f+1$ nodes.

Definition 4 (Conflicting Updates). *We consider three distinct updates u_1, u_2, and u_3 from a correct replica $\mathcal{R}_f$ that are delivered to $\mathcal{R}_i$ in the order of $u_1 \prec u_2 \prec u_3$. If another correct replica $\mathcal{R}_j$ receives $u_1 \prec u'_2 \prec u_3$, where $u_1(\mathcal{R}_i) = u_1(\mathcal{R}_j)$ and $u_3(\mathcal{R}_i) = u_3(\mathcal{R}_j)$, but $u'_2 \neq u_2$ (in terms of resulting in different states 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 $c = [u_1, u_2, \ldots, u_n]$ such that $Q_{prospective}(c) = latest(Q_{stable}(\mathcal{R}))$ where $latest()$ returns the last item in the stable history.*

5 JANUS: RELIABLE CRDT MIDDLEWARE

5.1 Overview

Janus consists of three main components: CRDT instances, an underlying consensus protocol, and an update auditor. Each replica of a reliable CRDT instance has two state variables for stable and prospective values, namely prospective and stable states, which are 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 history for the auditing process. Although other consensus protocols (or non-BFT consensus protocols if the network assumption exhibits only crash failures) can be used here, a separate channel for prospective update propagation and a method of tracking the update history are needed, which is less efficient.

An overview of *Janus*'s workflow is shown in Figure 2. For simplicity, we assume that there are no faulty clients, and replicas communicate in a peer-to-peer fashion. (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 by placing CRDT updates in DAG blocks. (4) When a DAG block is received from another replica, (5) the replica executes the CRDT 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 prospective state is *reversed* if there are rejected invalid updates. Finally, the clients who issue the *safe* updates are notified upon completion.

Janus strives for a balance between the performance of the CRDTs and the applicability of strong consistency. In contrast to traditional consensus protocols, it allows the preemptive execution of updates

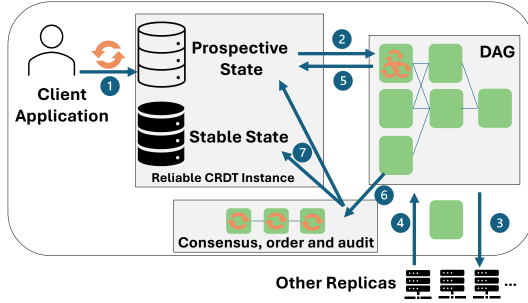


Figure 2: Workflow of *Janus*

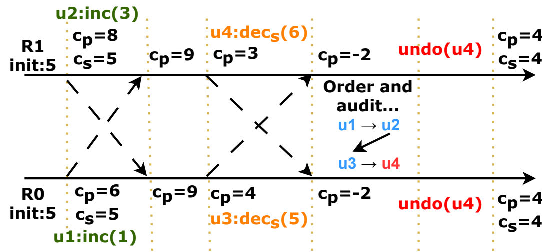


Figure 3: A reliable CRDT counter under *Janus*

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. In contrast to plain CRDTs, it offers the ability to enforce strongly consistent guarantees while incurring only a relatively small computational overhead.

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*[3], which retroactively undoes invalid updates after consensus is conducted by going through the DAG and checking if any received update is not committed. This process is performed independently on each replica at the time of a consensus commit.

We utilize the concept of *compensation* in *reversible CRDTs* [33] to reverse invalid updates. As mentioned in Section 2.1, if the effect of one update cancels out that of another update, the current state can be considered equivalent to the previous state. Note that the effect of compensation operations depends on the specific type, and must be predefined based on the use case. For example, the compensation for the *add* operation in a CRDT counter 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. In this figure, the *inc*() operation increases the counter by an integer, and the $dec_s()$ decreases the counter by an integer and is designated *safe*; variable c_p denotes the values of the prospective state, and c_s denotes the values of the stable state at a specific time.

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

Starting with an initial value of 5 for two replicas[4], $\mathcal{R}_0$ 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, both prospective values are 9 (the stable state has not changed yet). Next, both replicas conduct safe decrement operations: $u3$ and $u4$. Although $u3$ and $u4$ do not break the invariant during their local executions, the combined effect does, as the prospective states temporarily become negative; therefore, one of the updates is not valid. After the updates are ordered by consensus, $u4$ is rejected, yielding a final stable value of 4. The prospective state consequently reverses the executed $u4$.

5.3 *Janus* Algorithm

Algorithm 1 *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, is_safe: Bool)
5: $\quad$ $execution_result \leftarrow p_state.Execute(u)$ ▷ *Local execution*
6: $\quad$ **if** $execution_result$ **then**
7: $\quad\quad$ $consensus_result : awaitable \leftarrow dag.Propose(u)$ ▷ *Asynchronously send the update to the DAG instance*
8: $\quad$ **if** is_safe and $execution_result$ **then**
9: $\quad\quad$ **return** await $consensus_result$ ▷ *Caller can wait on the consensus result*
10: $\quad$ **else**
11: $\quad\quad$ **return** $execution_result$

12: **function** OnAcceptingBlock ▷ *On new DAG block*
13: $\quad$ $b : Block \leftarrow dag.GetLatestBlock()$
14: $\quad$ **for all** $u \in b.updates$ **do** ▷ *Execute all updates in the block*
15: $\quad\quad$ $p_state.Execute(u)$

16: **function** OnCommit ▷ *When the consensus commits updates*
17: $\quad$ $to_reverse \leftarrow Audit()$
18: $\quad$ $p_state.Reverse(to_reverse)$;
19: $\quad$ $stable_history.Append(s_state)$
20: $\quad$ Send $consensus_result$ to all waiting safe updates

The core design of *Janus* is presented in Algorithm 1, which follows the workflow in Figure 2. The variables s_state and p_state represent replicas for one or more CRDT instances. We only consider a single CRDT 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 present the process of block generation here since it is part of the DAG algorithm.

[4]We assume that the total number of replicas is sufficient for meeting the $3f + 1$ requirement.

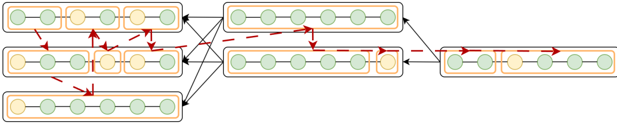


Figure 4: Ordering CRDT updates in committed DAG blocks

When the consensus process decides to commit in Line 16 in the *OnCommit*() function, the replica calls *Audit*() (Line 1) in Algorithm 2 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.

The audit process can be viewed 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 dependencies are preserved (the dependent updates must be 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 updates in a breadth-first manner from the earliest round in the wave and move in a round-robin fashion, as shown in Figure 4 where the safe updates are marked in red, and the orange boxes represent update groups. The order of execution is shown with red arrows. 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, which are identified by checking the updates 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.

With state-based CRDTs, we implement an optimization method called *state compaction* that combines updates of the same CRDT instance in a batch into one state as the update message for propagation to reduce the message size. This can be done because later states implicitly encapsulate all previous states when the state is disseminated [40].

5.4 Correctness

In this section, we show that *Janus* upholds 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 [40], 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 in each

Algorithm 2 Order and Audit

```
 1: function Audit
 2:     to_reverse: List
 3:     while u: Order().Dequeue() do
 4:         result ← s_state.TryExecute(u,InvariantCheck()); ▷ Execute on the stable state; if this breaks the invariant, return false and leave the state unchanged
 5:         if !result then
 6:             to_reverse.Add(u)
 7:     to_reverse.Add(∀u ∈ dag.uncommitted_blocks) ▷ Uncommitted blocks due to faulty replicas or retransmission
 8:     return to_reverse

 9: function Order
10:     execution_order: Queue
11:     for i ∈ 0..number_of_rounds_to_commit do ▷ Traverse from the earliest updates while following the causal order
12:         loop
13:             j ← 0
14:             b: Block ← dag.blocks_to_commit[i][j]
15:             while ¬b.updates.Empty() do
16:                 execution_order.Enqueue(b.updates.Pop())
17:                 if b.updates.is_safe then
18:                     j++ if dag.blocks_to_commit[j+1] ≠ null else j = 0
19:     return execution_order;
```

wave. Each block must contain the same set of updates (messages) because they cannot be forged since all the messages are signed; thus, the same set of updates must be performed on all the correct replicas, as this property follows directly from the DAG consensus protocol [20].

Theorem 1. *Querying stable_history in Janus satisfies the properties of the stable history specified 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 1.

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 in Line 4 of Algorithm 2, *s_state* must satisfy all the invariants. Therefore, *stable_history* satisfies the stable history properties in Definition 1. ◻

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

Proof. In Line 9 of Algorithm 1, if the *is_safe* 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 the updates deterministically, the set of safe updates must be totally ordered. Therefore, the safe updates satisfy the properties of safe updates in Definition 3. ◻

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

Proof. We show this by proving that no conflicting updates from Definition 4 is included in the *stable_history*.

We 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 stable states for all correct replicas must be equivalent. However, according to SEC, if the conflicting updates have different contents, 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 not executed on *s_state*, then it must be added to the *to_reverse* set in Line 6 of Algorithm 2.

If the block is not committed, it is never executed on *s_state* and added to the *to_reverse* set in 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. The *p_state* executes all received updates that either come from the user or are received by a DAG 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 undo all effects from updates in *to_reverse* [33]. 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 determined primarily by the underlying BFT protocol (which affects the messaging complexity) and the CRDT operations (which affect the message size) since they are responsible for communicating and updating the data objects. For Narwhal & Tusk, the messaging complexity given m nodes and n updates is $O(|n|m^3 \log m)$ [20, 47].

Our algorithms impact the processes of applying received updates, ordering, and auditing updates. Since there are two copies of a CRDT instance, namely, prospective and stable states, the time complexity of applying n updates in the worst-case (assuming that all updates are applied to both states) is $O(2n) = O(n)$, see Line 14 of Algorithm 1 and Line 3 of Algorithm 2.

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

6 EVALUATION

In this section, we evaluate a proof-of-concept implementation of *Janus*. It functions as a distributed key–value database server in which each key–value pair is a CRDT object using an open-source CRDT library in C# [2], and the *Janus* middleware that contains a C# port of the core DAG algorithm described in Narwhal & Tusk [20].

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 the performance characteristics of our algorithm. Furthermore, Narwhal was developed as part of the Sui blockchain written in Rust that only supports blockchain transactions[3], 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, which is hosted on the Compute Canada cloud. Each VM has two 2.4 GHz Intel Xeon vCPUs (Broadwell) and 15 GB of RAM, and are interconnected by a high-speed LAN with less than 1*ms* latency and 3Gbps bandwidth. Each VM hosts one server node, and the benchmark clients run on a separate VM but in the same cloud region.

We evaluate *Janus* using microbenchmarks with synthetic workloads and a demo implementation of the banking database application according to the example in Section 5 for a realistic evaluation.

6.2 Microbenchmarks

We use synthetic workloads because common database benchmarks, such as TPC-C and YCSB do not directly support CRDTs. Our 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 [39]. 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 against the incoming vectors. 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. Only elements with tags in the added set and not in the removed set are considered visible. Propagating updates requires sending both sets, and merging involves taking the union of the received sets. The average message size is $2.4KB$ in our experiments[5].

We compare our results against 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 the message size set to 380 bytes and 2.4 KB to mimic conducting consensus on all CRDT messages via HotStuff, and our implementation of Narwhal & Tusk using our

[5]Due to memory constraints of the VMs, each OR-Set instance is deleted after reaching 50 elements for most of the experiments, and then a new instance is created.

KVDB by passing all updates to the consensus protocol before they are executed.

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 a response to the previous request. The two main metrics are the 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 the end-to-end latency in milliseconds (ms) measured on the clients.

We adjust the following workload parameters in different experiments: The *safe update ratio* is the percentage of safe updates out of all update operations. The *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. The *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. The *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 indicated by the throughput) for different batch sizes b; greater loads are represented by increasing the sending rate of the clients. The experimental runs are configured with a balanced (50% update/50% read) access pattern, with 50% safe updates, 100 objects, and 4 nodes.

Each sample point is the mean of 5 runs with a fixed sending rate, and the error bars represent one standard deviation in average latency. HotStuff is evaluated by setting the message sizes to the average sizes of the PN-Counter and OR-Set messages.

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× the peak of Narwhal at 170,000 ops/s and 21× the peak of HotStuff at only 12,000 ops/s. This demonstrates the effectiveness of *Janus* over traditional protocols and non-CRDT solutions. However, the peak throughput of *Janus* is only 0.65× the peak throughput of plain CRDTs, showing the performance trade-off of the reliable CRDT features. The latencies of HotStuff and plain CRDTs are much lower when the load is light, at approximately 30ms and 3ms, respectively, but the advantage quickly diminishes as the load increases.

Increasing the batch size for *Janus* from 1 to 500 increases the peak throughput by 2.3×. 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 by 2.3× because of the time spent filling the batch.

For OR-Set, both *Janus* and Narwhal have a reduced 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× increase in message size from PN-Counter to OR-Set results in a 70% decrease in throughput. However, *Janus* is still 1.6× faster than Narwhal and 11× faster than HotStuff at only 7,000 ops/s, but has only 0.4× 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. The experimental runs are configured with a balanced access pattern, 50% safe updates, $b = 500$, and 4 nodes. For PN-Counter, there is no significant performance variation based on the number of objects. 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 count reduces the frequency of resetting OR-Set instances[5] and exhausts the node's memory, leading to frequent memory swapping to the disk and reduced performance.

We do not expect the number of objects to have a significant effect on the performance of *Janus* because it always operates on a single object at a time and there is no difference from the perspective of the system when the number of objects changes.

Message Size. We evaluate the direct impact of the message size using OR-Set by removing the 50-element-cap[5] for each OR-Set instance in this specific experiment, shown in Figure 7. The experimental runs are configured with a balanced access pattern, $b = 500$, 50% safe updates, 100 objects, sending rate set to 1000 ops/s, and 4 nodes. We measure 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 the 50-element cap, the latency remains stable around $100 - 200$ ms, as the states of the CRDTs do not grow indefinitely, nor does the message size.

However, with uncapped OR-Set, we observe that the 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 rises in a stepwise pattern after a certain point, likely due to the longer time required to receive larger messages, resulting in many updates being completed simultaneously, followed by a prolonged wait for the next group of updates.

Latency of Operations. Figure 8 depicts the latency changes of different operations when increasing the sending rate of the clients. We measure the mean, median, and 95th and 99th percentile latencies for safe updates, regular updates, and reads. The experimental runs are configured with a balanced access pattern, 50% safe updates, $b = 500$, 100 objects and 4 nodes.

There is a large discrepancy between safe and regular operations for both CRDTs, as expected, due to the slow 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 and their latencies increase to more than 10 seconds when the system becomes saturated, which is still much greater than even the outliers for regular operations.

This experiment shows that although the consensus process struggles when the system is saturated, the system can still handle some

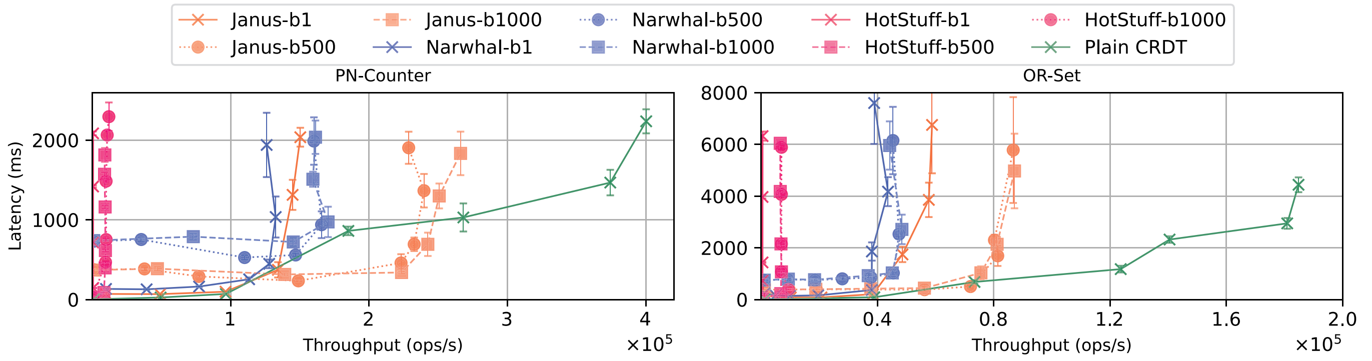


Figure 5: Throughput vs. latency

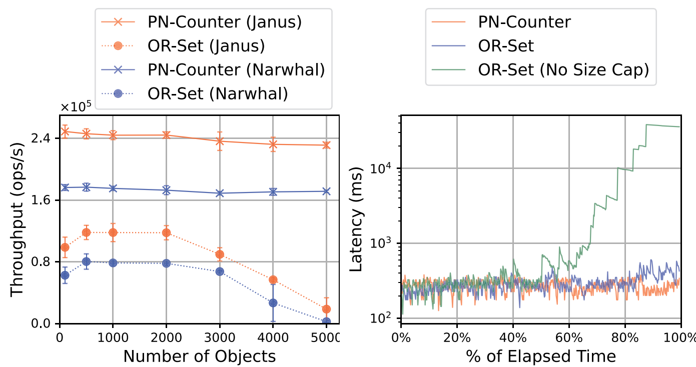


Figure 6: Peak throughput vs. number of objects

Figure 7: Latency of safe updates time series

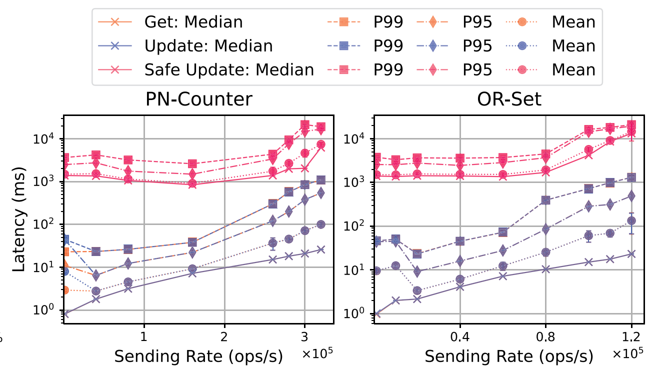


Figure 8: Latency of operations

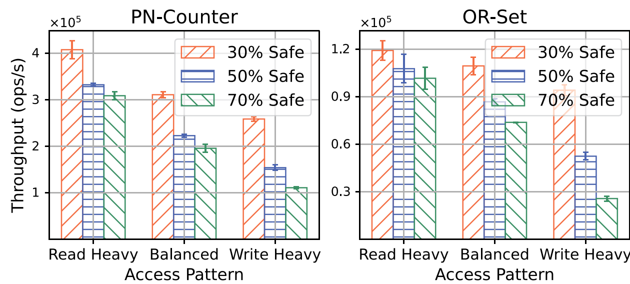


Figure 9: Peak throughput vs. access patterns and safe update ratios

regular operations within a reasonable time frame, which can be further proven by the increasing gap between the 95th and 99th percentiles and the average latency of regular operations. This is because there are more outliers in terms of latency under a heavy load, but half of the operations are still processed promptly, resulting in a low median latency.

Access Pattern and Safe Ratio. We vary the ratios of operation types 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. The experimental runs are configured with $b = 500$, 100 objects, and 4 nodes.

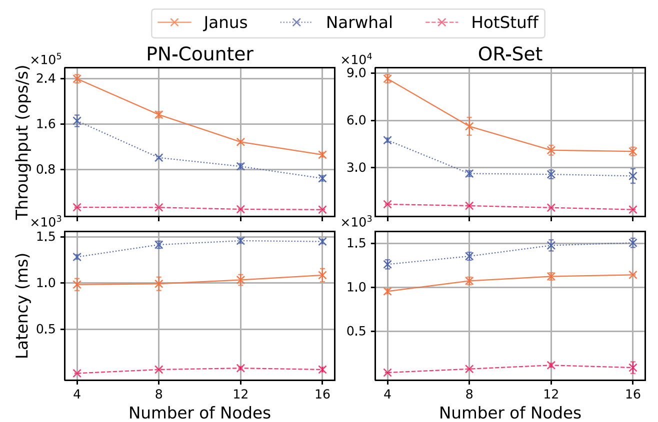


Figure 10: Scalability

As shown in Figure 9, the write-heavy workloads are more sensitive to the ratio of safe updates than are the read-heavy workloads, 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.

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

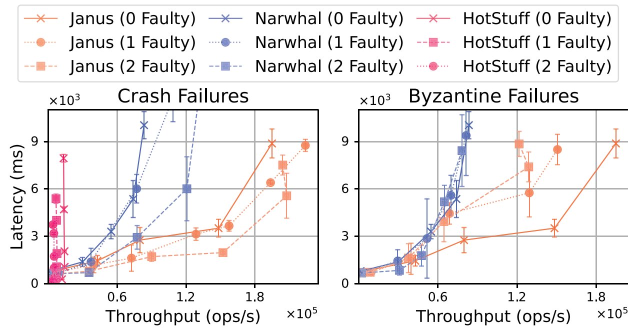
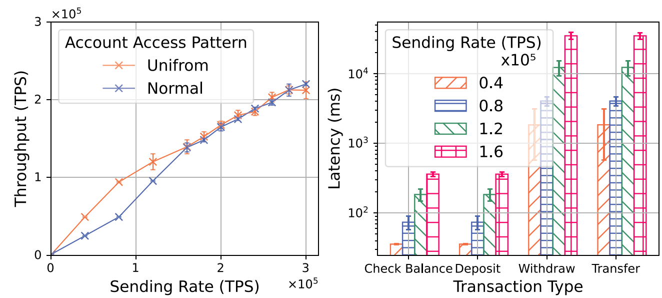


Figure 11: Throughput vs. latency with faulty nodes



(a) Throughput vs. request rate **(b) Transactions latency**

Figure 12: Banking database performance

throughput and the average latency when the systems are lightly loaded. The experimental runs are configured with a balanced access pattern, $b = 500$, 50% safe updates, and 100 objects.

For PN-Counter, the decrease in peak throughput is almost linear with the number of nodes, ranging from approximately 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, the 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 *Janus* is comparable to that of HotStuff.

The cluster size only marginally affects the latency for *Janus*. For PN-Counter, the latency increases by approximately 150ms when 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 are reached. For HotStuff, the latency increases from 20ms to 60ms for PN-Counter sized messages and from 30ms to 85ms for OR-Set sized messages. The advantage of HotStuff in terms of latency 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 demonstrated in Figure 11. The experimental runs are configured with a balanced access pattern, $b = 500$, 50% safe updates, and 100 PN-Counter objects.

For crash failures, we terminate 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 any messages, thus reducing the messaging complexity. For HotStuff, the performance decreases significantly 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 the DAG BFT protocols because there is no leader. However, if the crashed node is not the leader, a similar performance improvement can be expected [48].

To model Byzantine failures, we set 50% of the updates propagated by the faulty replicas to be invalid by not creating certificates for half of the blocks. HotStuff is not considered here because 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 are not included in the throughput measurement.

The effects of Byzantine failures 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 are no invalid updates requiring reversal. However, even with two faulty replicas, *Janus* still outperforms pure Narwhal.

6.3 Banking Database Performance

We implement the banking database example as a distributed application that utilizes *Janus* with 4 types of transactions: depositing, withdrawing, transferring money, and checking balances. Each account is represented by a PN-Counter object with an invariant check for nonnegative 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 on a 4–node cluster with 100 accounts, a balanced workload with 50% balance checks, and 50% update transactions where half of the update transactions are withdrawals and the other half 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 latencies increase by approximately 10× when the sending rate increases from

40,000 to 160,000 TPS, whereas the latency for transactions involving safe updates increases by approximately 18×.

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

6.4 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.

Message size and complexity within underlying CRDTs are the main factors that affect performance, as demonstrated by the performance difference between OR-Set and PN-Counter. Therefore, a future direction for optimization could be to further reduce the message size by using operation-based and delta-based CRDTs [5] 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 [8, 9, 31, 37]. There have also been efforts to combine different consistency levels into one system [4]. For example, RedBlue consistency [30] 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 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 [27], 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 [1, 32] 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.

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 [35] replaces ordering in blockchains with CRDTs that also allow for invariant checks. The Vegvisir blockchain [23] uses a DAG chain instead of a linear chain, and branching chains are merged via CRDTs. Both of these methods enable 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 enables the replicas to detect Byzantine failures and eliminate *conflicting messages*. The same authors (2013, 2016) presented similar approaches for enhancing the 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) [26] proposed a novel approach for addressing 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 via cryptographic hashes) to ensure the causal order and integrity of the updates. Barbosa et al. (2021) [12] and Jannes et al. (2022) [22] suggested methods for combatting 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 protocols, several works on BFT consensus endorse concurrency. For example, instead of using a totally ordered BFT log as in traditional consensus, Basil [42] 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*, the 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 that are 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.

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.

ACKNOWLEDGMENTS

This work was in part supported by NSERC and ORF.

REFERENCES

- [1] 2022. *Antidote: A planet scale, highly available, transactional database built on CRDT technology.* Retrieved August, 2024 from https://github.com/AntidoteDB/antidote
- [2] 2023. *MergeSharp.* Retrieved August, 2024 from https://github.com/yunhaom94/MergeSharp
- [3] 2024. *Sui.* Retrieved August, 2024 from https://github.com/MystenLabs/sui/
- [4] 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. (2019). arXiv:1902.03305 [cs.DC] https://arxiv.org/abs/1902.03305
- [5] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. 2015. Efficient state-based crdts by delta-mutation. In *International Conference on Networked Systems (NETYS 2015).* Springer, 62–76.
- [6] 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. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18).* ACM, Article 30.
- [7] 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.
- [8] Peter Bailis and Ali Ghodsi. 2013. Eventual Consistency Today: Limitations, Extensions, and Beyond. *Commun. ACM* 56, 5 (May 2013), 55–63.
- [9] 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* (New York, New York, USA) *(SIGMOD '13).* ACM, 761–772.
- [10] Leemon Baird. 2016. *The Swirlds Hashgraph Consensus Algorithm: Fair, Fast, Byzantine Fault Tolerance.* Technical Report. https://funkthat.com/gitea/jmg/medashare/raw/commit/7a9e714a6102774f5d0d84633c6cf5e8a408217a/papers/SWIRLDS-TR-2016-01.pdf
- [11] Valter Balegas, Diogo Serra, Sérgio Duarte, Carla Ferreira, Marc Shapiro, Rodrigo Rodrigues, and Nuno Preguiça. 2015. Extending Eventually Consistent Cloud Databases for Enforcing Numeric Invariants. In *34th Symposium on Reliable Distributed Systems (SRDS '15).* IEEE, 31–36.
- [12] 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).* ACM, 6–15.
- [13] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1987. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley.
- [14] Eric Brewer. 2012. CAP twelve years later: How the "rules" have changed. *Computer* 45, 2 (Jan. 2012), 23–29.
- [15] Lucas Brutschy, Dimitar Dimitrov, Peter Müller, and Martin Vechev. 2017. Serializability for Eventual Consistency: Criterion, Analysis, and Applications *(POPL '17).* ACM, 458–472.
- [16] Johannes Buchmann. 2004. *Introduction to cryptography.* Springer.
- [17] 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, 524–541.
- [18] 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).* ACM, 123–132.
- [19] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. 2011. *Distributed Systems: Concepts and Design* (5th ed.). Addison-Wesley Publishing Company.
- [20] 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* (Rennes, France) *(EuroSys '22).* ACM, 34–50.
- [21] Adam Gągol, Damian Leśniak, 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).* ACM, 214–228.
- [22] 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).* ACM, 31–36.
- [23] 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, 1150–1158.
- [24] 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* (Virtual Event, Italy) *(PODC'21).* ACM, 165–175.
- [25] Martin Kleppmann. 2016. *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems.* O'Reilly.
- [26] Martin Kleppmann. 2022. Making CRDTs Byzantine Fault Tolerant. In *Proceedings of the 9th Workshop on Principles and Practice of Consistency for Distributed Data* (Rennes, France) *(PaPoC '22).* ACM, 8–15.
- [27] 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.
- [28] Leslie Lamport. 2019. *Time, clocks, and the ordering of events in a distributed system.* ACM, 179–196.
- [29] Leslie Lamport, Robert Shostak, and Marshall Pease. 1982. *The Byzantine Generals Problem.* ACM, 203–226.
- [30] 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* (Hollywood, CA, USA) *(OSDI '12).* USENIX Association, 265–278.
- [31] 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, 313–328.
- [32] Pedro Lopes, João Sousa, Valter Balegas, Carla Ferreira, Ségio Duarte, Annette Bieniusa, Rodrigo Rodrigues, and Nuno Preguiça. 2019. Antidote SQL: Relaxed When Possible, Strict When Necessary. arXiv:1902.03576 [cs.DB] https://arxiv.org/abs/1902.03576
- [33] 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).* ACM, 295–307.
- [34] Pezhman Nasirifard, Ruben Mayer, and Hans-Arno Jacobsen. 2019. FabricCRDT: A Conflict-Free Replicated Datatypes Approach to Permissioned Blockchains *(Middleware '19).* ACM, 110–122.
- [35] 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* (Bologna, Italy) *(Middleware '23).* ACM, 137–150.
- [36] Nuno Preguiça. 2018. Conflict-free Replicated Data Types: An Overview. arXiv:1806.10254 [cs.DC] https://arxiv.org/abs/1806.10254
- [37] 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 *33rd International Symposium on Reliable Distributed Systems Workshops* (Nara, Japan) *(SRDS '14).* IEEE, 30–33.
- [38] Yasushi Saito and Marc Shapiro. 2005. Optimistic replication. *ACM Comput. Surv.* 37, 1 (March 2005), 42–81.
- [39] 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.
- [40] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *13th International Symposium on Stabilization, Safety, and Security of Distributed Systems* (Grenoble, France) *(SSS 2011).* Springer, 386–400.
- [41] 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* (Los Angeles, CA, USA) *(CCS '22).* ACM, 2705–2718.
- [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).* ACM, 1–17.
- [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* (Denver, CO, USA) *(DSN 2017).* IEEE, 25–36.
- [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).* ACM, Article 7.
- [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).* ACM, 347–356.
- [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. arXiv:2301.06210 [cs.DC] https://arxiv.org/abs/2301.06210

[47] Gengrui Zhang, Fei Pan, Yunhao Mao, Sofia Tijanic, Michael Dang'ana, Shashank Motepalli, Shiquan Zhang, and Hans-Arno Jacobsen. 2024. Reaching Consensus in the Byzantine Empire: A Comprehensive Review of BFT Consensus Algorithms. *ACM Comput. Surv.* 56, 5, Article 134 (Jan. 2024).

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

[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).

[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, Article 36.

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

[52] Wenbing Zhao and Mamdouh Babi. 2013. Byzantine fault tolerant collaborative editing. In *IET International Conference on Information and Communications Technologies (IETICT 2013)*. IEEE, 233–240.

[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 2016)*. IEEE, 0246–0251.