Ensuring Convergence and Invariants Without Coordination

Dina Borrego
NOVA School of Science and Technology, Caparica, Portugal
Vrije Universiteit Brussel, Belgium

Nuno Preguiça
NOVA School of Science and Technology, Caparica, Portugal

Elisa Gonzalez Boix
Vrije Universiteit Brussel, Belgium

Carla Ferreira
NOVA School of Science and Technology, Caparica, Portugal

Abstract

The CAP theorem demonstrates a trade-off between consistency and availability (and, by extension, latency) in systems where network partitions are unavoidable, such as in cloud computing and local-first software. While adopting weak consistency can preserve availability, it may result in inconsistencies that compromise application correctness. Replicated data types provide a principled, coordination-free approach to guarantee convergence but do not consider application invariants. Existing methods for maintaining invariants in replicated systems either rely on coordination – undermining the benefits of weak consistency – or suffer from limited applicability. This paper introduces the No-Op framework, a generic approach for enforcing consistency without coordination while guaranteeing both convergence *and* invariant preservation. The core idea of the No-Op approach is to resolve conflicts among concurrent operations by prioritising one operation over the other according to programmer-defined conflict resolution policies. This prioritisation transforms the less-preferred operation into a no-side-effect operation, ensuring conflict-free execution. We formalise the model underlying the No-Op framework and introduce a replication protocol built upon it, accompanied by a formal proof of correctness for both the framework and the protocol. Furthermore, we demonstrate the framework's applicability by showcasing the design of widely used replicated data types and the preservation of a wide range of application invariants.

2012 ACM Subject Classification Computing methodologies → Distributed computing methodologies; Software and its engineering → Correctness; Theory of computation → Distributed algorithms; Computer systems organization → Availability

Keywords and phrases distributed systems, conflict resolution, RDTs, invariant preservation

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2025.4

Supplementary Material *Software (ECOOP 2025 Artifact Evaluation approved artifact)*: `https://doi.org/10.4230/DARTS.11.2.1`

Funding This work is partially supported by PRT/BD/154519/2022 awarded by the EUTOPIA European University Alliance, UID/04516/NOVA LINCS with the financial support of FCT.IP and the European Commission through the TaRDIS project (agreement ID 101093006).

Acknowledgements We would like to thank Kevin De Porre for the helpful discussions about VeriFx and the anonymous reviewers for their valuable feedback, which helped improve the paper.

1 Introduction

Replication is a key technique in data management systems that enables high availability, fault tolerance, low latency, and scalability. In cloud storage systems [28, 19, 5, 16], data is typically geo-replicated across multiple geographic locations to ensure low latency for clients.

Similarly, local-first software [27] allows clients to maintain a local copy of the data, enabling fast execution and continuous operation, even when disconnected. However, achieving these benefits relies on clients accessing a replica without coordinating with others. As a result, replication creates a trade-off between consistency and latency [1].

Weak consistency [45, 2, 34, 10, 43, 27] sacrifices consistency in favour of low latency and offline availability. In these models, operations are first executed locally on a replica (without coordination), with their effects propagating asynchronously to other replicas. This execution model leads to different execution orders across replicas, requiring a mechanism to guarantee that all replicas eventually converge to the same state. Replicated data types (RDTs) [40, 18, 13, 26, 9] offer a principled, coordination-free approach to ensure that, after the execution of the same set of updates, replicas converge to the same state.

However, state convergence alone does not ensure application correctness, as combining the effects of concurrent operations may violate application invariants and lead to an incorrect state. Although synchronising all operations (i.e., employing strong consistency [11, 23]) can preserve invariants, this approach significantly compromises system performance [4].

Much research [15, 30, 7, 21, 31, 25, 18] has focused on reducing coordination by synchronising only operations that might violate invariants, typically identified through static analysis tools. Non-conflicting operations, in contrast, are executed without requiring coordination. An alternative strategy is to move coordination outside the critical execution path, building on escrow and reservation techniques [36, 8, 7, 33, 38, 41]. By reserving the right to execute specific operations in advance, replicas allow operations to be safely executed without synchronisation, provided the necessary rights are locally available. Otherwise, coordination is required to acquire the rights from other replicas, or the operation must abort.

Finally, recent research has focused on maintaining invariants using coordination-free mechanisms [35, 6]. These mechanisms enable asynchronous execution, automatically resolving invariant violations using automatic resolution policies. These approaches build on the principles of RDTs, extending them from ensuring data convergence to preserving invariants: concurrent operations are executed without coordination while guaranteeing that replicas converge to a state where application invariants remain upheld. However, while some solutions, such as Antidote SQL [35], focus solely on referential integrity, others, like IPA [6], address invariant conflicts but rely on RDTs to achieve convergence.

In this paper, we propose a coordination-free consistency framework that addresses both data convergence and invariant preservation in a unified way. Our approach builds on a simple observation: when concurrent operations occur, their effects can be combined to ensure that replicas converge without violating invariants unless the operations conflict with one another. In cases of conflicting operations, it becomes necessary to prioritise the effects of one operation over the other. Our framework conceptually achieves this by transforming the not-preferred operation into an operation with no side effect (or a No-Op).

To validate our claim, we employ two complementary methods. First, we formally define our No-Op approach and provide proof establishing its correctness. Second, we demonstrate its applicability by showing that this simple model can encode popular RDTs and enforce a wide range of invariants, including referential integrity.

We introduce a replication protocol that adopts this framework to maintain consistency in replicated applications. The protocol was implemented in VeriFx [17], a high-level programming language with built-in automated proof capabilities, enabling validation of its correctness through automated proofs on specific data type implementations. These results corroborate the formal proofs provided, confirming the correctness of the No-Op approach.

To summarise, the main contributions of this paper are:

- The No-Op generic framework for enforcing consistency without coordination, achieving both convergence and invariant preservation in a unified way.
- A replication protocol that integrates the No-Op framework.
- Formal proofs of correctness for the proposed framework and replication protocol.
- The applicability of the No-Op framework to the design of popular RDTs and the maintenance of general application invariants.

The remainder of this paper is organised as follows. Section 2 offers an overview of the No-Op approach, followed by Section 3, which introduces its formal model and a formal proof of its correctness. Section 4 details the No-Op replication protocol and provides the corresponding correctness proofs. Section 5 explores the applicability of the proposed solution. Section 6 discusses limitations of the No-Op approach and possible solutions to avoid these limitations. Then, Section 7 discusses related work. Finally, Section 8 concludes the paper.

2 Overview

This section overviews our approach using an album management system as a running example. This application manages a collection of artists and their albums, supporting operations such as adding, removing, and updating artists, as well as adding and removing albums. This example was previously used in [35] and is similar to an example from [16].

The goal of our No-Op approach is to guarantee state convergence and invariant preservation without requiring coordination. Figure 1 and Figure 2 illustrate a convergence and an invariant conflict that could occur when implementing the album management system. Consider that the application's state is modelled as a set of pairs (artist, country) representing artists and their respective countries, along with a set of elements representing albums. Figure 1 shows a convergence conflict. Initially, both replicas are in an equivalent state, each containing one artist (Sam, UK). Replica 1 updates the artist, resulting in a state with one artist (Sam, USA). Replica 2 removes the artist, leaving its state empty. After propagating both updates, the replicas have applied all operations yet reached divergent states. Figure 2 shows how a referential integrity invariant can be violated. The application must ensure that albums are not associated with non-existent artists. Starting again from an initial state where both replicas contain one artist (Sam, UK), replica 1 adds an album, A1, for Sam, updating its state to include the album. Meanwhile, replica 2 removes the artist, leaving its state empty. After propagating both updates, both replicas converge to a state containing the album but not the artist. Therefore, even though the replicas converge after applying all operations, the referential integrity invariant is violated, as the replicas' state contains an album whose associated artist does not exist.

The core idea of the No-Op approach is to avoid by design *conflicting operations*, i.e., operations that violate state convergence or application invariants. When two conflicting operations are executed concurrently, one is converted into a no-side-effect operation (i.e., a No-Op) based on a predefined conflict resolution policy. These policies vary based on the application's intended semantics, and it is the programmer's responsibility to select the most suitable one for the application. The No-Op approach implements conflict resolution using an auxiliary operation called *block*, which encodes the conflict resolution policy defined by the programmer. A *block* is triggered atomically when a conflicting operation is executed and specifies which operations cannot run concurrently due to conflicts. The system then transforms any operation that conflicts with the *block* into a No-Op.

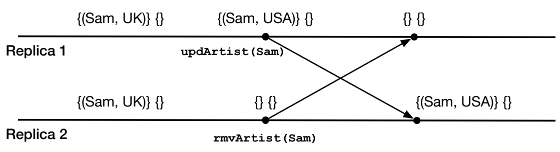


Figure 1 Convergence conflict.

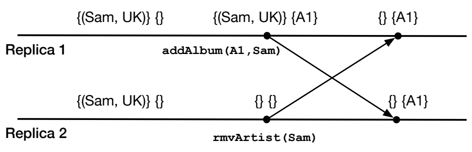


Figure 2 Invariant conflict.

By leveraging conflict resolution policies, the No-Op approach enforces both convergence and invariant preservation in a unified manner. Let us revisit Figure 1 and Figure 2 to see which conflict resolution policies could be applied to solve these conflicts. A conflict resolution policy for solving the convergence conflict in Figure 1 is to prioritise the remove operation by discarding the update operation. This behaviour aligns with the remove-wins policy used in existing remove-wins CRDT sets [40, 9]. Our No-Op approach can encode remove-wins semantics by having remove operations trigger a *block* on updates for the same artist. The *block* on updates will render any concurrent update on the deleted artist a No-Op.

An alternative policy for the convergence conflict is a privilege-based policy suitable for systems where users are assigned privilege levels. Under this policy, both update and remove operations trigger a *block*. An update operation triggers a *block* on all remove operations for the same artist issued by users with lower priority. Similarly, a remove operation triggers a *block* on all update operations for the same artist submitted by users of lower priority. When users share the same privilege level, the policy must specify how to resolve ties. For example, a total order could determine the winning operation among users with equal privilege levels.

Note that an operation only issues a *block* if it can win. For example, under the remove-wins semantics, the update operation never issues a *block* because it never wins against any other operation. In contrast, in a privilege-based policy, both update and remove operations issue a *block*, as either operation can be the winner depending on the user's privilege level.

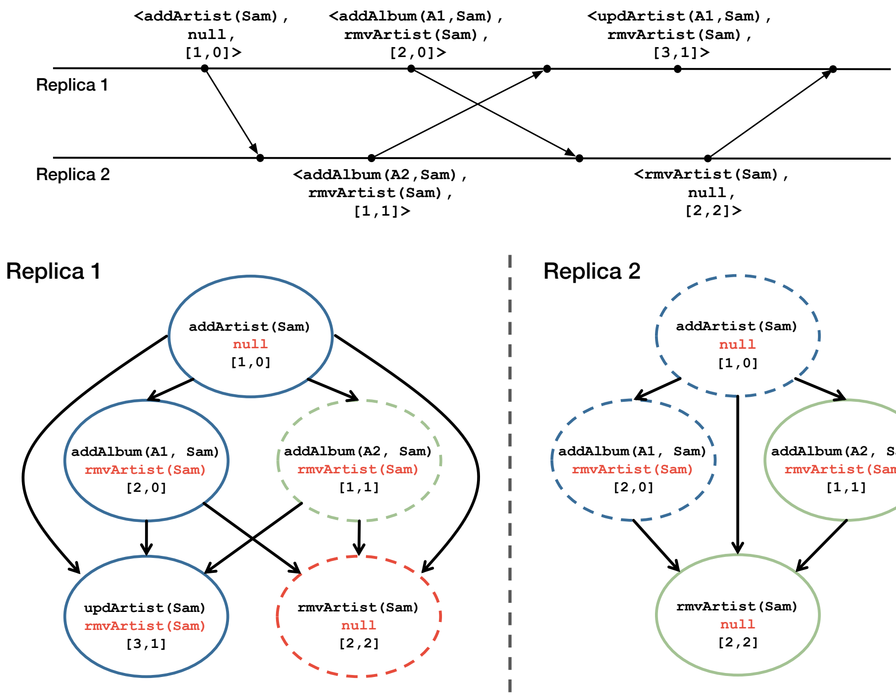
Similar to convergence conflicts, various conflict resolution policies can be applied to address invariant conflicts. In the case of the referential integrity conflict shown in Figure 2, we can prioritise the operation that adds the album, rendering the remove artist operation ineffective. This behaviour corresponds to an update-wins conflict resolution policy similar to the one used in Antidote SQL [35]. Alternatively, we can prioritise removing the artist by dropping the add album operation, corresponding to remove-wins semantics. These policies are implemented akin to the policies for handling convergence conflicts: the winning operation type issues a *block* on the losing operation type.

2.1 The No-Op Approach by Example

We will now illustrate how the No-Op approach works through a concrete execution trace of the album management system shown in Figure 3.

The execution trace consists of five operations involving two replicas. Replica 1 executes an `addArtist(Sam)` operation, which gets delivered to both replicas. Next, two concurrent `addAlbum` operations are executed and delivered to both replicas. Finally, two concurrent operations follow: `updateArtist` in replica 1 and `rmvArtist` in replica 2.

The No-Op approach stores operations in a graph to determine when a conflict occurs and applies the appropriate conflict resolution policy. This graph forms the core of the No-Op mechanism, representing a partial order of operations that accurately captures concurrency and causality of operations. Figure 3 depicts both replicas' execution graphs for the aforementioned execution trace. Solid circles represent local operations; dashed ones,



<addArtist(Sam), null, [1,0]>
<addAlbum(A1,Sam), rmvArtist(Sam), [2,0]>
<updArtist(A1,Sam), rmvArtist(Sam), [3,1]>
Replica 1
Replica 2
<addAlbum(A2,Sam), rmvArtist(Sam), [1,1]>
<rmvArtist(Sam), null, [2,2]>
Replica 1
addArtist(Sam) null [1,0]
addAlbum(A1, Sam) rmvArtist(Sam) [2,0]
addAlbum(A2, Sam) rmvArtist(Sam) [1,1]
updArtist(Sam) rmvArtist(Sam) [3,1]
rmvArtist(Sam) null [2,2]
Replica 2
addArtist(Sam) null [1,0]
addAlbum(A1, Sam) rmvArtist(Sam) [2,0]
addAlbum(A2, Sam) rmvArtist(Sam) [1,1]
rmvArtist(Sam) null [2,2]

Figure 3 Execution graph for an execution trace of the album management system.

delivered remote operations. Blue, green, and red indicate operations from replica 1, replica 2, and No-Ops, respectively. Each vertex lists the operation call, the *block* operation, and its timestamp. In this example, both referential integrity and convergence conflicts are resolved following the update-wins semantics.

The first node in the operation graph corresponds to the `addArtist(Sam)` operation, which got delivered to both replicas before any subsequent concurrent operations were executed, causally preceding all other operations in the graph. Since the operation was originally executed at replica 1, the node in replica 2 for this operation is dashed. Following this, the two replicas concurrently execute two `addAlbum` operations. Both operations issue a *block* on concurrent `rmvArtist(Sam)` to uphold the referential integrity invariant (according to the update-wins semantics). However, the operations do not conflict, as neither operation blocks the other. Consequently, after propagation, both graphs have two concurrent `addAlbum` operations (one local and one remote), but neither operation is marked as a No-Op.

Lastly, two additional operations are executed concurrently. Replica 1 executes an `updArtist(A1, Sam)` operation, which issues a *block* on concurrent `rmvArtist` on the updated artist, namely `Sam`. Concurrently, replica 2 executes a `rmvArtist(Sam)` operation that does not block any operation. After this operation is propagated to replica 1, the conflict between the *block* on `rmvArtist(Sam)` and the delivered `rmvArtist(Sam)` operation is detected, marking the latter as a No-Op to resolve the conflict. Since the `updArtist` operation has not yet been delivered to replica 2, the graphs currently diverge. Once the `updArtist` operation is delivered to replica 2, the system will also detect the conflict, and the local `rmvArtist(Sam)` operation will be marked as a No-Op.

Note that there are no edges between concurrent operations (e.g., the two concurrent `addAlbum` operations), so there are different possible serialisations of the execution graph. For example, in the graph of replica 2, there are two possible serialisations: i) `addArtist(Sam)`, `addAlbum(A1, Sam)`, `addAlbum(A2, Sam)`, and `rmvArtist(Sam)`, and ii) `addArtist(Sam)`, `addAlbum(A2, Sam)`, `addAlbum(A1, Sam)`, and `rmvArtist(Sam)`. However, as we prove in Section 3.1, all serialisations are safe and equivalent.

2.2 The No-Op Approach Workflow

We now present the envisioned workflow for developing replicated systems that guarantee state convergence and maintain application invariants using the No-Op approach. This workflow follows a three-step process, which we outline below.

Step 1. Identifying Conflicting Operations. Given the application state and the operations, programmers start by identifying conflicting operations within the system. We assume programmers can leverage existing analysis tools, such as Hamsaz [24] or Ordana [18], to detect conflicts between operations automatically.

Step 2. Defining Conflict Resolution Policies. For each pair of conflicting operations identified by the analysis tool, programmers must define a conflict resolution policy. These policies determine how conflicts are resolved, ensuring consistency while preserving the intended system behaviour. As noted in [37] and illustrated earlier in this section, different concurrency semantics can be applied to solving a conflict, with some being more suitable for specific applications. Our prototype implementation offers a range of conflict resolution policies (e.g., add-wins, remove-wins, last-writer-wins) as shown later in Section 5. These predefined policies enable programmers to adapt them when implementing new data types. Section 4.1.1 shows conflict resolution policy examples.

Step 3. Verification. After defining all conflict resolution policies, we foresee programmers verifying the correctness of the data type or application. In our work, we use VeriFx [17] to validate the correctness of our implementation, though other verification tools could be used. Note that correctness is independent of the conflict resolution semantics, i.e., VeriFx does not validate whether a conflict resolution policy aligns with the programmer's intended semantics. For instance, if a programmer intends to implement an add-wins set but defines a remove-wins policy, the system will enforce the policy as specified without detecting a mismatch. Since we aim to provide flexibility in defining conflict resolution policies, we assume that the programmer correctly specifies policies that align with the application's intended behaviour. To simplify, developers can use or adapt our predefined standard policies covering common concurrency semantics.

3 Model

We now present our model and provide a formal proof of its correctness. The No-Op framework ensures strong convergence and safety, guaranteeing that replicas remain consistent while preserving application invariants without coordination. Strong convergence [39] ensures that correct replicas processing the same set of updates – potentially in different orders – reach equivalent states. Safety ensures that the replicated state consistently upholds the invariants.

We consider a causally consistent database fully replicated across multiple data centres. Replicas may also be stored in mobile devices, which may temporarily be offline. We assume a fail-recover model that preserves durable state and excludes Byzantine faults.

The application state comprises a set of objects, each of which can be accessed and modified at any time through application-defined operations. In turn, the replica state comprises two components: i) the initial application state, and ii) a partial order of operations that tracks concurrency and allows the identification of conflicts as they arise. The current application state of a replica is calculated by applying all operations in partial order to the initial state.

Algorithm 1 Execution model of an update operation under the No-Op approach.

```
   types
      $m = \langle op, block \rangle$        ▷ a message is a pair containing operation $op$ and the $block$ operation
1: on operation$_i(op)$
2:    $m \leftarrow$ PREPARE$_i(op)$
3:    if $m \neq \bot$ then        ▷ $m$ is set to $\bot$ when the operation's precondition is not satisfied
4:       bcast$_i(m)$
5: on deliver$_i(m, t)$        ▷ we assume causal delivery
6:    $c \leftarrow \langle op, block, t \rangle$   ▷ call $c$ contains the operation $op$, the $block$ operation, and the timestamp $t$
7:    EFFECT$_i(c)$
```

An operation consists of a sequence of reads and writes that a replica executes atomically. In line with causal consistency, operations are propagated between replicas using a broadcasting mechanism that guarantees causal delivery. Causally-related operations are ordered according to the "happens-before" relation [29], whilst concurrent operations remain unordered relative to each other. Consequently, the execution order of concurrent operations may differ across replicas, potentially leading to consistency conflicts. These conflicts fall into two categories: i) convergence conflicts, which occur when replicas reach different states by applying concurrent operations in different orders, and ii) invariant conflicts, which arise when concurrent operations violate application-specific invariants.

An operation is *conflicting* if it can cause convergence or invariant conflicts. As such, read-only operations, which do not modify the application state, are non-conflicting.

Algorithm 1 outlines the execution model of the No-Op framework. It draws inspiration from operation-based CRDTs [40] and divides the execution of an update operation into two phases: the prepare phase and the effect phase.

When a user invokes *op* on replica i, the operation$_i$ event on that replica is triggered. This event triggers the prepare phase, executed exclusively on the replica where the operation was invoked. During this phase, the system validates the operation's preconditions. If the preconditions are satisfied, the PREPARE$_i$ returns a message m containing the operation *op* and a *block* operation, codifying the conflict resolution policy to handle conflicts involving the submitted operation. Based on the predefined policy, the *block* specifies which operations cannot be executed concurrently with the operation that issued it. In other words, the *block* encodes the less preferred operations. Consequently, there may be operations where the *block* is null, meaning that it represents the losing operation in the conflict. Conversely, an operation may have a *block* on multiple operations, indicating that it conflicts with and prevails over several others. Then, bcast$_i$ broadcasts the message to all replicas by calling the reliable causal broadcast API (which will add a timestamp to the message).

When a message m with timestamp t is delivered to replica i, the deliver$_i$ event is triggered. The event first creates a tagged call c, associating the message with its timestamp. As the message includes the operation call and the issued *block*, the tagged call is a triple consisting of the operation call, the *block*, and the timestamp. EFFECT$_i$ incorporates the tagged call (or simply call) into the partial order of operations and resolves any resulting local conflicts. A local conflict occurs when the replica detects that the delivered operation has a *block* against a local operation or vice versa. When the conflict is detected, the system transforms the blocked operation into a no-side-effect operation, called a No-Op. This transformation ensures that only the preferred operation (the one that issued the *block*) affects the application state, thus maintaining consistency and enforcing the predefined conflict resolution policy. Conflicts can also arise with operations that have already been transformed into No-Ops. For instance, if a No-Op has a *block* for a newly received operation, a conflict exists between the two operations, and the system resolves it by transforming the new operation into a No-Op.

3.1 Model Correctness

We now prove that the No-Op framework guarantees strong convergence and safety. To do so, we assume three key prerequisites: i) all replicas receive all updates, ii) updates are delivered in causal order, and ii) a resolution policy is defined for every pair of conflicting operations.

As mentioned before, our model represents the partial order of operations as an execution graph, $G = \langle V, E\rangle$. G is a labelled directed acyclic graph (DAG) where vertices (V) represent tagged operation calls, and the edges (E) denote sequential dependencies between these calls. No-Ops are maintained in a set N, which consists of all the vertices marked as No-Ops. Concurrent operations remain unconnected in the graph, allowing for multiple topological orders. We prove that all topological orders are safe serialisations (Theorem 5) and that replicas receiving the same set of calls will converge to the same state (Theorem 7).

For simplicity, the proofs focus on the effect phase, as the prepare phase only creates the message to be broadcast to all replicas and does not modify the replica state. We assume the prepare phase has been executed, and the *block* has been issued according to the predefined conflict resolution policies. Additionally, the proofs do not distinguish between local and remote replicas, as only $\textsc{effect}_i$ modifies the state.

▶ **Lemma 1.** *Two replicas that observed the same calls have the same execution graph and the same set of No-Ops:*

$$\forall r_1 = \langle G_1, N_1\rangle,\ r_2 = \langle G_2, N_2\rangle :$$
$$(G_1 = \langle V_1, E_1\rangle \land G_2 = \langle V_2, E_2\rangle \land V_1 = V_2) \implies (E_1 = E_2 \land N_1 = N_2)$$

Proof. The graphs are equivalent iff, upon receiving the same set of calls, they contain the same vertices and edges. When executing the effect phase, the replica adds the call to its execution graph. Consequently, two replicas that observe the same calls will have identical vertices in their execution graphs. We now show that the graphs contain identical edges across replicas. The effect phase computes the edges between sequential calls by comparing each new call c with every other call in the graph. Causal consistency ensures that operations are causally delivered, allowing all replicas to process sequential operations in the same order. As a result, all calls preceding c are consistently compared to c, generating the necessary edges. Thus, all replicas maintain the same edges in their execution graphs. As the vertices and the edges are the same, we can infer that both graphs, G_1 and G_2, are identical.

Lastly, we demonstrate that even when replicas process concurrent calls in different orders, the model marks the same operations as No-Ops. When a tagged call c is added to the graph, the model checks whether c conflicts with its concurrent operations. For each pair of concurrent operations, we distinguish two scenarios. First, if one operation issues a *block* against the other, the blocked operation is marked as a No-Op. Second, if the operations do not conflict, no further action is taken. Since we assume that all operations are eventually delivered, all updates are processed, ensuring all blocks are also processed. Consequently, after processing the same set of operations, all replicas will have identified the same No-Ops. ◀

To demonstrate that all topological orders are safe serialisations, we first define safety based on the definition from [18] adapted to our model.

▶ **Definition 2** (Safe calls). *Two concurrent calls are safe iff applying them in any order preserves the operations' preconditions and invariants. Otherwise, they are unsafe.*

▶ **Definition 3** (Safe operations). *Two operations are safe iff all pairs of concurrent calls to those operations are safe.*

▶ **Definition 4** (Safe serialisation). *A serialisation of correct calls is safe iff every pair of concurrent calls is either safe or includes at least one call that has been marked as a No-Op.*

▶ **Theorem 5.** *All topological orderings of an execution graph G are safe serialisations.*

Proof. By induction on the length of the topological order.

Base Case. No calls occurred, so the topological ordering is empty and trivially safe.

Induction Step. Assume a safe topological order of dimension n derived from an execution graph. When a message is delivered, the effect phase incorporates the new call into the execution graph and calculates the No-Ops according to the issued blocks. We distinguish two cases: either the new call is safe or unsafe.

Case 1. If the new call is safe, indicating no conflicts with any other concurrent call in the execution graph, we can add the call to the graph, ensuring that the resulting topological order(s) are safe serialisations.

Case 2. If the new call is unsafe, the new call invariant conflicts with at least a concurrent operation in the execution graph. Since the call is conflicting, it either appended a *block* during the prepare phase or the conflicting calls did. The blocked operations are then transformed into No-Ops, ensuring that the resulting topological order(s) are safe serialisations. Therefore, assuming the prepare phase was correctly executed, and conflict resolution policies are defined for all conflicts, a *block* must exist for every pair of conflicting operations, ensuring safety.

Starting from an execution graph with a safe topological order of dimension n, our model builds an expanded graph whose topological order of dimension $n+1$ is a safe serialisation. ◀

▶ **Lemma 6.** *Two topological orderings from the same graph G converge to equivalent states.*

Proof. We follow a proof by contradiction. Assume two topological orderings, t_1 and t_2, that result in divergent states. We know that, in both $t1$ and $t2$, all sequential operations appear in the same order. However, concurrent operations may appear in different orders. Given the assumption that the states diverge, there must be at least two concurrent non-commutative calls, c_1 and c_2, appearing in different orders in t_1 and t_2. Let us assume, without loss of generality, that $t_1[c_1] < t_1[c_2]$ and $t_2[c_1] \nless t_2[c_2]$, where $t_i[c_i]$ represents the position of call c_i in the topological ordering t_i. Since c_1 and c_2 are non-commutative, executing them in different orders would lead to divergent outcomes. However, according to the conflict resolution policy, non-commutative operations will issue a *block* in the prepare phase. Therefore, one operation will block the other for all pairs of non-commutative conflicting operations. Consequently, either c_1 or c_2 would become a No-Op. As No-Ops do not affect the application state, it follows that c_1 and c_2 commute, which contradicts our initial assumption that c_1 and c_2 are non-commutative and lead to divergent states. By processing all operations in the graph this way, we ensure that all remaining operations are commutative, guaranteeing that any topological ordering will yield equivalent results. ◀

▶ **Theorem 7.** *Two replicas that observed the same tagged calls converge to equivalent states.*

Proof. Assuming both graphs have observed the same calls, it follows from Lemma 1 that they have the same execution graph. Additionally, by Lemma 6, any topological order within the same graph leads to an equivalent state. Considering the graphs are equal, we can infer that their topological orders are equivalent, thus converging to an equivalent state. ◀

4 No-Op Replication Protocol

We now introduce a coordination-free replication protocol that leverages the No-Op approach to ensure strong convergence and uphold invariants in distributed systems without requiring coordination. The protocol enables replicas to execute operations without synchronisation and handles conflicts as they arise. As explained in Section 2, conflicts are identified and resolved using a *block* operation that encodes a programmer-defined resolution policy. After resolving the conflict, *at most one* of the conflicting operations affects the state.

Algorithm 2 provides an overview of the replication algorithm that implements the model proposed in Section 3. Each replica maintains an execution graph, a No-Op (losing calls) set, and the application's initial state. As in the model, the execution graph is a labelled directed acyclic graph (DAG) in which vertices represent tagged operation calls, and edges denote the sequential dependencies between them. When two concurrent operations conflict – one operation blocks the other – the algorithm designates the blocked operation as a No-Op, ensuring that only the preferred operation affects the application's state.

Since each replica only maintains the initial state, the current state of the application must be calculated based on the operations in the execution graph. The GETSTATE function computes the current state by executing the operations according to the partial order maintained in the graph. The function begins by determining a topological order of the graph. Starting from the initial state, each operation in the topological order is applied, except for No-Op calls, which are ignored and left unexecuted.

The PREPARE function, executed exclusively by the replica where the operation was submitted, verifies whether the operation is enabled in its current state and, if necessary, issues a *block*. The function begins by calculating the replica's local state and evaluating the precondition. If the precondition does not hold, the operation cannot execute, and the function returns $\perp$. Otherwise, it returns a message containing the operation and the corresponding *block*. The *block* is determined based on the operation, the current state, and the predefined conflict resolution policies. In some cases, the application's state does not influence the *block*. However, specific policies may use the state to decide which operations to block (as discussed later in Section 5.2.4).

The EFFECT function operates on all replicas and incorporates the received call into each replica's local operation graph. The function begins by adding the edges between the new call c and all preceding calls in the graph. Since the model assumes causal delivery, no calls occurring after c exist in the graph at this stage. Next, the function checks if the new call conflicts with its concurrent operations. Conflicts are detected using the OPSCONFLICT function, which returns *true* if one operation has a *block* on the other. If a conflict is detected, the algorithm gets the blocked operation through the GETBLOCKEDOP function and adds it to the No-Ops set. Finally, the function adds the call to the set of vertices.

The algorithm allows a replica to validate an operation's precondition while concurrently delivering a remote operation. This is not a problem, as any invalidation of the precondition by the remote operation indicates a conflict, which the system detects and resolves when the effect function integrates the local operation into the graph.

Algorithm 2 The No-Op algorithm main functions.

```
replica local state
    $G = \langle V, E \rangle$                                  ▷ execution graph
    $N$                                                         ▷ set of calls marked as No-Ops
    $\sigma_0$                                                  ▷ initial state
1: function PREPARE($op$)                                      ▷ check if an operation can be executed in the current state
2:     $\sigma \leftarrow$ GETSTATE()                           ▷ calculate the current state based on the execution graph
3:     if PRE($op, \sigma$) then                                ▷ check if the operation precondition holds in the current state
4:         $b \leftarrow$ GETBLOCK($op, \sigma$)                ▷ generate the *block* according to the conflict resolution policy
5:         return $\langle op, b \rangle$                       ▷ return the operation and the *block* if the precondition is satisfied
6:     else
7:         return $\perp$                                       ▷ indicate the operation is not enabled in the current state
8: function EFFECT($c$)                                        ▷ integrate call $c$ into the execution graph
9:     for $v \in V$ do                                         ▷ determine edges and No-Ops generated by call $c$
10:        if $v \prec c$ then                                  ▷ call $v$ happens-before call $c$
11:            $E \leftarrow E \cup \{\langle v, c \rangle\}$   ▷ add edge between call $v$ and call $c$
12:        else if $v \parallel c \wedge$ OPSCONFLICT($v, c$) then   ▷ operation blocks the concurrent operation
13:            $l \leftarrow$ GETBLOCKEDOP($v, c$)              ▷ get the blocked operation
14:            $N \leftarrow N \cup \{l\}$                      ▷ add the losing call $l$ to the No-Op set
15:    $V \leftarrow V \cup \{c\}$                              ▷ add call $c$ to the graph vertices
16: function GETSTATE()                                        ▷ calculate the current state
17:    $to \leftarrow$ GETTOPOLOGICALORDER()                    ▷ get a topological order from the graph
18:    $\sigma \leftarrow \sigma_0$
19:    for $c \in to$ do                                        ▷ apply all calls according to the topological order
20:        if $c \notin N$ then                                 ▷ No-Ops are ignored and are not executed
21:            $\sigma \leftarrow$ EXECUTE($\sigma, c$)         ▷ execute call $c$ over state $\sigma$
22:    return $\sigma$                                          ▷ return the current state
```

4.1 Implementation in VeriFx

We implemented the core of the No-Op replication protocol in VeriFx [17], a programming language for RDTs with automated proof capabilities. VeriFx allows developers to implement RDTs in a high-level language atop functional collections and express correctness properties that are verified automatically. Alongside the algorithm implementation, we developed formal proofs to automatically verify the algorithm's correctness properties when applied to specific data types or application implementations. The implementation considers only the effect phase of Algorithm 2. As in the formal proofs in Section 3.1, we assume the prepare phase was correctly executed according to the programmer-defined conflict resolution policies.

4.1.1 Conflict Resolution Policies Specification

We start by detailing how conflict resolution policies are specified in our VeriFx implementation. These policies are implemented by means of two functions:

- The `blocksEncoding` function verifies whether an operation call is valid. It takes `TaggedOp` type, combining the operation and a corresponding *block*, and defines the *block* value according to the operation type. Since we assume that the prepare phase was correctly executed and VeriFx will generate any possible combination of operation and *block*, we use this function to restrict the set of operations considered in the proofs to those that are correct (i.e., those whose blocks match GETBLOCK in Algorithm 2). Programmers must define this function.

- The `isBlockedBy` function takes two calls and defines the condition under which a given operation instance is blocked by another. It corresponds to the opsConflict in Algorithm 2. Typically, an operation blocks all instances of another operation of a given type. Accordingly, the default implementation of this function returns `true` if a given operation has a *block* that matches the other operation. For some data types (e.g., the LWW Register), an operation may not block all instances of another type. In such cases, the programmer must provide their custom implementation.

The implementation of `blocksEncoding` and `isBlockedBy` depends on the data type semantics. We now illustrate how programmers specify conflict resolution policies through these functions in the context of two concrete data types.

Add-Wins Set. The implementation of `blocksEncoding` for a replicated set defining an add-wins policy is shown below:

```
def blocksEncoding(v1: TaggedOp[Time, ID, SetOp[V]]): Boolean =
  v1.op match {
    case Add(e) => v1.b == new Rmv(e)
    case Rmv(_) => v1.b == new NullOp[V]()
    case _ => false
  }
```

Given an operation, the `blocksEncoding` function encodes how blocks should have been assigned during the prepare phase. In this case, if the operation is an `Add(e)`, the *block* is equal to `Rmv(e)`. Conversely, if the operation is a `Rmv(_)`, the *block* is null (i.e., the operation does not block any operation) since it represents the losing side of the conflict. In VeriFx, we represent this null value using a special operation called `NullOp`. Accordingly, for the case of `Rmv`, the function returns `true` if the associated *block* is `NullOp`. The last case ensures pattern matching covers all possible scenarios, including `NullOp`. Since `NullOp` does not correspond to an actual operation, there are no operations of this type, and the function returns `false`. This encoding guarantees that only operations with correctly triggered blocks are considered. For example, it ensures that in the proofs we only consider `Add(e)` operations that correctly blocks `Rmv(e)`, and excludes any `Add(e)` that does not block any operation.
In the add-wins set, the *block* matches precisely the operation that should be transformed into a No-Op, i.e., the removes on the same element. Therefore, we can rely on the default implementation of the `isBlockedBy` function.

LWW Register. Listing 1 shows the specification of a register defining a last-writer-wins (LWW) policy. The `blocksEncoding` function defines that an `Assign` operation must block other concurrent `Assign` operations that assign the same value to the register (i.e., `Assign(5)` would only block concurrent `Assign(5)` operations). However, to implement the LWW policy, an `Assign` operation should block all operations lower in the total order for the LWW, regardless of the assigned value. For instance, a *block* on `Assign(5)` must block all concurrent instances of `Assign(e)` operations that are older in the total order, independently of the value of `e`. We specify this requirement in the custom implementation of the `isBlockedBy` function where blocking is defined by operation identifiers (`opID`). As shown in lines 8–10, the operation in the first call (`op1.op`) is considered blocked if both `op1.op` and the *block* of the second call (`op2.b`) are of type `Assign(_)`, regardless of their arguments (denoted by the underscore in pattern matching), and if the identifier of the first call (`op1.opID`) is smaller than that of the second (`op2.opID`). Line 14 ensures exhaustive pattern matching. In the LWW register, it only matches `NullOp`, which is never triggered and thus never evaluated.

Listing 1 Conflict resolution policy specification for a last-writer-wins register.

```
def blocksEncoding(v1: TaggedOp[Time, ID, RegOp[V]]): Boolean =
  v1.op match {
    case Assign(x) => v1.b == new Assign(x)
    case _ => false
  }

override def isBlockedBy(op1: TaggedOp[Time, ID, RegOp[V]],
                         op2: TaggedOp[Time, ID, RegOp[V]]) = {
  op1.op match {
    case Assign(_) => op2.b match {
      case Assign(_) => this.smaller(op1.opID, op2.opID)
      case _ => false
    }
    case _ => false
  }
}
```

4.1.2 Implementation and Verification of the No-Op Replication Protocol

Our implementation closely follows the formal model, with a few optimisations. In the model, the prepare phase assigns blocks according to conflict resolution policies, and all *block* operations are issued during this phase. In the VeriFx implementation, we assume the prepare phase has been correctly applied. We encode this assumption using the `blocksEncoding` function, which specifies which operations are invalid and should not be considered in the verification process. Additionally, rather than issuing all *block* operations, our implementation only allows one type of concurrent operation to be blocked. Therefore, we need the `isBlockedBy` function to determine whether a specific operation instance is blocked.

To prove the algorithm's correctness, we must guarantee that the implementation upholds the properties of the model proposed in Section 3: strong convergence and safety. In VeriFx, we encoded these properties through two graph-based properties. The first property aligns with the property proven in Lemma 1, demonstrating that applying two concurrent operations in different orders on the same state produces an identical graph and the same No-Ops. We only need to verify this property for concurrent operations, as sequential operations are processed in the same order due to the assumption of causal delivery. The second property encodes the structural characteristics of the graph. It ensures that edges connect all sequential operations and that, for any conflicting pair of concurrent operations, one operation is marked as a No-Op and added to the set of No-Ops. This second property encodes the properties of Theorems 5 and 7.

Listing 2 presents the proofs used to verify the algorithm's correctness. The proofs employ three helper functions present in VeriFx to facilitate the verification process of RDTs:

- The `compatible` function returns true if the two operations denote different operations.
- The `enabledOp` function assesses whether an operation is enabled in the current state. It first checks that the operation has not yet been delivered and that all predecessor operations have already been delivered. This verification is needed because our model assumes causal delivery of operations. Additionally, the function ensures the tagged operation call is correct using the `blocksEncoding` function.
- The `reachable` function defines the states the system can attain. It integrates the structural properties of the graph, ensuring that conflicts between concurrent operations are appropriately resolved and that edges exist between sequential operations.

Listing 2 Proof verifying the algorithm's correctness under concurrent operations, leveraging graph-based properties.

```
proof noOpStrongConverges[Time, ID] {
  forall(s: T[Time, ID], x: TaggedOp[Op, ID, Time],
                         y: TaggedOp[Op, ID, Time]) {
    (s.compatible(x, y) && s.reachable() &&
     s.enabledOp(x) && s.enabledOp(y) &&
     s.execGraph.timeProps.concurrent(x.stamp, y.stamp)) =>:
       s.effect(x).effect(y).equals(s.effect(y).effect(x))
  }
}

proof noOpIsSafe[Time, ID] {
  forall(s: T[Time, ID], x: TaggedOp[Time, ID, Op]) {
    (s.reachable() && s.enabledOp(x)) =>: s.effect(x).reachable()
  }
}
```

The proof `noOpStrongConverges` ensures that starting from a reachable state with two concurrent operations enabled, applying these operations in different orders results in the same graph state. The proof `noOpIsSafe` verifies that integrating an enabled operation into the graph preserves the safety properties encoded in the `reachable` function.

Like our manual proofs (cf. Section 3.1), the VeriFx proofs assume that all possible conflicts have a defined conflict resolution policy. Thus, the analysis may incorrectly indicate that the data type is correct if the resolution policies do not cover all conflicts.

We used the VeriFx implementation of our approach to implement some data types (discussed later in Section 5) and successfully verified their correctness. In this work, data type implementations are used exclusively for verification purposes. However, VeriFx also supports the transpilation of the data types into mainstream languages such as Scala and JavaScript. Thus, if the No-Op algorithm were integrated into a database, developers could design and verify their data types in VeriFx, subsequently transpiling them into a mainstream language for seamless database integration.

4.2 Graph Compaction

Although the algorithm maintains consistency properties, i.e., ensuring convergence and preserving invariants, it is impractical for real-world use due to the linear growth of each replica's state with the number of operations. To make our approach practical and more efficient, we leverage the concept of causal stability [9] to determine when an operation call can be committed and removed from the graph. This optimisation reduces space and prevents the need to reapply the entire operation history for every incoming call. By committing causally stable calls, we ensure that the graph remains compact, improving efficiency while preserving the correctness guarantees of the original algorithm.

An operation is *causally stable* at a replica i when the replica can no longer receive any additional concurrent operations. Once stabilised, the operation can no longer be transformed into a No-Op nor cause other operations to become No-Ops. Therefore, replicas can locally commit causally stable calls without requiring coordination.

The COMMITSTABLECALLS function, described in Algorithm 3, implements the causal stabilisation of our approach. The function takes a stable timestamp t as its argument and begins by identifying the stable calls in the graph. The GETSTABLECALLS function identifies all calls with a timestamp less than or equal to t and returns them as a sequence of operations. This sequence respects the causal ordering defined by the graph. Next, each stable call in the sequence is removed from the graph and applied to the state σ_0. If an operation is a

Algorithm 3 Commiting causally stable calls.

```
1: function COMMITSTABLECALLS($t$)
2:     $stableCalls \leftarrow$ GETSTABLECALLS($t$)                 ▷ get the stable calls
3:     for $c_s \in stableCalls$ do          ▷ we assume $stableCalls$ respect causal ordering
4:         $V \leftarrow V \setminus \{c_s\}$                 ▷ remove the stable call from the vertices
5:         $E \leftarrow E \setminus$ IN($c_s$) $\setminus$ OUT($c_s$)          ▷ remove incoming and outgoing edges
6:         if $c_s \notin N$ then                    ▷ the stable call is not a No-Op
7:             $\sigma_0 \leftarrow$ EXECUTE($\sigma_0, c_s$)      ▷ apply the stable call $c_s$ in the state $\sigma_0$
8:         else                                  ▷ the stable call is a No-Op
9:             $N \leftarrow N \setminus \{o_s\}$        ▷ remove the stable call from the set of No-Ops
```

No-Op, it does not affect the stable state, as it has no side effects.

Some applications may benefit from being notified when operations are committed. For example, in an online sales application, a developer may want a purchase to remain in an *accepted pending* state until it is final. Only after the operation is committed (and can no longer be modified) should a notification be sent to the customer, confirming that the purchase has been successfully processed. To support this behaviour, our framework can be extended to let developers register callbacks on operations, which are triggered asynchronously when the operation is committed. Note that the callback notification can be sent as soon as a single replica commits the operation; there's no need to wait for every replica to commit.

4.2.1 Causal Stabilisation and No-Op Operations

During execution, conflicts are resolved by transforming operation calls into no-side-effect operations, known as No-Ops. Since conflicts are handled locally by each replica as they arise, different replicas may detect and resolve them in different orders due to variations in operation arrival. To ensure determinism, the computation of No-Ops must consider all concurrent operations, including those already marked as No-Ops. Consequently, although No-Ops do not modify the application state, they must remain in the graph until causal stabilisation to ensure deterministic conflict resolution and prevent divergence.

To illustrate why No-Op operations must remain in the graph, consider the example in Figure 4, based on the albums management system introduced in Section 2. Initially, the database contains an artist named *Sam*. Now, consider three concurrent operations: `addAlbum(A1, Sam)`, which adds an album *A1* by *Sam*; `rmvArtist(Sam)`, which removes *Sam* from the database; and `updArtist(Sam)`, which updates *Sam*'s information. The operations are executed on replicas 1, 2, and 3, respectively. For simplicity, the figure depicts only two of the three replicas, showing their stable and current state. Replica 1 first executes `addAlbum(A1, Sam)`(denoted as `c1` in the graph), and then it receives the `rmvArtist(Sam)` operation. When the effect phase for `rmvArtist(Sam)` is triggered, replica 1 detects a conflict between the operation and the *block* issued by call `c1`. Consider that the conflict is resolved by removing `c2` from the graph (as the operation would be marked as a No-Op). When `updArtist(Sam)` is delivered to replica 1, it is added to the graph without conflict.

Let us turn our attention to the execution graph in replica 2. Consider that replica 2 executed the `rmvArtist(Sam)` operation, but its effect phase is only triggered after the effect phase of the two remote operations: `addAlbum(A1, Sam)` and `updArtist(Sam)`. First, `addAlbum(A1, Sam)` is processed and added to the graph (denoted as `c1`). Later `updArtist(Sam)` is also added (denoted as `c3`) as the operation does not conflict with `c1`. However, two conflicts arise when the effect phase of `rmvArtist(Sam)` is triggered. The *block*

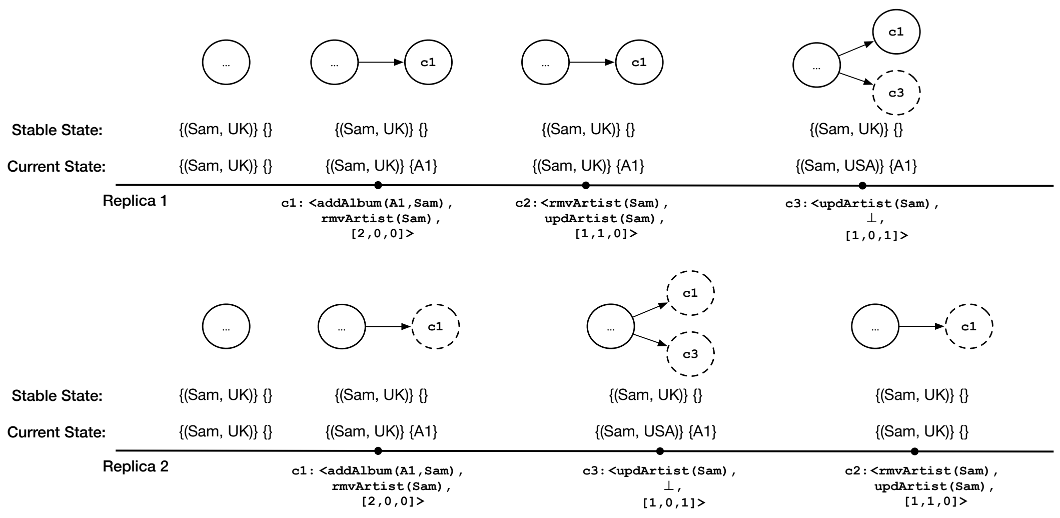

Stable State:
Current State:
{(Sam, UK)} {}
{(Sam, UK)} {A1}
{(Sam, USA)} {A1}
Replica 1
c1: <addAlbum(A1,Sam), rmvArtist(Sam), [2,0,0]>
c2:<rmvArtist(Sam), updArtist(Sam), [1,1,0]>
c3:<updArtist(Sam), ⊥, [1,0,1]>
Replica 2
c1
c3

Figure 4 Convergence problems caused by removing No-Op operations from the graph before stabilisation.

on `updArtist(Sam)` issued by call `c3` leads to the removal of call `c2` from the graph. On the other hand, the *block* issued by `addAlbum(A1, Sam)` on `rmvArtist(Sam)` also results in call `c2` being removed from the graph. After executing the same set of operations, the state of the replicas diverges. The problem is that removing operations from the graph when marked as a No-Op before they are causally stable does not guarantee convergence. As such, our algorithm retains unstable No-Op operations in the graph. Maintaining these operations, along with comparing each operation against all its concurrent operations – including those already marked as No-Ops– ensures consistent conflict resolution and prevents divergence.

To optimise metadata storage in the graph, we can replace the operation marked as No-Op with null (i.e., store $\langle \perp, \mathit{block}, \text{timestamp}\rangle$). This optimisation is possible because the only information required for a No-Op is the associated *block* operation and the timestamp to detect causal stability. Retaining the operation information is thus unnecessary since a No-Op does not affect the system state.

5 Applicability of the Solution

This section demonstrates how the No-Op framework supports the implementation of CRDTs (Section 5.1), enforces a wide variety of application invariants (Section 5.2), and enables the development of distributed geo-replicated applications such as the RUBiS system (Section 5.3).

The No-Op approach supports a wide range of conflict resolution policies, allowing flexibility in handling concurrent operations. Examples include operation-type-based policies, such as update-wins and delete-wins, where one type of operation takes precedence over the other. Additionally, it supports total-order-based policies, such as the last-writer-wins and first-writer-wins policies, which resolve conflicts based on a total order of operations. Finally, privilege-based policies can resolve conflicts based on criteria, such as user roles, giving precedence to operations initiated by higher-privileged users within the system. Besides these static policies, the framework supports state-dependent policies, where decisions are made based on the system's current state. Ultimately, our mechanism's generality allows programmers to define any policy expressible in terms of blocks.

All policies and mechanisms outlined in this section were implemented and verified using VeriFx. This confirms the correctness of our approach and demonstrates its practicality for real-world applications.

5.1 CRDTs

Conflict-free Replicated Data Types (CRDTs) [40] are the most well-known class of RDTs. They automatically resolve conflicts by leveraging mathematical properties to ensure convergence by design. These data types can be classified into two categories: those with commutative operations, which naturally avoid conflicts, and those with non-commutative operations, which require conflict resolution policies.

Commutative data types inherently avoid convergence conflicts and do not require conflict resolution policies. Under causal delivery assumptions, if all operations are commutative, they can be applied directly to the state without needing an execution graph. Examples of such data types include the grow-only set [40] and the counter [40]. Since these data types are conflict-free, no operations become No-Ops, making them immediately causally stable and directly applicable to the stable state. In other words, these data types can be implemented as a sequential set and a sequential counter, respectively.

In contrast, non-commutative data types require embedding a conflict resolution policy within the data structure to prioritise specific operations when conflicts arise. These CRDTs leverage conflict resolution policies, making their implementation straightforward with the No-Op framework. For instance, to implement an add-wins set, the conflict resolution policy specifies that add operations take precedence over remove operations on the same element (cf. Section 4.1.1). During the prepare phase of an add operation, the No-Op framework generates a *block* on concurrent removes targeting the same element. As a result, those remove operations are transformed into No-Ops.

Overall, CRDTs typically rely on operation-type-based or total-order-based policies, both supported by the No-Op framework, allowing diverse CRDT implementations. We have implemented in our VeriFx implementation different registers, flags, counters, and sets.

5.2 Invariant Maintenance

This section describes how our model can enforce a wide variety of invariants, ranging from simple invariants on single objects to more complex invariants over multiple objects. To illustrate invariant's examples, we use the albums management system presented in Section 2.

5.2.1 Equality and Inequality of Attributes

Equality and inequality of attributes [4] restrict the values assigned to an attribute. In our running example, an example of attribute equality is defining that the *genre* of an album can only take predefined values, such as *"rock"*, *"pop"*, *"jazz"*, or *"classical"*. This invariant type does not generate conflicts between operations, not requiring any conflict resolution policy to enforce it. Since this invariant does not generate conflicts, it is trivially maintained within our framework without requiring any conflict resolution policy.

5.2.2 Uniqueness

Uniqueness [4, 7, 6] includes all constraints regarding attribute uniqueness and is commonly used to denote primary keys or identifiers. For instance, in our running example, requiring artist identifiers to be unique is a typical uniqueness invariant.

We can classify the identifiers into system-generated identifiers and user-chosen identifiers. In system-generated identifiers, we assume a mechanism that automatically generates them, ensuring their uniqueness [7, 30]. By generating unique identifiers, the system guarantees that no conflicts related to identifier uniqueness will arise, eliminating the need for a policy to solve uniqueness conflicts. For user-chosen identifiers, two operations conflict if attempting to add distinct objects with the same identifier (insertions with different identifiers do not conflict). When n operations attempt to add different objects with the same identifier, $n - 1$ operations need to be converted into No-Ops, allowing only one operation to be successfully executed, thereby preserving the invariant. Therefore, the No-Op approach can resolve these conflicts through a total-order-based policy. Each operation blocks all preceding operations, as determined by a total order (e.g., enforced by Lamport clocks), that attempt to add a different object with the same identifier. As a result, after processing the operations, the No-Op framework converts all but one into No-Ops, ensuring the invariant is preserved.

5.2.3 Referential Integrity

Referential integrity [4, 7, 6] defines dependency relationships between objects, with a parent object representing the referenced entity and a child object referencing the parent. In the context of the running example, a referential integrity constraint could ensure that albums by nonexistent artists cannot exist in the system.

Conflicts arise when an operation inserts or updates an object that references a parent object which has been concurrently deleted. To manage these conflicts without coordination, Antidote SQL [35] introduced two conflict resolution policies: update-wins and delete-wins. Under the update-wins policy, the conflict is resolved by keeping the parent object in the database. In contrast, the delete-wins policy resolves the conflict by removing both the parent object and the concurrently inserted or updated child.

Our model supports the implementation of both policies. In the update-wins policy, insert and update operations on child objects take precedence by blocking the parent deletion, rendering it a No-Op. In the delete-wins policy, the parent deletion takes precedence, blocking insert and update operations on child objects and transforming them into No-Ops.

Furthermore, our model enables the implementation of variations of these policies by simply adjusting how the operations are defined. Consider the update-wins policy. If the operation that deletes the parent object also removes its children, applying the policy would prevent both actions from being executed. As a result, after resolving the conflict, both the parent and its children remain in the database. In contrast, if the operation that deletes the parent first triggers a separate operation to delete its children, the outcome changes. Since deleting the children is an independent operation, it does not conflict with inserting a new child. As a result, the parent remains in the database, while the deleted children do not. This behaviour aligns with the update-wins semantics proposed in Antidote SQL.

5.2.4 Disjunctions

A disjunction invariant requires at least one of several conditions to be true. A violation occurs when all true predicates concurrently transition to false. Therefore, conflicting operations are the operations that transition a predicate from true to false.

Consider an extension to the album management system where each album can contain multiple tracks but must always include at least one. This version introduces two additional operations: adding a track to an album and removing a track from an album. Given this property, operations that remove different tracks conflict, as they may violate the invariant. For example, if an album initially contains two tracks and two operations attempt to remove them concurrently, the album would end up with no tracks, which breaks the invariant.

The disjunction invariant illustrates how issuing blocks based on the system's state can be beneficial. The remove track operation is self-conflicting, which makes it responsible for triggering blocks. However, traditional policies, such as operation-type or total-order-based, are overly restrictive, limiting execution to at most one operation. Total-order-based policies allow only a single operation to proceed, while operation-based policies prevent all operations from executing, as self-conflicting operations block each other. However, enforcing a disjunction invariant only requires ensuring that at least one predicate remains true.

To address this, we propose a state-based policy where the *block* operation is issued based on the application's current state. This approach preserves the truth value of a predicate by transforming any operation that would transition it to false into a No-Op. By blocking all operations that would make the predicate false, the policy ensures that at least one predicate remains true, thereby maintaining the invariant.

In the example, the *block* operation would guarantee that a specific track could not be removed from an album. In this scenario, the *block* operation would block all operations that transition the predicate "track x is in album y" to false. Consequently, any operation that concurrently attempts to remove track x from album y would conflict with the *block* operation, as it alters the truth value of the blocked predicate. The conflicting remove operation would then become a No-Op.

The specification of such a state-based policy follows a similar approach to the LWW register shown in Section 4.1.1. The specification is provided in Appendix A in [12]. Assuming the application defines the predicate to be blocked and returns the selected predicate to the framework, the framework then issues blocks on all operations that would transition the predicate to false.

Issuing blocks based on the system's state enables the implementation of more fine-grained policies, reducing the number of conflicts. However, it is important to note that some operations may still be unnecessarily undone, depending on how the blocks are assigned. For instance, if the predicate to be blocked is chosen randomly, different operations may block multiple predicates, turning several operations into No-Ops. In contrast, if all operations attempt to block the same predicate, only the operations affecting that specific predicate are undone, minimising the number of No-Ops.

5.2.5 Numerical Invariants

Numerical invariants [4, 7, 6] are usually associated with attributes manipulated by increment or decrement operations, and where there must be some control over the values of the attribute since there is a lower or upper bound. An example of a numerical invariant is guaranteeing that a user's balance does not become negative.

Although the No-Op approach preserves numerical invariants, it can be overly restrictive. Due to the self-conflicting nature of the operations, they block each other and, in some cases, generate more No-Ops than necessary. Consider a counter with a lower bound of 0 and an initial value of 100. Decrements are inherently self-conflicting. Applying the No-Op approach would require decrements to trigger blocks on other decrements. If each decrement blocked another, none of the operations would execute, as the decrements would cancel each other out. Similarly, policies based on a total order would allow only one operation to succeed. For example, if five concurrent decrements of 10 units each were issued, only one decrement would be applied (e.g., the latter following the total order). This behaviour is overly restrictive, as executing all decrements would maintain the invariant and respect the lower bound.

To overcome this limitation, we propose integrating escrow techniques [36, 7, 8] into the model. These techniques enable replicas to pre-allocate permissions for executing future updates, ensuring constraints are maintained. For example, consider a counter $x = 2$ with the invariant $x \geq 0$. In this case, the counter can be decremented by up to two units without violating the invariant. This decrement capacity represents the available rights in the system, which are distributed among replicas. In a system with three replicas, the rights could be allocated as follows: replica 1 and replica 2 each hold one right, while replica 3 has none. If a decrement operation is submitted to replica 1, it can safely execute as it acquires an available right, preserving the invariant. Conversely, if a decrement operation is submitted to replica 3, it is aborted due to the lack of rights.

If the system supports escrow mechanisms, the necessary rights for an operation would be acquired in advance during the prepare phase, ensuring that the operation can be safely executed across replicas. In other words, escrow techniques eliminate conflicts by converting conflicting operations into non-conflicting ones. As a result, operations that previously required conflict resolution policies can now be executed safely within our framework, as conflicts no longer arise. A bounded counter [8] can, therefore, be implemented as a sequential counter, eliminating the need for an execution graph, much like in commutative CRDTs.

5.2.6 Aggregation Constraints

Aggregation constraints [6] enable the establishment of limits on the size of collections or enumerable attributes. Since the counted objects are in the database, aggregation constraints can be expressed as a disjunction of predicates. For instance, the example provided for the disjunctions – the number of songs in an album must be greater than one – is also an example of an aggregation constraint, as this property restricts the size of a collection of tracks.

Aggregation constraints comprise both lower and upper bounds. In lower bounds, the operations that remove an element from the collection are self-conflicting. Therefore, blocks are triggered whenever an element is removed from the collection. The number of blocked operations depends on the specified limit. For example, if the limit is 1, removing an element from the collection triggers a *block* on operations that remove a specific element. If the limit is 2, removing an element must block the removal of two different elements. Therefore, it triggers blocks on operations that remove one of the two blocked elements.

Conversely, in upper bounds, the conflicting operations are the operations that insert elements into the collection. However, identifying which operations to block is challenging, as it would require preemptively blocking any potential insertion, including block insertion of elements that may not yet exist in the system. Since there is no mechanism to block specific, non-existent elements, the *block* must be generic, restricting all insert operations. This approach can be overly restrictive and shares the same limitations as those faced with numerical invariants. In the future, we plan to explore alternative strategies to reduce the number of blocked operations without relying on escrow techniques.

5.2.7 Linear Resources

Linear resources [7] are applied to partitionable objects/resources to ensure no overlap. For example, a linear resource can be applied to ensure that a seat on a plane is not booked by more than one person. This type of invariant functions similarly to uniqueness, allowing only one operation from a set of concurrent operations to succeed. Consequently, any policy based on a total order is well-suited for maintaining this type of invariant.

5.3 Real-World Example

Ensuring application correctness requires preserving all defined invariants throughout execution and guaranteeing that all replicas converge after applying the same operations. So far, we have demonstrated how the No-Op framework supports the development of CRDTs and enables the creation of data structures that automatically maintain invariants. In this section, we bring these concepts together to illustrate how the No-Op framework can be used

	`registerUser(u2, ...)`	`unregisterUser(u2)`	`update(u2, ...)`	`placeBid(a2, u2, b2, ...)`	`closeAuction(a2)`
`registerUser(u1, ...)`	🟥 $u_1 = u_2$				
`unregisterUser(u1)`	✓	✓			
`update(u1, ...)`	✓	🟨 $u_1 = u_2$	🟨 $u_1 = u_2$		
`placeBid(a1, u1, b1, ...)`	✓	🟥 $u_1 = u_2$	✓	🟥 $a_1 = a_2 \wedge b_1 = b_2 \wedge u_1 \neq u_2$	
`closeAuction(a1)`	✓	✓	✓	🟨 $a_1 = a_2$	✓

Figure 5 Analysis of conflicts for the RUBiS system. Convergence conflicts are shown in yellow, whereas invariant conflicts are shown in red.

for developing a real-world distributed application. To this end, we develop an eBay-like auction system akin to the RUBiS system [14].

RUBiS supports operations such as registering and unregistering users, updating user information, opening and closing auctions, and bidding on open auctions. Additionally, it allows for direct sales, with operations to list items for sale and to purchase items. Based on the supported operations, we define five key invariants for RUBiS:

- System-assigned identifiers must be unique.
- User-chosen nicknames must be unique.
- Item stock cannot be negative.
- Each bid on an open auction must be associated with an existing user.
- There cannot be two bids with the same value for the same auction from two users.

Figure 5 illustrates all considered conflicting operations and their potential conflicts. Read operations are excluded, as they are safe-commutative and do not modify the database. The operations `openAuction`, `sellItem`, and `storeBuyNow` were analysed but are not shown in the figure since they do not conflict with any other operation. In the figure, yellow rectangles represent convergence conflicts, while red rectangles indicate invariant conflicts. Additionally, the relevant attributes responsible for each conflict are specified. For instance, conflicts between two user registration operations occur only when both use the same identifier.

Our RUBiS implementation in VeriFx assumes the system operates with escrow techniques and that its identifier generation mechanism ensures unique identifiers. These mechanisms guarantee that system-assigned identifiers are unique and that the item stock remains non-negative. As a result, there are no conflicting operations concerning these two invariants.

Recall that the No-Op approach relies on programmer-defined conflict resolution policies to handle conflicts between operations. The framework applies these policies at runtime to trigger blocks, which are used to detect and resolve conflicts. In the remainder of this section, we describe the resolution policies chosen for each identified conflict in the RUBiS system.

We opted for the last-writer-wins policy to resolve the uniqueness conflict between two `registerUser` operations. Similarly, we applied this same policy to address the convergence conflict caused by two `updateUser` operations targeting the same user. In both cases, the conflicts stem from self-conflicting operations, and only one operation from a set of concurrent operations must be executed. Consequently, any total-order-based policy is appropriate for resolving these conflicts.

For the same reasons, the uniqueness conflict caused by two users placing bids of the same value can also be resolved using a total-order-based policy. However, in this case, we have defined a policy that prioritises users based on the number of bids they have won. Specifically, the user who has won more auctions takes priority. If both users have won the same number of auctions, the policy resolves the tie using a total order over the operations.

Conflicts between `unregisterUser` and `placeBid` operations, which affect referential integrity, can be resolved using any of the policies described in Section 2. In our VeriFx implementation, we adopted the remove-wins policy. This decision aligns with the application's semantics, favouring the removal of the user and their associated bids over forcing the user to remain in the system, where they could win the auction.

In convergence conflicts between `closeAuction` and `placeBid`, we defined `closeAuction` as the winning operation. Prioritising `closeAuction` ensures the auction closes properly, avoiding inconsistencies such as reopening the auction due to a concurrent bid. If a `placeBid` operation is delivered late due to a failure, it could inadvertently reopen the auction long after it has been closed. By using this policy, we avoid these situations.

Finally, for conflicts involving `updateUser` and `unregisterUser` operations, we selected `updateUser` as the winning operation.

6 Discussion

This section examines the practical implications of the No-Op approach, including its applicability to different applications and the impact of poorly designed conflict resolution policies. Additionally, it explores the trade-off between optimisation and policy determinism.

6.1 When and How to Apply the No-Op Approach

The No-Op approach resolves conflicts by discarding operations, transforming them into no-side-effect operations. Each replica maintains its local view of the system and, using only local knowledge, determines which operations should be transformed into No-Ops to ensure convergence while preserving invariants.

The model may result in operations being discarded after a response has already been sent to the client. For example, consider the referential integrity conflict in Figure 2. Under an update-wins semantics, a client may initially observe an artist as removed but later find that the artist is still present. This occurs because the remove operation is transformed into a No-Op only after the concurrent update operation has been delivered and the conflict resolved. As some operations may be discarded after execution, this mechanism is unsuitable for applications that cannot tolerate such behaviour. Based on this, applications can be classified into two categories:

- **Applications that can handle reversals through compensation mechanisms.** These applications can use the framework alongside compensation mechanisms [20, 44]. When an operation is transformed into a No-Op, the system can trigger a compensation action to maintain consistency.
- **Applications that do not support operation reversals.** These applications require strong consistency and cannot operate under weak consistency models, making coordination the only viable solution.

Many conflict resolution policies prioritise certain operations over others. For example, in the last-writer-wins policy, after applying all updates, the *oldest* update does not affect the system's state, which is similar to transforming it into a No-Op. By explicitly transforming an operation into a No-Op, our approach has the advantage of clearly identifying the operations with no side-effect, making it easier to integrate a recovery or notification mechanism, such as the mentioned compensations.

In some cases, only a specific pair of operations may not support reversals, while all other conflicting pairs do. For example, in the RUBiS scenario, a programmer may believe that their application does not support reversals on `placeBid` and `closeAuction`, meaning neither operation can be transformed into No-Ops. In this case, they may opt for coordination to handle these operations [30]. However, using coordination to prevent the concurrent execution of operations that cannot be transformed into a No-Op does not preclude applying the No-Op approach to resolve other conflicts within the application. In such scenarios, the No-Op framework can be augmented with strong consistency mechanisms without affecting the rest of the system. For example, coordination could be implemented using locks, ensuring their correctness while allowing the remaining operations in the application to continue following the No-Op framework. This hybrid approach maintains efficiency while addressing specific consistency requirements where necessary.

6.2 Challenges with Poorly Designed Conflict Resolution Policies

Although the No-Op approach is coordination-free, it may sometimes mark operations as No-Ops unnecessarily. This issue can arise for different reasons. Sometimes, the model is overly restrictive for specific invariants, such as numerical invariants (cf. Section 5.2.5). In others, it may stem from the design of the conflict resolution policies implemented by the programmer. This section examines how poorly designed policies can increase the number of No-Ops and explores strategies to identify policy definition improvements.

To illustrate how poorly designed conflict resolution policies can lead to unnecessary No-Ops, let us revisit the example in Figure 4. Consider that the programmer specifies that `addAlbum` wins in conflicts with `rmvArtist`. In turn, `rmvArtist` wins in conflicts with `updArtist`. When `addAlbum`, `rmvArtist`, and `updArtist` are executed concurrently, the algorithm marks both `rmvArtist` and `updArtist` as No-Ops. As a result, fewer concurrent operations can be executed, impacting performance. However, defining `rmvArtist` as the losing operation in both cases solves the conflicts and ensures that only one of the three operations becomes a No-Op. Redefining the conflict resolution policies can thus improve performance without compromising the application's correctness.

Poorly designed conflict resolution policies could, however, lead to more severe issues hindering the overall system progress. For example, operation op_1 marks operation op_2 as a No-Op, operation op_2 marks operation op_3 as a No-Op, and operation op_3, in turn, marks operation op_1 as a No-Op. While this circular dependency does not compromise correctness, it causes all conflicting operations to be marked as No-Ops, hampering progress.

A policy graph provides a systematic way to identify and address those issues. In a policy graph, vertices represent conflicting operations, while edges show how conflicts are resolved. Long path lengths in the graph highlight scenarios where an excessive number of operations are unnecessarily marked as No-Ops. Meanwhile, cycles in the graph indicate circular dependencies. Therefore, programmers should define conflict resolution policies that ensure a cycle-free policy graph while minimising path lengths to reduce inefficiencies. In future work, we plan to explore an automated analysis tool that generates and analyses the policy graph based on programmer-defined conflict resolution policies. This tool will provide feedback, guiding developers to the policies that can be refined to enhance performance.

6.3 Optimisation for Deterministic Resolution

The correctness of our model, described in Algorithms 1 and 2, does not depend on the deterministic nature of conflict resolution policies. The policies only determine how blocks are triggered. If a policy is non-deterministic, two replicas in the same state executing the same operation may generate different blocks. However, since blocks are generated locally during the prepare phase, they are triggered only once per operation, ensuring their assignment remains deterministic. For example, consider a policy that resolves conflicts by randomly selecting an operation to be transformed into a No-Op. In this case, the blocked operation is chosen randomly during the prepare phase. However, since the prepare phase is executed only by the local replica, the same *block* assignment is propagated to all remote replicas. During the effect phase, each delivered operation is compared against all its concurrent operation calls, including those already marked as No-Ops. As a result, all replicas process blocks uniformly, ensuring No-Ops converge. Therefore, by design, the No-Op approach ensures that replicas processing the same operations produce identical No-Op sets. This is achieved through the block abstraction, which guarantees that conflicts are resolved deterministically regardless of policy non-determinism.

When implementing the model, the creation of blocks could be optimised. Rather than creating the *block* when an operation is submitted (in the prepare phase), it is possible to simply perform the effect of blocks – transform concurrent conflicting operations into No-Ops according to the policy – upon detecting a conflict when an operation is delivered (in the effect phase). This optimisation reduces the amount of metadata that must be broadcast over the network and minimises the data stored in the graph, as blocks are no longer appended to operations.

Note that this optimisation is only applicable to deterministic conflict resolution policies. Since the policy is applied directly during the effect phase, a non-deterministic policy could lead to divergence, as different replicas may resolve conflicts inconsistently. Consider again the policy that resolves conflicts by randomly transforming an operation into a No-Op. In this case, two replicas processing the same conflict could make different decisions, resulting in divergent execution graphs. Another example is state-dependent policies, which issue blocks based on the application's current state.

Optimising the model comes at the cost of imposing stronger assumptions on the conflict resolution policies. While this optimisation reduces metadata overhead and storage costs by eliminating blocks, it requires deterministic conflict resolution to ensure convergence. In contrast, the original model allows for greater flexibility in defining policies, including non-deterministic ones, at the cost of maintaining extra metadata. If the optimised version is adopted, the verification step should verify whether the resolution policies are deterministic.

7 Related Work

Ensuring correctness in applications built on weakly consistent storage is an active research area, with two core challenges: achieving state convergence across replicas and preserving application invariants. Traditional replicated data types (RDTs) [40, 13, 26, 9] provide a principled, coordination-free approach to ensure replicas converge to the same state after executing the same operations. However, these data types are typically hand-crafted for specific use cases. In contrast, our framework offers a unified approach that can be applied consistently to implement different policies. Pure operation-based CRDTs [9] also maintain a graph of operations, encoding conflict resolution in the read operations. Our approach differs fundamentally by resolving conflicts during the write operations. While these approaches for building RDTs guarantee convergence, they do not address invariant maintenance.

The remainder of this section reviews methods for enforcing invariants, classifying them into coordination-based, reservation-based, and coordination-free approaches, and examines their characteristics and trade-offs.

Coordination-based approaches. Coordination-based approaches rely on synchronisation to prevent operations from violating invariants. As described in [4], invariant confluence provides a necessary and sufficient condition for determining whether invariants can be maintained under concurrent operations. This principle identifies the operations requiring coordination: non-$\mathcal{I}$-confluent operations. Several techniques aim to minimise the level of coordination required. Lucy [46] achieves this by segmenting the state space and restricting transactions within each segment, ensuring each segment becomes $\mathcal{I}$-confluent. Similarly, Blazes [3] identifies locations needing coordination and automatically synthesises application-specific coordination code to reduce synchronisation overhead. Other approaches focus on generating replicated objects that inherently maintain invariants. For instance, Hamsaz [24] and Hampa [32] automatically synthesise replicated objects by coordinating unsafe operations. CISE [21] provides an automated proof rule to verify whether invariants are preserved on replicated databases under a given replication protocol. To ensure invariant safety, CISE coordinates conflicting operations. Quelea [42] determines the weakest consistency model that upholds invariants by statically analysing contracts to assign the most efficient and sound consistency level to each operation. Similarly, Q9 [25] identifies the weakest consistency model needed to prevent invariant violations. Indigo [7] gives programmers more control by allowing them to either coordinate unsafe operations or automatically repair invariants post-facto. It also introduces a multi-level mask reservation mechanism, using fine-grained locking to prevent specific conflicting operations. ECROs [18] reduce synchronisation by reordering conflicting operations, but it still relies on coordination when reordering is impossible. LoRe [22] automatically verifies developer-supplied safety properties for local-first applications, resolving invariant violations due to concurrency by adding coordination logic as needed. Coordination-based approaches optimise synchronisation but remain conservative, since they still rely on coordination to uphold invariants. In contrast, we propose a coordination-free alternative with the No-Op approach, which eliminates the need for synchronisation entirely, offering a novel and efficient solution for maintaining consistency in replicated systems.

Reservation-based approaches. Reservation-based approaches [36, 8, 7, 33, 38, 41] reduce coordination costs by moving synchronisation outside the execution path of operations. By reserving the rights to perform specific operations in advance, replicas can execute operations safely without requiring immediate coordination. However, these mechanisms assume the necessary rights are available at the local replica. Otherwise, coordination is needed to acquire rights from other replicas or operations are aborted. Escrow transactions [36] apply this concept to enforce numerical invariants. The difference between the current value of a numerical variable and its bound determines the number of rights distributed among replicas. The bounded counter CRDT [8] builds on escrow transactions by implementing this mechanism asynchronously to uphold numerical invariants. Partition lock reservation [7] extends these ideas to enforce linear resource constraints. It allows replicas to reserve rights over non-overlapping value intervals, enabling operations within these intervals to execute locally without coordination. However, operations that require values outside the reserved intervals must synchronise with other replicas. Although these mechanisms maximise concurrency for operations that might otherwise violate invariants, they still rely on locks or coordination when local rights are exhausted. In our work, we incorporate escrow techniques specifically for numerical invariants, as the No-Op approach can be overly conservative in such cases. By pre-allocating execution rights, replicas can safely execute concurrent operations without unnecessary No-Ops, ensuring safety. This integration reduces the constraints of the No-Op model for numerical invariants while preserving its coordination-free nature for other invariants.

Coordination-free approaches. Coordination-free approaches maintain invariants by transforming non-$\mathcal{I}$-confluent operations into $\mathcal{I}$-confluent operations. Existing approaches, as well as our approach, do so by introducing conflict resolution mechanisms. IPA [6] resolves conflicts by extending conflicting operations with updates that ensure invariant preservation. It leverages static analysis to identify conflicts and suggests to programmers conflict resolution policies, enabling them to choose the policy that best suits their application. Compared to IPA, our No-Op approach introduces two key differences. First, while IPA resolves conflicts by adding updates, our approach discards conflicting operations. Second, IPA relies on CRDTs for convergence, whereas our approach natively supports convergence and invariant preservation without additional structures. Antidote SQL [35] enforces referential integrity through resolution policies. However, these policies are limited to referential integrity and cannot be extended to other invariants. In contrast, as shown in Section 5.2, our approach provides broader applicability, enabling the preservation of a wider range of invariants.

8 Conclusion

This paper introduced a coordination-free consistency framework that unifies data convergence and invariant preservation through the novel No-Op approach. Conflicts are resolved by designating one operation in each conflict as a no-side-effect operation. By transforming non-preferred operations into no-side-effect operations, the No-Op approach ensures consistency and safety across all replicas even under concurrent updates, as established by formal proofs. To the best of our knowledge, this is the first unified approach that ensures strong convergence and invariant preservation while eliminating the need for synchronisation.

Our approach allows programmers to define custom conflict resolution policies tailored to the specific semantics of their applications. Additionally, it eliminates the need to design and implement new data structures to support these policies, thereby simplifying the development of highly available distributed systems. We implemented the framework in the VeriFx programming language to demonstrate its practicality, leveraging its automated proof engine to validate each data-type implementation. This implementation confirmed the manual proofs of our approach, bridging the gap between theoretical guarantees of consistency and real-world applicability. Additionally, we showcased its ability to encode popular CRDTs and enforce a wide range of invariants, including referential integrity. Finally, we demonstrate our approach for developing real-world applications, such as the geo-distributed RUBiS system.

While the No-Op approach is coordination-free, it can sometimes unnecessarily transform operations into No-Ops. To address this challenge, we propose to integrate escrow techniques to minimise superfluous No-Ops and leverage policy graphs to identify inefficiencies.

References

1 Daniel Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *Computer*, 45(2):37–42, February 2012. doi:10.1109/MC.2012.33.

2 Mustaque Ahamad, Gil Neiger, James E. Burns, Prince Kohli, and Phillip W. Hutto. Causal memory: Definitions, implementation, and programming. *Distrib. Comput.*, 9(1):37–49, March 1995. doi:10.1007/BF01784241.

3 Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and David Maier. Blazes: Coordination analysis and placement for distributed programs. *ACM Trans. Database Syst.*, 42(4), October 2017. doi:10.1145/3110214.

4 Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. Coordination avoidance in database systems. *Proc. VLDB Endow.*, 8(3):185–196, November 2014. doi:10.14778/2735508.2735509.

5 Jason Baker, Chris Bond, James C. Corbett, J. J. Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing scalable, highly available storage for interactive services. In *Proceedings of the Fifth Biennial Conference on Innovative Data Systems Research, CIDR 2011, Asilomar, CA, USA, January*

9-12, 2011, Online Proceedings, pages 223–234. www.cidrdb.org, 2011. URL: `http://cidrdb.org/cidr2011/Papers/CIDR11_Paper32.pdf`.

6 Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, and Nuno Preguiça. Ipa: invariant-preserving applications for weakly consistent replicated databases. *Proc. VLDB Endow.*, 12(4):404–418, December 2018. `doi:10.14778/3297753.3297760`.

7 Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno Preguiça, Mahsa Najafzadeh, and Marc Shapiro. Putting consistency back into eventual consistency. In *Proceedings of the Tenth European Conference on Computer Systems*, EuroSys '15, New York, NY, USA, 2015. Association for Computing Machinery. `doi:10.1145/2741948.2741972`.

8 Valter Balegas, Diogo Serra, Sérgio Duarte, Carla Ferreira, Marc Shapiro, Rodrigo Rodrigues, and Nuno M. Preguiça. Extending eventually consistent cloud databases for enforcing numeric invariants. In *34th IEEE Symposium on Reliable Distributed Systems, SRDS 2015, Montreal, QC, Canada, September 28 - October 1, 2015*, pages 31–36. IEEE Computer Society, 2015. `doi:10.1109/SRDS.2015.32`.

9 Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. Pure operation-based replicated data types. *CoRR*, abs/1710.04469, 2017. `arXiv:1710.04469`.

10 Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. A critique of ANSI SQL isolation levels. In Michael J. Carey and Donovan A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, USA, May 22-25, 1995*, pages 1–10. ACM Press, 1995. `doi:10.1145/223784.223785`.

11 Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

12 Dina Borrego, Nuno Preguiça, Elisa Gonzalez Boix, and Carla Ferreira. Ensuring convergence and invariants without coordination (appendix), 2025. URL: `https://soft.vub.ac.be/Publications/2025/vub-tr-soft-25-02.pdf`.

13 Sebastian Burckhardt, Manuel Fähndrich, Daan Leijen, and Benjamin P. Wood. Cloud types for eventual consistency. In James Noble, editor, *ECOOP 2012 - Object-Oriented Programming - 26th European Conference, Beijing, China, June 11-16, 2012. Proceedings*, volume 7313 of *Lecture Notes in Computer Science*, pages 283–307. Springer, 2012. `doi:10.1007/978-3-642-31057-7_14`.

14 Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. Performance and scalability of EJB applications. In Mamdouh Ibrahim and Satoshi Matsuoka, editors, *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2002, Seattle, Washington, USA, November 4-8, 2002*, pages 246–261. ACM, 2002. `doi:10.1145/582419.582443`.

15 Brian F. Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.*, 1(2):1277–1288, August 2008. `doi:10.14778/1454159.1454167`.

16 James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally distributed database. *ACM Trans. Comput. Syst.*, 31(3), August 2013. `doi:10.1145/2491245`.

17 Kevin De Porre, Carla Ferreira, and Elisa Gonzalez Boix. VeriFx: Correct Replicated Data Types for the Masses. In Karim Ali and Guido Salvaneschi, editors, *37th European Conference on Object-Oriented Programming (ECOOP 2023)*, volume 263 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:45, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.ECOOP.2023.9`.

18 Kevin De Porre, Carla Ferreira, Nuno Preguiça, and Elisa Gonzalez Boix. ECROs: Building global scale systems from sequential code. *Proc. ACM Program. Lang.*, 5(OOPSLA), October 2021. doi:10.1145/3485484.

19 Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. Association for Computing Machinery. doi:10.1145/1294261.1294281.

20 Hector Garcia-Molina and Kenneth Salem. Sagas. *SIGMOD Rec.*, 16(3):249–259, December 1987. doi:10.1145/38714.38742.

21 Alexey Gotsman, Hongseok Yang, Carla Ferreira, Mahsa Najafzadeh, and Marc Shapiro. 'Cause I'm strong enough: Reasoning about consistency choices in distributed systems. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 371–384, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2837614.2837625.

22 Julian Haas, Ragnar Mogk, Elena Yanakieva, Annette Bieniusa, and Mira Mezini. Lore: A programming model for verifiably safe local-first software. *ACM Trans. Program. Lang. Syst.*, 46(1), January 2024. doi:10.1145/3633769.

23 Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, July 1990. doi:10.1145/78969.78972.

24 Farzin Houshmand and Mohsen Lesani. Hamsaz: replication coordination analysis and synthesis. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi:10.1145/3290387.

25 Gowtham Kaki, Kapil Earanky, KC Sivaramakrishnan, and Suresh Jagannathan. Safe replication through bounded concurrency verification. *Proc. ACM Program. Lang.*, 2(OOPSLA), October 2018. doi:10.1145/3276534.

26 Gowtham Kaki, Swarn Priya, KC Sivaramakrishnan, and Suresh Jagannathan. Mergeable replicated data types. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019. doi:10.1145/3360580.

27 Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. Local-first software: you own your data, in spite of the cloud. In *Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2019, pages 154–178, New York, NY, USA, 2019. Association for Computing Machinery. doi:10.1145/3359591.3359737.

28 Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010. doi:10.1145/1773912.1773922.

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

30 Cheng Li, Daniel Porto, Allen Clement, Johannes Gehrke, Nuno M. Preguiça, and Rodrigo Rodrigues. Making geo-replicated systems fast as possible, consistent when necessary. In Chandu Thekkath and Amin Vahdat, editors, *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, pages 265–278. USENIX Association, 2012. URL: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/li.

31 Cheng Li, Nuno M. Preguiça, and Rodrigo Rodrigues. Fine-grained consistency for geo-replicated systems. In Haryadi S. Gunawi and Benjamin C. Reed, editors, *Proceedings of the 2018 USENIX Annual Technical Conference, USENIX ATC 2018, Boston, MA, USA, July 11-13, 2018*, pages 359–372. USENIX Association, 2018. URL: https://www.usenix.org/conference/atc18/presentation/li-cheng.

32 Xiao Li, Farzin Houshmand, and Mohsen Lesani. Hampa: Solver-aided recency-aware replication. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part I*, pages 324–349, Berlin, Heidelberg, 2020. Springer-Verlag. doi:10.1007/978-3-030-53288-8_16.

33 Jed Liu, Tom Magrino, Owen Arden, Michael D. George, and Andrew C. Myers. Warranties for faster strong consistency. In Ratul Mahajan and Ion Stoica, editors, *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2014, Seattle, WA, USA, April 2-4, 2014*, pages 503–517. USENIX Association, 2014. URL: `https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/liu_jed`.

34 Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Don't settle for eventual: Scalable causal consistency for wide-area storage with cops. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 401–416, New York, NY, USA, 2011. Association for Computing Machinery. `doi:10.1145/2043556.2043593`.

35 Pedro Lopes, João Sousa, Valter Balegas, Carla Ferreira, Sérgio Duarte, Annette Bieniusa, Rodrigo Rodrigues, and Nuno M. Preguiça. Antidote SQL: relaxed when possible, strict when necessary, 2019. `arXiv:1902.03576`.

36 Patrick E. O'Neil. The escrow transactional method. *ACM Trans. Database Syst.*, 11(4):405–430, December 1986. `doi:10.1145/7239.7265`.

37 Nuno M. Preguiça. Conflict-free replicated data types: An overview. *CoRR*, abs/1806.10254, 2018. `arXiv:1806.10254`.

38 Nuno M. Preguiça, José Legatheaux Martins, Miguel Cunha, and Henrique João L. Domingos. Reservations for conflict avoidance in a mobile database system. In Daniel P. Siewiorek, Mary Baker, and Robert T. Morris, editors, *Proceedings of the First International Conference on Mobile Systems, Applications, and Services, MobiSys 2003, San Francisco, CA, USA, May 5-8, 2003*, pages 43–56. USENIX, 2003. `doi:10.1145/1066116.1189038`.

39 Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *Stabilization, Safety, and Security of Distributed Systems - 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings*, volume 6976 of *Lecture Notes in Computer Science*, pages 386–400. Springer, 2011. `doi:10.1007/978-3-642-24550-3_29`.

40 Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of convergent and commutative replicated data types. Technical report, INRIA, 2011.

41 Liuba Shrira, Hong Tian, and Douglas B. Terry. Exo-leasing: Escrow synchronization for mobile clients of commodity storage servers. In Valérie Issarny and Richard E. Schantz, editors, *Middleware 2008, ACM/IFIP/USENIX 9th International Middleware Conference, Leuven, Belgium, December 1-5, 2008, Proceedings*, volume 5346 of *Lecture Notes in Computer Science*, pages 42–61. Springer, 2008. `doi:10.1007/978-3-540-89856-6_3`.

42 KC Sivaramakrishnan, Gowtham Kaki, and Suresh Jagannathan. Declarative programming over eventually consistent data stores. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 413–424, New York, NY, USA, 2015. Association for Computing Machinery. `doi:10.1145/2737924.2737981`.

43 Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. Transactional storage for geo-replicated systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 385–400, New York, NY, USA, 2011. Association for Computing Machinery. `doi:10.1145/2043556.2043592`.

44 D. B. Terry, M. M. Theimer, Karin Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. *SIGOPS Oper. Syst. Rev.*, 29(5):172–182, December 1995. `doi:10.1145/224057.224070`.

45 Werner Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, January 2009. `doi:10.1145/1435417.1435432`.

46 Michael Whittaker and Joseph M. Hellerstein. Interactive checks for coordination avoidance. *Proc. VLDB Endow.*, 12(1):14–27, September 2018. `doi:10.14778/3275536.3275538`.