Echo: A CRDT-based Framework for Highly-Available Applications with Domain-Specific Properties

Alexandre Piveteau
École Polytechnique Fédérale de Lausanne
Switzerland
alexandre.piveteau@epfl.ch

Bertil Chapuis
HES-SO, HEIG-VD, IICT
Switzerland
bertil.chapuis@heig-vd.ch

Benoît Garbinato
Université de Lausanne
Switzerland
benoit.garbinato@unil.ch

***Abstract*—Conflict-free Replicated Data Types (CRDTs) are a well-known mechanism for maintaining consistency and reliability in high-availability applications, especially for optimistic replication. However, current CRDT implementations often fall short in preserving domain-specific properties critical for practical applications. For instance, in distributed file systems with a directory tree, it is crucial to prevent cyclic paths when moving documents concurrently. This demonstration paper presents Echo,[1] a framework designed to convert any stateful, single-node application with unique domain-specific requirements into a replicated application. Echo, implemented as a causally and totally ordered replicated event log, introduces a generic CRDT abstraction to developers. It guarantees strong eventual convergence and preserves domain-specific properties related to the semantics of the application. Using a collaborative text editor as a case study, we empirically establish the suitability of our framework for enabling the replication of applications with rich semantics. Finally, we assess the efficiency of Echo through multiple benchmarks.**

I. Introduction

Originally termed Brewer's Conjecture and later known as the CAP Theorem, the impossibility of simultaneously achieving *consistency*, *availability*, and *partition tolerance* in distributed systems is a well-known result [6]. To address this impossibility, the CALM (Consistency As Logical Monotonicity) theorem was introduced. It demonstrates the feasibility of designing a distributed system that is *eventually convergent*, *available* and *partition tolerant* by composing monotonic data structures or algorithms [7]. Conflict-Free Replicated Data Types (CRDTs) are an application of the CALM theorem, since they are monotonic data structures by design [18]. CRDTs come in two forms shown to be equivalent [18]: Commutative Replicated Data Types (CmRDTs) are *operation-based* and rely on causal broadcasting of operations; Convergent Replicated Data Types (CvRDTs) are *state-based* and rely on gossiping to propagate their state to all replicas.

CRDTs are typically used to implement optimistic replication in highly-available distributed applications. However, they often lack the ability to enforce domain-specific properties. For example, in distributed file systems, there is a need to maintain a tree structure without cyclic links and ensure unique file names in each directory. Collaborative editing applications require atomic operations for cursor movements in texts. Several libraries, discussed in Section VI, offer ready-to-use CRDTs for building these applications. However, few extend beyond core eventual convergence semantics to enforce domain-specific properties.

The main contribution of this demonstration paper consists in a generic CRDT-based framework named *Echo*, which enables developers to define and enforce domain-specific properties in distributed applications managing replicated states. The novelty of our approach does not lie in the underlying CRDT mechanisms, but in the reduced development effort required to introduce replication in any application that follows the application model presented in Section III, without requiring an extensive rewrite. This is achieved by relying on a generic log-based CRDT. If the original application does not follow this model, it might be necessary first to refactor it to conform to the application model. This is no different than most software frameworks, which usually put constraints on the architecture of applications using them in exchange for the simplification they offer.

Practically, our generic CRDT relies on a causally and totally ordered replicated operation log. The remainder of this paper is organized as follows. Section II defines our system model, provides background definitions, and illustrates how Echo can be used to add replication to existing applications. Section III delves into Echo's internals, demonstrating its capability to convert local application models into replicated ones. Section IV demonstrates how we used Echo to develop a fully functional collaborative text editor. Performance evaluation and analysis of Echo are presented in Section V. Related work is reviewed in Section VI, and the paper concludes with a discussion on future work in Section VII.

II. The Echo framework

We consider a peer-to-peer distributed system with equipotent and asynchronous processes. Processes follow a crash-recovery fault model [4], i.e., they do not exhibit Byzantine behaviors. New processes may join at any time, i.e., their

[1] [redacted name for blind review]

number may vary over time. When a new process joins, a unique identifier is assigned to it. Processes execute application models by exchanging operations through a fair-lossy broadcast. Intuitively, an application model is responsible for enforcing domain-specific properties. Hereafter, we formally define the concept of *application model*.

Definition 1 (Application model)**.** *Let M be the set of all the possible application **states**, E be the set of application **operations**, and V be the set of application **values**. An **application model** is a triplet, $(init_{app}, query_{app}, update_{app})$, where:*

- $init_{app} \in M$ *is the **initial state**,*
- $query_{app} : M \to V$ *is the **query function**, and*
- $update_{app} : M \times E \to M$ *is the **update function**.*

The query function computes the current value of the model, while the update function installs its new state. Together, these functions ensure the domain-specific properties of the application model. The concept of application model is rooted in the unidirectional data flow paradigm, which distinguishes operations from state. In this model, operations are explicit and yield updated immutable states.

Example 1 (Shopping cart)**.** *To illustrate the application model concept, consider a basic shopping cart system where users can add or remove items to purchase. The shopping cart system must enforce domain-specific properties, particularly accommodating concurrent additions or deletions of items. Managing these cases is not trivial: conflicts in additions and removals can lead to previously deleted items resurfacing [5]. In a replicated environment, these properties must be ensured across all replicas in a consistent manner. Let $A = \{add, remove\}$ be the set of available actions on an item, and let I be the set of items that a user may buy. Consequently, $A \times I$ represents the set of possible application operations on the shopping cart. We can define the following application model:*

- $init_{app} = \emptyset$ *is the initial model, with no items present in the cart,*
- $query_{app}(m) = m$ *is the query function, and returns the current set of items in the cart, and*
- $update_{app}(m, (a, i))$ *the update function, which adds or removes items from the cart:*

$$update_{app}(m, (a, i)) = \begin{cases} m \cup \{i\} & \text{if } a = add \\ m \setminus \{i\} & \text{if } a = remove \end{cases}$$

Given an original application model, Echo generates a semantically equivalent *replicated* model. This model maintains the original domain-specific properties expressed by the query and update functions, as illustrated in Figure 1. Echo achieves this by propagating created operations across all replicas. The replicated application provides the following five guarantees when computing the value of the model:

- **No creation.** An operation will only be used to compute the value if it has previously been generated by a process.

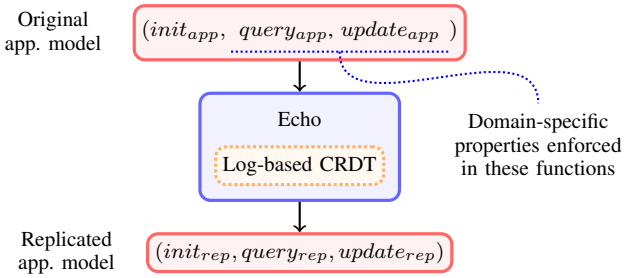


Fig. 1: Echo transforms an application model into a replicated application model.

- **No duplication.** Each operation will not be applied more than once when computing the value.
- **Causal order.** If the creation or reception of an operation o_1 *happens-before* [13] the creation of operation o_2 on any process, then no participant will apply operation o_2 before operation o_1 when computing the value.
- **Total order.** If two processes p and q both apply the operations o_1 and o_2 to compute the value and p applies o_1 before applying o_2, then q will apply o_1 before applying o_2.
- **Eventual agreement.** If a correct participant applies an operation in the computation of the value or creates an operation, then all correct participants will eventually apply this operation in the computation of the value.

These guarantees ensure the *liveness* and the *safety* of the replicated application model. That is, they ensure that user operations eventually propagate to all correct replicas (liveness) and that no spurious operations are applied (safety).

III. Inside Echo

Echo automatically generates a replicated application model from an existing non-replicated one, ensuring strong eventual convergence. It does so by establishing a causal and total order for all application operations and eventually enforcing it across all peers. Subsequently, events are processed following this total order, resulting in a state that preserves the domain-specific properties and the guarantees listed in Section II. This order is obtained by constructing a generic CRDT consisting of a log of operations.

At a high level, this replicated data type is a set pairing each operation with a logical timestamp, hence establishing a total and causal order of the operations in the log. Consequently, two processes with identical CRDT states, i.e., having the same set of operations, will traverse these operations in the same total and causal order and compute the same value.

A. State-based CvRDTs

A state-based Convergent Replicated Data Type (CvRDT) is an object replicated across multiple processes. It includes an *initial state*, a *query function*, an *update function*, and a *merge function*. The query and update functions enable

the process to read and write the state of the local replica. The CvRDT then employs a broadcast protocol to ensure that the updated state is eventually gossiped to every correct process in a fault-tolerant manner. Processes merge remote updates to their local replicas using the merge function, which must be **commutative**, **associative**, and **idempotent**. Repeated applications of the *merge* function ensures the strong eventual convergence of the object's state.

We now define CvRDTs, using the formalism from [18]. Note that the *merge* function forms a semi-lattice over the set S and acts as its upper join.

Definition 2 (State-based object)**.** *Let S be the set of **states**, U be the set of **update operations**, and V be the set of **values**. A **state-based object** is a 4-tuple, $(init_{crdt}, query_{crdt}, update_{crdt}, merge_{crdt})$, where:*

- $init_{crdt} \in S$ *is the **initial state**,*
- $query_{crdt} : S \to V$ *is the **query function**,*
- $update_{crdt} : U \times S \to S$ *is the **update function**, and*
- $merge_{crdt} : S \times S \to S$ *is the **merge function**.*

B. Generic log-based CvRDT

Echo leverages a generic CvRDT that contains the log of operations, each one associated with a unique Lamport timestamp [13]. When a new operation is inserted, its associated timestamp is computed to be strictly greater than all the timestamps present in the CvRDT, and associated with the local site identifier to serve as a disambiguator in case two timestamps have the same numeric value. The *merge* operation is a set union of the operation logs, which guarantees the strong eventual convergence of the data type, and makes the CvRDT a specialized GSet [17]. Thus, processes sharing identical CvRDT states, i.e., containing the same operations, traverse these operations in the same causal and total order.

Formally, the set of CvRDT states is defined as follows. Let E be the set of application operations within the generic CvRDT, and $\mathbb{I}$, the set of process identifiers. Thus, the state of the CvRDT is a set of elements from $E \times (\mathbb{N} \times \mathbb{I})$, with each Lamport timestamp being a natural number. Its initial state $init_{crdt}$ is the empty set $\emptyset$. Its query function $query_{crdt}$ traverses all the operations of the set by increasing timestamp and returns their sequence. Its update function $update_{crdt}$ assigns a Lamport timestamp strictly superior to all the timestamps present in the set at the time of update. Finally, its merge function $merge_{crdt}$ is the set union $\cup$. The $merge_{crdt}$ function is *commutative*, *idempotent*, and *associative*, so this replicated object is indeed a CvRDT. Note that the $merge_{crdt}$ function will be identical for all the instances of the CvRDT, and does not have to be defined by the developer.

C. Replicated application model using CvRDTs

Utilizing the log-based generic CvRDT, we can build a replicated application model derived from the user-provided model. Intuitively, we use the log-based generic CvRDT to store all the operations issued to the application model and replay them in causal and total order when the model is queried. The inherent properties of the CvRDT ensure that the application model, once replicated, will guarantee strong eventual convergence.

Consider a user provided application model denoted as $(init_{app}, query_{app}, update_{app})$. We can now instantiate a log-based CvRDT $(init_{crdt}, query_{crdt}, update_{crdt}, merge_{crdt})$ to implement the replicated application model $(init_{rep}, query_{rep}, update_{rep})$ on a site with identifier ID. The initial state of the replicated application model $init_{rep}$ has value $init_{crdt}$, which is the empty set. Algorithms 1 and 2 detail a possible implementation of the replicated model.

Algorithm 1 $query_{rep}(m)$: Query function of the replicated application model

```
acc ← init_app
for (operation, ts) ← query_crdt(m) do      ▷ Sorted by increasing
    acc ← update_app(acc, operation)        ▷ Lamport timestamp ts
end for
return query_app(acc)
```

Algorithm 1 shows how $query_{rep}$ computes the application model's current value. First, it creates an accumulator, acc, set to the model's initial value. Then, the log-based CvRDT is queried to retrieve all timestamped operations. These operations are sorted by increasing Lamport timestamp and are being applied consecutively to the accumulator. Finally, $query_{app}$ performs a query on the accumulator, which makes sure that the values returned by $query_{rep}$ are of the same type as the values returned by the original $query_{app}$ function. This strategy maintains domain-specific properties and constraints by using the original $update_{app}$ and $query_{app}$ functions. Thus, each operation in the CvRDT log is applied exactly once.

By introducing a total and causal ordering with the Lamport timestamps, Echo always favors operations with the smallest timestamp. The drawback of this approach is the loss of direct control by the application developer over the ordering of operations with no causal relationship. The rationale behind this approach is that, in collaborative applications where conflicts commonly arise, resolving these issues often requires user intervention. One strategy consists in blocking the progress of the application until users resolve the conflict. Another strategy consists in letting the application make a somewhat arbitrary decision to resolve the conflict, which might require replaying some of the users' actions. This second strategy, adopted by Echo, has two advantages. Firstly, it prevents the application from being blocked. Secondly, it relieves application developers from the burden of writing specific conflict-management code. Developers are rarely familiar with the intricacies of concurrency conflicts and often resort to asking users to resolve the conflict themselves. This approach can be seen as an optimistic replication strategy.

Algorithm 2 $update_{rep}(m, e)$: Update function of the replicated application model

```
ts ← (n, ID)        ▷ n is chosen s.t. ∀(e', (n', s')) ∈ m : n > n'
return m ∪ {(e, ts)}
```

```
enum class Op { Inc, Dec }
class Application(initialState = 0) {
  var model = initialState
  private fun update(value: Int, op: Op) =
    when (op) {
      Inc -> value + 1
      Dec -> value - 1
    }
  fun increment() {
    model = update(model, Op.Inc)
  }
  fun decrement() {
    model = update(model, Op.Dec)
  }
  fun query(): Int = model
}
```

(a) Original application model of a simple counter (`update` and `query` functions)

```
enum class Op { Inc, Dec }
class Application(initialState = 0) {
  val model = mutableSite(initialState,::update)
  private fun update(value: Int, op: Op) =
    when (op) {
      Inc -> value + 1
      Dec -> value - 1
    }
  suspend fun increment() {
    model.event { yield(Op.Inc) }
  }
  suspend fun decrement() {
    model.event { yield(Op.Dec) }
  }
  fun query(): Int = model.value
}
```

(b) Corresponding replicated application model with Echo (code changes in blue)

Fig. 2: Building a simple replicated counter in Kotlin with Echo.

Algorithm 2 describes updating the state of a generic log-based CvRDT with a new operation, removing the need for an explicit *merge* function. First, it identifies an integer n that exceeds all existing operation timestamp, to form the Lamport timestamp associated with the newly inserted operation. Then, the set of operations in the underlying log-based CvRDT is updated by adding the new timestamped operation.

The replicated application model construction adheres to the five properties guaranteed by the Echo framework (Sec. II). Firstly, ***no creation*** is enforced as operations must be written to the log via the application model's update function in order to be applied in the query function. Secondly, ***no duplication*** is met as each operation is applied only once in the query function. Thirdly, ***causal order*** is maintained as an operation e_1 preceding e_2 results in e_2 receiving a higher timestamp. Fourthly, the ***total order*** is consistent across participants with identical pair of operations due to the increasing timestamps. Finally, the CvRDT's strong eventual convergence ensures the ***eventual agreement*** on the state, with all replicas eventually sharing the same set of operations annd thus computing the same application model value.

D. Practical implementation considerations

While Section III-C presents Echo's conceptual implemenetation, practical applications necessitate further improvements and optimizations discussed hereafter.

- **Eager aggregation.** Because the timestamps define a total order, a practical implementation of the query function can be efficiently implemented. Each inserted operation could be paired with the state of the accumulator before applying the operation. Upon the insertion of a new operation, the algorithm would start by skipping to the accumulator value of the preceeding operation in the total order, and only recompute a subset of the associated aggregations. This way, if a single operation was to be inserted with a timestamp greater than all previously received operations, the query result could be incrementally computed in constant time.
- **Causality tracking using Hybrid Logical Clocks.** By using a generic operation log, refinements applied to the underlying operation log will benefit all the application models implemented on top of Echo. For instance, Echo uses Hybrid Logical Clocks [11], an extension of Lamport timestamps that respect the *happens-before* relationship but can determine a more precise total order of concurrent operations using physical time.
- **Domain-specific optimizations.** The construction presented in Section III-C makes no assumptions over the characteristics or properties of the application operations. However, some optimizations may be possible. For example, idempotent operations could be applied unconditionally by dropping the *no duplication* property. Also, commutative operations could be applied in any order (while preserving causal dependencies), and consecutive operations could be represented compactly using techniques like run-length encoding.
- **Incremental state propagation.** Similarly to a GSet, the presented generic log-based CvRDT can be incrementally gossiped since it is a δ-CRDT [1]. This optimization is implemented in the Echo framework.
- **Garbage collection.** If the number of participants in the system is known in advance and the participants do not crash, it is possible to implement a garbage collection scheme for log operations that have been propagated to all the replicas. In many applications, however, such as online text editors where users typically connect via several short sessions on different devices, the number of participants is unpredictable. Therefore, Echo does not currently implement this optimization. For specific use-cases, Echo does however provide a low-level API that lets developers explicitly purge operations from the replicated log.

IV. Building applications with Echo

Our goal in designing the Echo API is to allow developers to seamlessly transition a non-replicated application into a replicated one, while preserving the original application semantics.

Echo comes in the form of a library written in Kotlin multi-platform and, as a result, can be integrated in JVM-based applications, mobile applications and Web applications. It assumes that the application already uses a unidirectional data flow paradigm, which is for instance the case in the code pictured in Figure 2. In other words, if the original single-node application follows the model of Definition 1, replicating it with Echo only requires minor code patches that do not affect the implementation of the $init_{app}$, $query_{app}$ and $update_{app}$ functions.

The Echo framework is open-source and available at https://github.com/markdown-party/mono. It consists of 7699 lines of Kotlin code, including tests. This comprises its basic data structures, its replication framework, and bindings for WebRTC and Websocket peer-to-peer data exchange between replicas. The Markdown Party editor is written as 4888 lines of Kotlin code, including tests. Compared to the direct use of a log-based CRDT, Echo makes it easier to replicate an application by including an incremental replication protocol supporting either WebRTC or Websocket as transport layer.

The primary objective of Echo is to be a drop-in library that does not require architectural changes to allow replication in an existing single-node application. To reduce the adoption friction, Echo uses Kotlin's built-in language features for asynchronous operations and reactive streams. Moreover, our framework perfectly preserves the existing application invariants. If used by one replica (single user, single node), no operations will ever be reverted and the behavior will be strictly identical to the one from the original application, which is something developers expect.

For this, we capitalize on some of Kotlin's specific features, including suspending functions for handling asynchronous operations and `Flows` as observable reactive streams. In the following, we illustrate the use of this API and discuss how we built a complete application with it.

A. Building a simple replicated counter

In Figure 2, a counter application supporting increment and decrement operations is illustrated. The original counter follows a uni-directional flow architecture: an `update` method holds the application update logic, which combines an operation with the current state, and returns the updated state (Fig. 2a). Note that the `initialState` variable, as well as the `update` and `query` methods directly map to the $init_{app}$, $update_{app}$ and $query_{app}$ definitions in an application model (Def. 1).

The translation of the program to use the Echo framework is systematic. The `initialState` value, `update` method, and `query` method remain untouched (Fig. 2b). The initial model is wrapped in a `MutableSite`, which is a generic type of Echo to represent a replicated state. Updates to the model are performed using a dedicated API that lets developers apply one or multiple operations atomically. The model's value can be read through the `.value` property of `MutableSite`.

Moreover, Echo embraces Kotlin suspending functions to make operations asynchronous. This makes the model updates observable (including upon integration of remote updates) through *flows*, the language's feature for observable streams.

B. Building a real-time collaborative editor

Having demonstrated how Echo can be used on a simple application model, we present how it was used to implement the replication layer of Markdown Party.[2] As showcased in Figure 3, Markdown Party is a real-time collaborative Markdown editor that lets users edit text documents in a file hierarchy. Users see their collaborators' cursors as they write and delete text. Moreover, the tree-like file and folder hierarchy is synchronized for all participants, who can choose to work online or offline. The application is web-based, local-first [10], and open-source.[3]

1) Document hierarchy invariants: Working on a hierarchy of text documents organized in directories requires respecting certain properties: it should not be possible to create cycles in the document hierarchy (Fig. 4) or to name several documents identically within the same folder. For example, concurrent modifications of a folder's parent (Fig. 4b) could result in creating an impossible cyclic state (Fig. 4c).

These semantic properties cannot be guaranteed using existing CRDTs. For example, an operation of moving documents in a tree would require the use of a log-based CRDT [9], but would still have to be extended to support the heterogeneity of the tree's elements (directories and files).

Markdown Party uses Echo to handle incompatible operations: invalid operations are skipped by the application model's *update* function, as depicted in Figure 5. The update function of an application model must be totally defined. That is, if the application semantics naturally define a partially-defined update function u', then we may replace u' by a function $f(m, x) = u'(m, x)$ if $(m, x) \in \operatorname{dom} u'$, or $f(m, x) = m$ otherwise, which is totally defined on the set of application operations E.

The choice of skipping operations if they are found to violate tree invariants is specific to Markdown Party, and moves are rarely skipped in practice. This strategy is very similar to that of [9], as concurrent moves in distinct subtrees will typically not conflict. A noticeable specificity of Echo is that it does not *require* the developer to skip conflicting operations. If the application is able to detect conflicts, it can apply a resolution strategy chosen by the developer based on the semantics of the application. Skipped operations are simply the symptom of the possible mismatch between the operations that can be generated and applied only when the application is in specific states, and the operations that can be applied when the application is in any state.

[2] https://markdown.party

[3] https://github.com/markdown-party/mono

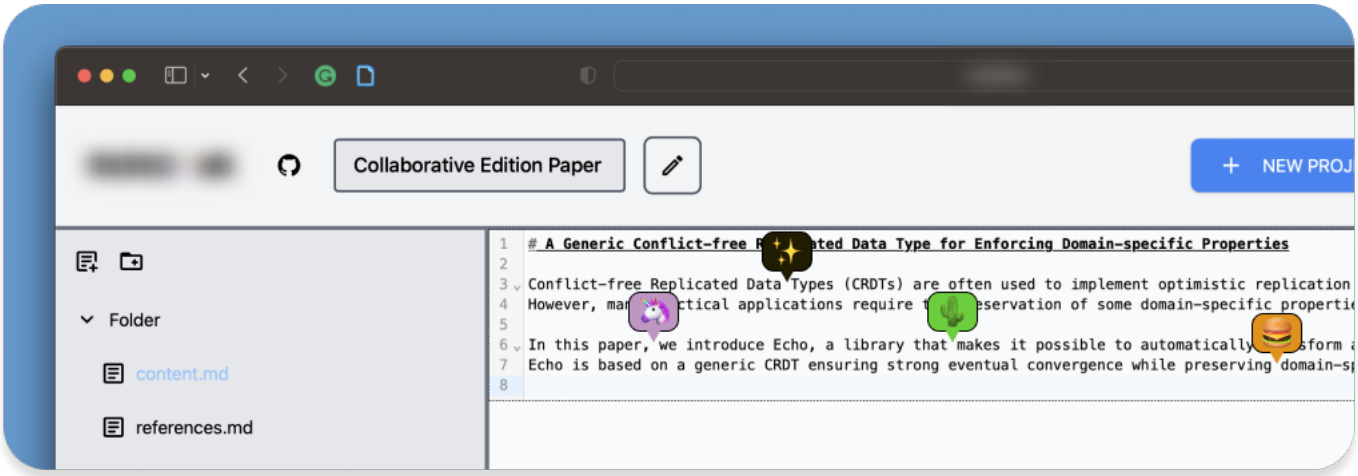
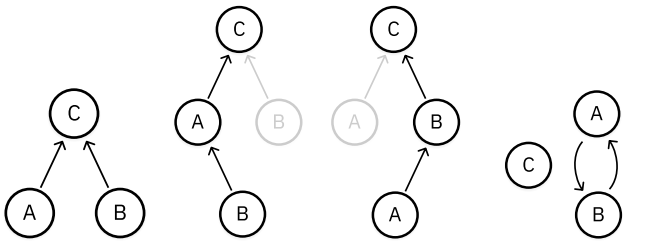


Fig. 3: Real-time collaboration in Markdown Party – Each bubble represents the cursor position of a connected user working collaboratively on the document.



(a) Initial state (b) Concurrent moves (c) Invalid tree

Fig. 4: Concurrent tree edits with unsatisfied invariants

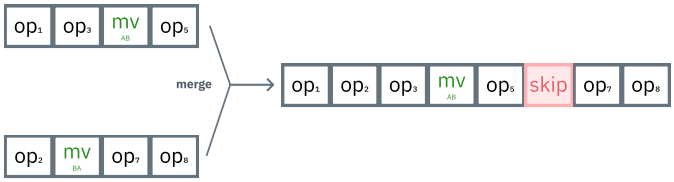


Fig. 5: The incompatible mv_{BA} operation is skipped.

2) Insertions and cursor moves atomicity: The atomicity of some operations (e.g., inserting a character and moving the cursor) would be lost if the string was implemented with a list CRDT, and the position of the cursor separately saved via a last-writer-wins (LWW) register. In Markdown Party, cursor positions are computed by taking the position of the last character insertion, deletion, or cursor move operation for each user. Therefore, the atomicity of the insertion of a character and the associated cursor movement is guaranteed at no extra cost. This would not have been the case if two specialized CRDTs (such as a Replicated Growable Array for the text and a last-writer-wins register for the last cursor position) had been used instead, and kept in sync using applicative logic. The use of the Echo framework allows for finer and more specific domain control through the instantiation of our generic CRDT that preserves domain-specific properties.

A benefit of using Echo is that the application developer does not have to care about how to implement distributed data structures that are composable, since the framework essentially treats the whole application model as a single data structure. This can be seen with the insertion and cursor moves atomicity: even if the cursor positions and text buffer of the editor are stored in two distinct data structures, such as a map and a gap buffer, if the application follows the application model structure of presented in Section III, all updates across the two different data structures forming the application state keep their single-node semantics.

V. Performance Evaluation

To evaluate the performance of Echo, we measured its throughput on a single-threaded server with an AMD Ryzen 9 5950X processor. All benchmarks were run using the Java Microbenchmark Harness (JMH) framework,[4] with 15 warm-up iterations followed by 20 measurement iterations. Despite the benchmarks being designed as single-threaded to enhance determinism, the usage of Kotlin's coroutines library enables the simulation and support of concurrent participants.

The focus of our first experiment was on measuring the anticipated performance of the Echo framework, by determining whether there were any benefits to employing a commutative and idempotent application of operations, when the domain-specific operations allow it (Section III-D), and on estimating the overhead of using the default operation serializer to store operations in the distributed log.

The following variants of a *PNCounter* [18] were implemented: 1) non-idempotent operations, 2) idempotent operations with the default serializer, and 3) idempotent operations with an optimized serializer. We evaluated each variant when varying the number of replicas connected via a star topology. In this conservative scenario, each site emitted batches of 500 increments and waited to receive all the increments of the

[4]https://github.com/openjdk/jmh

remote sites before emitting the next batch. The benchmark measured the average operation throughput per replica.

Figure 6 illustrates the importance of the implementation choices to achieve good performance when designing a CRDT: the *PNCounter* with idempotent operations and an optimized serializer processes 3 times more operations per second than a naively implemented counter. Through additional profiling, we identified that in the optimized counter, 67.5% of the time is spent integrating remote operations into the operation log, and this proportion increases to 79.5% with the naive implementation. The optimized serializer is simply some code that outputs the integer binary representation in a byte array directly. On the other hand, the default serializer is provided by the `kotlinx.serialization` framework and uses an encoding scheme similar to protocol buffers, but with non-negligible overhead.

Our second experiment measured the performance of Echo when implementing a tree hierarchy of documents and preventing the unsatisfied invariants case described in Figure 4 from happening. We evaluated two scenarios: 1) moving random folders concurrently (Fig. 7) within a hierarchy of 50 pre-existing folders, and 2) inserting documents with identical names into a single folder (Fig. 8). For both experiments, we measured the time until all participants had received all the operations and eventually reached the same state to determine the total amount of operations processed per second by all participants. The participants were organized as a fully connected network in all measurements.

Figure 7 shows that, when two participants simultaneously perform 5 concurrent moves each, the system processes up to 1749 operations per second. This number decreases to 513 operations per second when 8 participants are exchanging messages. However, the results stay equivalent when 10 moves are performed concurrently and the performance significantly decreases when 100 move operations are performed concurrently. This is probably due to the limited number of folders (50) and the large amount of concurrent moves (100), which creates more conflicts and requires additional processing time for them to be detected when applying or skipping the operations (Fig. 5).
Figure 8 also illustrates that the performance obtained when the participants generate 10 concurrent operations is worse than when the participants generate 5 concurrent operations. Here, the invariants to check (the uniqueness of a file name) are less costly than verifying the tree structure of a graph.

Note that our goal is not to achieve outstanding performance but rather to show that concurrent applications with rich semantics built with a generic log-based CRDT framework can sustain a large number of users with decent performance.

Finally, a key consideration in the evaluation of Echo's performance is the impact of the operation log size on the throughput of operations. As mentioned in Section III-D, our practical implementation eagerly computes the aggregated model, thus assuming that most concurrent operations will not require traversing the whole log. Figure 9 illustrates this property: as the number of operations in the log increases

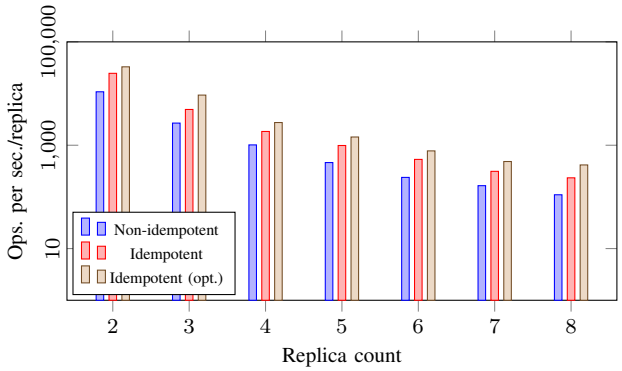


Fig. 6: Comparison of some *PNCounter* implementations

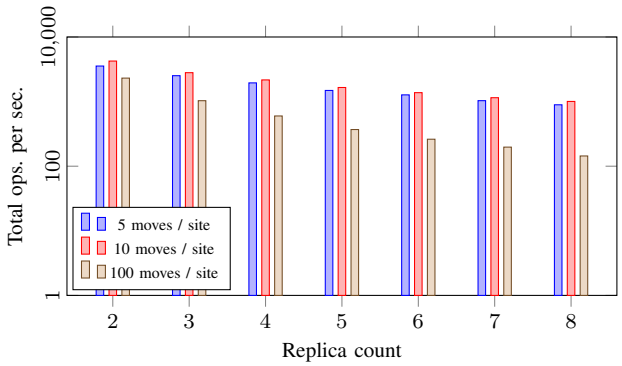


Fig. 7: Tree operations throughput in Markdown Party: each replica generates random concurrent moves in a random hierarchy of 50 folders.

from 1 to 10,000 operations, the total throughput remains identical, and Echo gracefully scales to handle large and long-lived collaborative sessions.

VI. Related work

The idea of using a distributed log has been explored in the past, starting in systems like Bayou [19]. Kleppman et al. use a log of events to implement a highly-available move operation in a tree CRDT [9], and Balegas et. al explore the preservation of application-specific invariants in [3]. Saquib et al. detail the use of version trees to represent generic operations as a log by introducing a total order to guarantee eventual convergence [15], [16]. We use the concept of an application model that maintains domain-specific properties. Markdown Party's implementation of a file hierarchy follows the same principles as [9], but we extend it with more invariants on the structure of the tree and nodes, such as enforcing the uniqueness of file names within a folder.

As our application model targets developers, we examined their practices when building CRDT applications. To do so, we searched GitHub for repositories matching the following query: "`crdt stars:>20`". The 194 results[5] belonged to different categories: libraries, distributed databases, distributed

[5]Last fetched on 2023-09-08.

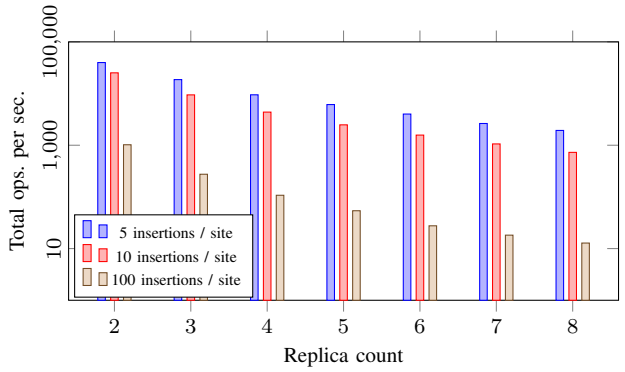


Fig. 8: Tree operations throughput in Markdown Party: each replica performs random file creations in a single folder.

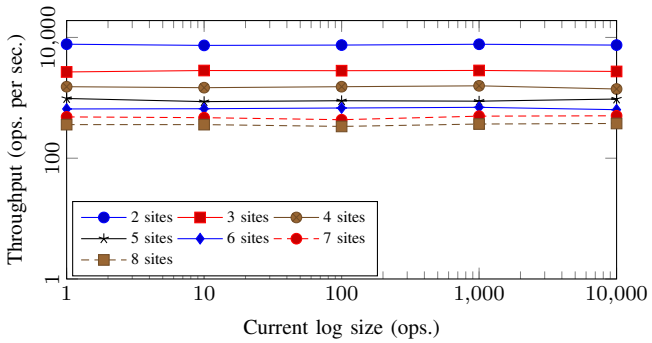


Fig. 9: Evolution of throughput with increasing log size

caches, collaborative applications, academic experiments, documentation and tutorials.

Some of the most popular libraries, such as *Automerge* and *Yjs*, were listed among the top results. These libraries usually implement robust and predefined data structures (e.g., counters, sets, lists, etc.) and require the developers to choose the best one for their domain model. *Automerge* is modeled around OpSets [8], which represent causally and totally ordered operations, but does not offer an API to build replicated data types. These data structures are often used by other projects. For instance, *Peritext* builds upon *Automerge* and introduces an algorithm consisting of three parts: generating operations, applying operations, and producing a document [14]. Likewise, several collaborative applications listed among the results rely on ipfs-log[6], an operation-based append-only log CRDT persisted on *IPFS*. Our work builds on these approaches by defining the properties of a replicated application model, and proposing a framework that uses a CRDT to replicate any application model or simulate any existing CRDT. Unlike these frameworks, Echo does not require changing the internal data structures of the application nor a complete rewrite. Instead, it leverages the existing application logic and can be used as-is with very little effort if the application already already follows a unidirectional data flow programming paradigm.

[6]https://github.com/orbitdb/ipfs-log

Many projects and libraries aim at facilitating the creation of local first software [10]. In general, the intention is to hide the complexity of building a distributed application with abstractions. For instance, the Blazes framework [2] analyzes programs to find areas needing coordination and automatically synthetizes the necessary coordination code. More recently, the *Katara* system introduced an approach that synthesizes CRDTs on the basis of a user-defined sequential type [12], hence enabling the generation of CRDTs for specific domains. Evolu[7] exposes a mutation-based API. Credt[8] enables the user to define custom data structures with a reducer-based API. Crdx[9] uses a reducer to resolve conflicts involving several properties. Our work builds upon similar ideas and introduces formal definitions.

VII. Conclusion and Future Work

In this demonstration paper, we introduced Echo, a framework that allows replicating applications with domain-specific properties. We defined the concept of an application model, an abstraction that models domain-specific operations, and a generic state-based CvRDT that replicates any application model. Our approach uses a generic and strongly eventually convergent operation log, by introducing a causal and total order on the domain-specific operations.

Two promising avenues for future work would include the automatic detection of the commutativity of domain operations, so the framework could apply the aggregated operations more efficiently to the model, and the automatic elimination of operations which can not change the state when applied.

References

[1] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Delta State Replicated Data Types. *Journal of Parallel and Distributed Computing*, 111:162–173, January 2018. arXiv:1603.01529 [cs]. URL: http://arxiv.org/abs/1603.01529, doi:10.1016/j.jpdc.2017.08.003.

[2] Peter Alvaro, Neil Conway, Joseph M. Hellerstein, and David Maier. Blazes: Coordination Analysis and Placement for Distributed Programs. *ACM Transactions on Database Systems*, 42(4):23:1–23:31, October 2017. URL: https://dl.acm.org/doi/10.1145/3110214, doi:10.1145/3110214.

[3] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, and Nuno Preguiça. IPA: invariant-preserving applications for weakly consistent replicated databases. *Proceedings of the VLDB Endowment*, 12(4):404–418, December 2018. URL: https://dl.acm.org/doi/10.14778/3297753.3297760, doi:10.14778/3297753.3297760.

[4] Christian Cachin, Rachid Guerraoui, and Luís Rodrigues. *Introduction to Reliable and Secure Distributed Programming*. Springer, Berlin, Heidelberg, 2011. URL: http://link.springer.com/10.1007/978-3-642-15260-3, doi:10.1007/978-3-642-15260-3.

[5] 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. *ACM SIGOPS Operating Systems Review*, 41(6):205–220, October 2007. doi:10.1145/1323293.1294281.

[6] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59, June 2002. doi:10.1145/564585.564601.

[7] Joseph M. Hellerstein and Peter Alvaro. Keeping CALM: when distributed consistency is easy. *Communications of the ACM*, 63(9):72–81, August 2020. doi:10.1145/3369736.

[7]https://github.com/evoluhq/evolu

[8]https://github.com/eldh/credt

[9]https://github.com/HerbCaudill/crdx

[8] Martin Kleppmann, Victor B. F. Gomes, Dominic P. Mulligan, and Alastair R. Beresford. OpSets: Sequential Specifications for Replicated Datatypes (Extended Version), May 2018. arXiv:1805.04263 [cs]. URL: http://arxiv.org/abs/1805.04263, doi:10.48550/arXiv.1805.04263.

[9] Martin Kleppmann, Dominic P. Mulligan, Victor B. F. Gomes, and Alastair R. Beresford. A Highly-Available Move Operation for Replicated Trees. *IEEE Transactions on Parallel and Distributed Systems*, 33(7):1711–1724, July 2022. Conference Name: IEEE Transactions on Parallel and Distributed Systems. doi:10.1109/TPDS.2021.3118603.

[10] 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, October 2019. Association for Computing Machinery. doi:10.1145/3359591.3359737.

[11] Sandeep S. Kulkarni, Murat Demirbas, Deepak Madappa, Bharadwaj Avva, and Marcelo Leone. Logical Physical Clocks. In Marcos K. Aguilera, Leonardo Querzoni, and Marc Shapiro, editors, *Principles of Distributed Systems*, Lecture Notes in Computer Science, pages 17–32, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-14472-6_2.

[12] Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, and Joseph M. Hellerstein. Katara: Synthesizing CRDTs with Verified Lifting, September 2022. arXiv:2205.12425 [cs]. URL: http://arxiv.org/abs/2205.12425.

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

[14] Geoffrey Litt, Sarah Lim, Martin Kleppmann, and Peter van Hardenberg. Peritext: A CRDT for Collaborative Rich Text Editing. *Proceedings of the ACM on Human-Computer Interaction*, 6(CSCW2):1–36, November 2022. URL: https://dl.acm.org/doi/10.1145/3555644, doi:10.1145/3555644.

[15] Nazmus Saquib, Chandra Krintz, and Rich Wolski. Log-Based CRDT for Edge Applications. In *2022 IEEE International Conference on Cloud Engineering (IC2E)*, pages 126–137, September 2022. URL: https://ieeexplore.ieee.org/document/9946360, doi:10.1109/IC2E55432.2022.00021.

[16] Nazmus Saquib, Chandra Krintz, and Rich Wolski. Ordering operations for generic replicated data types using version trees. In *Proceedings of the 9th Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '22, pages 39–46, New York, NY, USA, April 2022. Association for Computing Machinery. doi:10.1145/3517209.3524038.

[17] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. *A comprehensive study of Convergent and Commutative Replicated Data Types*. report, INRIA, January 2011. Pages: 50. URL: https://hal.inria.fr/inria-00555588.

[18] Marc Shapiro, Nuno 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*, Lecture Notes in Computer Science, pages 386–400, Berlin, Heidelberg, 2011. Springer. doi:10.1007/978-3-642-24550-3_29.

[19] 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. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, SOSP '95, pages 172–182, New York, NY, USA, December 1995. Association for Computing Machinery. URL: https://dl.acm.org/doi/10.1145/224056.224070, doi:10.1145/224056.224070.