Thesis Proposal: Increasing the Flexibility of Collaborative Data Structures

Matthew Weidner

Abstract

A collaborative data structure is a data structure whose state can be shared and edited across multiple devices, in the style of a collaborative app. Conflict-free Replicated Data Types (CRDTs) are a class of collaborative data structures that have seen growing interest from researchers and developers, especially as a building block for local-first software. However, CRDTs are not yet widely used in practice. This is in part due to their inflexibility: traditional CRDTs are a small number of expert-designed algorithms, which do not meet the needs of every application.

In this thesis, I propose to increase the flexibility of CRDTs. Towards this goal, I have already described composition techniques for CRDTs, and I implemented Collabs, an open-source web framework that lets programmers compose and modify CRDTs. I propose to extend this work by developing a distributed version control system for arbitrary document types, based around a flexible generalization of CRDTs.

1 Introduction

In a collaborative app, users expect to see their own operations immediately, without waiting for a round-trip to a central server. Local-first apps [25] take this further and allow users to perform local operations even when they are not connected to a central server, whether due to offline work, decentralization, or Git-style "forks" of documents [34]. Local operations may make users temporarily see different states; later, when they synchronize with each other, they will converge to a state that combines all of their operations.

In distributed systems terms, local-first apps are Available and Partition-tolerant, but only (Strongly) Eventually Consistent [41]. This is the AP side of CAP [11].

In local-first apps, as well as traditional collaborative apps and distributed version control systems, it is possible for multiple users to perform operations concurrently. The app must then decide how to "merge" those operations and give a result that makes sense to users. This is difficult: it depends not just on the app's sequential (single-user) semantics, but also on what users expect to happen in situations with no sequential analog [32, 13, 55, 49, 20].

For example, consider Figure 1, which shows a possible operation history for a collaborative recipe editor. First, one user creates a recipe with three ingredients. Proceeding to the right, two users then concurrently make conflicting edits: one deletes the second ingredient (Oil), while another edits its text to read "Olive Oil". Already there are multiple choices for what the app could do—it could delete the ingredient, ignore the deletion in favor of the text edit, or (in some implementations [2]) leave the dangling text "Olive ". Next, after seeing both operations, the first user adjusts an amount, while the second halves the entire recipe. The users probably intend

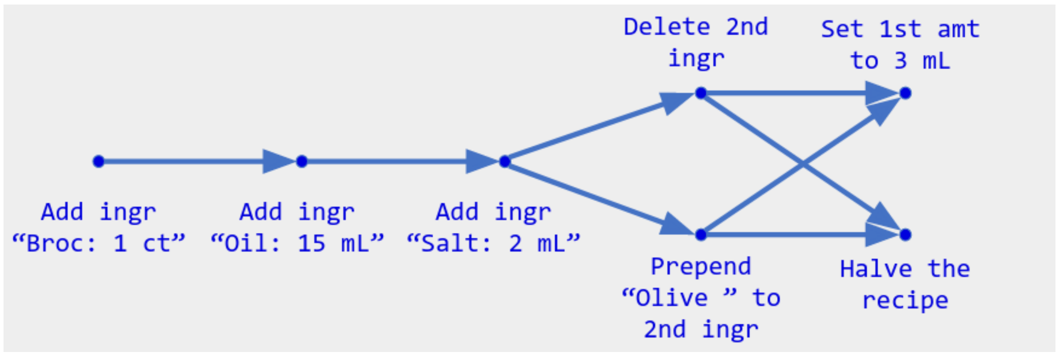


Figure 1: A possible operation history for a collaborative recipe editor. Time goes to the right, while arrows indicate operations that were aware of each other (causally related).

```
class CCounter {
  // Returns the current value.
  getValue(): number;

  // Increments the counter's value for all collaborators.
  increment(): void;

  // Calls the given callback whenever the counter's value changes,
  // including due to remote increments.
  on(eventName: "Inc", callback: (newValue: number) => void): void;
}
```

Figure 2: Possible API for a counter collaborative data structure. Real version: Collabs's CCounter docs.

for the recipe to be first adjusted, *then* halved. Even after making these choices, though, the app developer faces challenges: What state should result from general operation histories, with arbitrary concurrency? How can one compute that state efficiently?

Conflict-free Replicated Data Types (CRDTs) [40] provide a potential solution to these challenges. They are a particular class of *collaborative data structures*: programming primitives that have the API of an ordinary data structure, but wrap a "collaborative" state that is replicated across multiple devices. For example, Figure 2 shows one possible API for a collaborative counter.

Among general collaborative data structures, CRDTs are characterized by their relaxed network requirements: they can tolerate unbounded latencies, peer-to-peer networks with no central server, and end-to-end encryption. In return, CRDTs guarantee only strong eventual consistency [41]; they are incapable of strong consistency or locking.

CRDT algorithms and implementations exist for basic data structures like sets and maps [40], as well as for a few app-specific data structures like rich text [32, 17]. Academic work focuses on incremental algorithmic improvements (e.g., text-editing performance [54, 37] or specific semantic choices [55]), but these ideas are rarely implemented beyond academic prototypes. Further, existing implementations are siloed; prior work does not provide a reusable framework that researchers can extend with their own CRDT ideas and also connect to real apps.

For the vast majority of apps, including basic apps like spreadsheets and slideshows, there is no published CRDT algorithm or implementation. Creating such an algorithm from scratch is considered difficult for non-expert programmers [22]. Programmers can use general-purpose

CRDT libraries [17, 6], but these provide "one-size-fits-all" data types and semantics that are not sufficient for all applications—e.g., they do provide the list-move operation [21] needed to move ingredients in a recipe while also accepting concurrent updates to those ingredients. **Thus in practice, programmers need algorithmic techniques and frameworks that are *flexible*: they let non-expert programmers customize collaborative data structures for their own needs, instead of only providing a menu of built-in CRDTs.**

1.1 Thesis

I propose a thesis that addresses the need for flexible collaborative data structures. My proposed thesis has three main contributions.

First, I will describe composition techniques for CRDTs at a theoretical (algorithmic) level—in particular, a novel and powerful composition technique that I published at ICFP. Composition techniques let programmers create new, richer CRDTs from existing ones, allowing novice developers to add domain- and application-specific functionality while preserving correctness and performance guarantees. This is important for flexibility because creating collaborative data structures "from scratch" is challenging even for experts: one must often consider how each pair of operations interacts, and prove algebraic properties to be certain that their data structure behaves predictably [14, 22]. I will also demonstrate the composition techniques' utility by describing novel composed CRDTs for collaborative recipes, spreadsheets, and block-based rich text [45, 44].

Second, I will present Collabs, a flexible and performant CRDT collaboration framework. Collabs is a TypeScript library that lets programmers use, customize, and implement new CRDTs—in particular, they can implement new CRDTs using composition. This allows the library to support a wider variety of collaborative apps than what is currently typically attempted with CRDTs, such as collaborative recipes, slideshows, and multiplayer games, and it enables nuanced semantics like in the above collaborative recipe example. This is in contrast to the approach of most existing libraries', which focus on offering only one or a small handful of uncustomizable CRDT implementations, typically focused on some specific application like rich-text editing. Additionally, Collabs has state-of-the-art performance, supporting over 100 simultaneous active users on a rich-text editing benchmark. I have completed this work and will submit it to USENIX ATC '24 in January.

Third, for the final chapter of my thesis, I propose to design and implement a new CRDT-inspired distributed version control system. Like Collabs, this system will aim to be a practical and flexible data layer for collaborative apps. However, instead of assuming that all users want to converge to the same state, it will allow Git-style branches, forks, and merges. This is a desirable end-user feature that enables deliberate change reviews, maintaining a palette of ideas, etc. [34, 43]. Additionally, my proposed system will allow arbitrary programmer-defined updates in document histories, such as find-and-replace operations, instead of only CRDT operations that need to satisfy algebraic rules (e.g. commutativity). Thus the system contributes to my overall research goal of increasing the flexibility of collaborative data structures.

1.2 Related Work

A number of prior papers describe composition techniques for CRDTs. These include lattice composition techniques in Bloom^L [8] and LVars [28], patterns in traditional CRDT designs

[31, 1], and example composed constructions of JSON and list-with-move CRDTs [20, 21]. I am inspired by these works and implement many of their techniques in Collabs. My own papers on composition techniques fill in gaps in this literature: academic descriptions of techniques seen "in the wild" (Section 3.1.1), a new technique achieving novel semantics (Section 3.1.2), and novel examples using composition (Section 3.1.3). For details, see the related work in [48, 49].

Relatively fewer works address CRDT *flexibility*. Of the practical collaboration frameworks in Table 1 below, only the two non-CRDT frameworks allow flexible collaborative data structures [42, 39]; the CRDT frameworks each provide a few specific CRDT implementations and do not allow programmer-defined extensions. In particular, Yjs provides specific CRDTs (list, map, text, and XML) that can be nested, but not customized or encapsulated in the style of Collabs [17]; Automerge provides a single JSON CRDT that also supports counter and rich-text fields [6]; OWebSync also provides a single JSON CRDT [18]; and Legion provides basic list, map, and counter CRDTs [51]. In contrast, Collabs allows arbitrary programmer-defined CRDTs, including encapsulated objects, and allows these to be nested arbitrarily within its built-in collection types.

In the programming languages community, a recurring them in most CRDT-related works is to derive CRDT implementations from a sequential data structure. Although these works technically let one derive CRDTs for arbitrary data types, they provide only weak control of the resulting semantics (behavior under concurrency), if any. This precludes the nuanced, application-specific semantics that I seek to enable (e.g., the scaling behavior in Figure 5 below), as well as programmer-directed performance optimizations of the derived CRDTs (*performance flexibility*). MRDTs interpret a sequential data structure in terms of sets and replaces those sets with a specific set CRDT [19]. OpSets apply sequential operations in an eventually consistent total order [23]. Both systems output only a single CRDT semantics per sequential data structure. ECROs [10] and Katara [29] also derive CRDTs from sequential data structures, but they allow additional programmer input in the form of formal invariants. ECROs uses these invariants to order concurrent operations, but in doing so, it often arbitrarily *discards* operations that it cannot handle, causing unusual semantics. Katara uses program synthesis to find code respecting the application invariants, but the synthesized code's overall semantics are opaque, and so far only synthesizes simple CRDTs (e.g., counters and sets). Two more recent systems, Flec [4] and ReScala's ARDTs [27], were developed concurrently to Collabs and share some design decisions, including flexibility in the form of programmer-defined CRDTs. However, they do not yet demonstrate optimized list- or rich-text-editing CRDTs like Collabs (see Section 3.2.2), which are simultaneously difficult to implement and important for practical use [15, 16]. For details, see the related work in [50].

Turning to my proposed distributed version control system, Upwelling [34] is the most similar prior work. It is a prototype rich-text editor that supports "drafts" (forked documents) that can be edited independently of the main document and merged later, powered by a log of CRDT operations. My proposal extends Upwelling to support more general content types, branching patterns, and operations. Beyond Upwelling are Irmin [36] and Pijul [56], which are version control systems that involve CRDTs, but not in the same way as my proposal: one can build CRDTs on top of Irmin, while Pijul's text-merge algorithm uses a specific CRDT. Sterman et al. used interviews with creative practitioners (e.g., digital artists) to identify ways that the practitioners could use version control if it were available for non-code artifacts, but they do not implement a concrete system [43]. For details, see the related work in [46].

One can also relate my proposal to traditional, non-CRDT distributed version control systems like Git, RCS, Mercurial, etc., which are surveyed by Baudiš [3] and Koc and Tansel [26]. I contrast my proposal with Git, the most popular system, in Section 4. RCS's reverse- and forward- deltas, Mercurial's "revlog", and Git Pack are system-level ideas that I might reuse for my own performance optimizations. Git, Mercurial, Bazaar, and Darcs all use *textual merges*, meaning that they track and merge line-based changes to plain text files inferred from diffs [3]; this precludes the character-based granularity of live collaboration, and its diffs can fail to reflect user intent (e.g., indenting a block of text shows up as numerous mismatched line replacements). Mens [35] surveys academic works that use other merge techniques in the hopes of achieving better merge semantics. The most similar technique to my proposed system is *operation-based merging*, in which each application-level operation is tracked. While there are papers from the 1990s on operation-based merging,[1] they do not give algorithms or concrete systems to actually compute merged states from operation histories; instead, they focus on identifying merge conflicts for the end user to resolve.

2 Background

2.1 Conflict-free Replicated Data Types (CRDTs)

For brevity, this section only gives an informal definition of CRDTs, describing how they are used in a collaborative app.

A CRDT is a data structure that is shared between a group of *users*, each of whom has read or read-write access to the CRDT. To access a CRDT, a user creates a *replica* on their local device. Each replica functions as a node in a distributed system. It stores a copy of the CRDT's entire state. A user may read their local replica's state at any time, without any locking or coordination.

At any time, a user with read-write access to a CRDT may perform an *operation* on their replica of that CRDT. The replica immediately updates its copy of the CRDT's state to reflect that operation. It then communicates this operation to other replicas in the background, so that they eventually update their own states as well.

These *optimistic operations*—appearing immediately on the local device, and only propogating to other devices in the background—are a natural fit for collaborative apps, where users expect to see their own updates immediately. However, they make it possible for multiple users to update the state concurrently, possibly with unbounded communication latency. Thus designing consistent and appropriate CRDTs for a given application is challenging even at the algorithmic level.

The exact strategy for communicating operations in the background varies with the type of CRDT. Traditional academic work defines two types, *op-based CRDTs* and *state-based CRDTs*, described below. Both types are decentralized: they do not require a central authority such as a server; instead, all replicas are treated equally. In practice, CRDT libraries (including Collabs, Section 3.2) often implement "hybrid" CRDTs that support both op- and state-based communication strategies. More recent work describes additional strategies [52, 24].

Op-based CRDT Usage An *operation-based (op-based) CRDT* communicates operations to other replicas as follows. Whenever a user performs an operation on their local replica of an op-based CRDT, the CRDT generates a *message* that the user must broadcast to other replicas. Any other user may deliver that message to their own replica of the CRDT. The broadcast network

[1] These papers predate CRDTs but do anticipate some of their properties, e.g., the utility of commutative operations.

only needs to guarantee eventual at-least-once message delivery; it may be decentralized, and it may suffer unbounded network latencies.

When a replica receives a message, it does not necessarily update its own state immediately. Instead, an on-device network middleware delays or ignores messages in order to guarantee *exactly-once causal order delivery* to the underlying CRDT algorithm: each message is delivered exactly once, and a message m is not delivered until after all messages that are causally prior to m (i.e., happened-before m) [30].[2]

Assuming exactly-once causal order delivery, an op-based CRDT guarantees *strong convergence*: two replicas that have received the same messages will be in equivalent states. For example, if two users make concurrent offline edits to their own replicas of a shared recipe, then once they come online and exchange messages, they will converge to the same state. (The contents of this state are determined by the CRDTs' *semantics*; there may be multiple possible semantics for a given abstract data type.) It follows that, in any network that eventually delivers messages at-least-once, a CRDT satisfies *strong eventual consistency*: two replicas who stop performing operations will eventually be in equivalent states [41].

Internally, op-based CRDTs guarantee strong convergence by ensuring that concurrent messages *commute*: delivering them in either order results in the same state. This is necessary because different users may receive concurrent messages in different orders.

State-based CRDT Usage A *state-based CRDT* communicates operations to other replicas as follows. At any time, a user's app may send a serialized copy of its local replica's state to other users. Any user may "merge" this state with their own state-based CRDT state. Doing so updates their state to reflect all of the operations incorporated into the serialized state, skipping duplicates.

Merging two state-based CRDT states effectively merges the corresponding operation histories.[3] In a hybrid op-/state-based CRDT library like Collabs, merging a state has the same effect as delivering all of the op-based messages that contributed to that state, skipping duplicates.

2.2 Distributed Version Control Systems

A *distributed version control system* is a version control system that stores the entire project history on each user's device and allows them to make edits even when offline [3]. Users share changes peer-to-peer using push and pull operations, instead of relying on a central repository server.

Git [5] is the current most popular distributed version control system. It is primarily used for version control of source code, and its commits and merges track line-based changes to plain text files. These changes are inferred from before-and-after files on disk using a diff algorithm. Users group changes into "commits", which are further grouped into linear *branches*. Branches may fork (diverge), merge (converge), or exchange commits via rebasing or cherry-picking. Git attempts to merge concurrent commits automatically using a 3-way merge that aligns lines of text; when this fails, the user is presented with a conflict that they must merge manually. *Pull requests* create a process around merging that gives an opportunity for manual review regardless of whether conflicts occur.

[2]The *causal order* is the partial order $<$ on messages defined by the transitive closure of the rule: if a replica sent or received message m before sending m', then $m < m'$.

[3]Note that a state may lossily encode the original operations, e.g., by omitting text that has since been deleted.

Distributed version control systems and CRDTs both feature state replicated between users, optimistic updates, and support for decentralized networks. One major difference is that CRDTs typically learn of operations directly from an application (via data structure method calls), while version control systems typically infer operations indirectly from changes to files on disk. My proposed version control system in Section 4 adopts the former (CRDT) technique as a key design decision.

3 Completed Work

3.1 Composition Techniques for CRDTs

A CRDT composition technique is a construction that inputs one or more existing CRDTs and outputs a new "composed" CRDT. Composition contrasts with the traditional approach to CRDT design, in which one directly specifies algorithms for op-based message processing and state-based merging, then manually proves strong convergence. Instead, composition techniques ensure that their outputs automatically satisfy correctness properties like strong convergence. They also let one reuse the component CRDTs' existing implementations, which are probably already tested and optimized. Thus composition aids flexibility by making it easier to create new CRDTs.

In my thesis, I will describe the following completed work, which concerns CRDT composition techniques at a theoretical level.

3.1.1 Collections of CRDTs

One class of composition techniques inputs a CRDT and outputs a collection whose elements are instances of that CRDT. For example, a recipe has a list of ingredients; thus a CRDT modeling a collaborative recipe can contain a list of "ingredient CRDTs".

Several collections of CRDTs appear in production systems but not the academic literature. In published work [48, 49], I and collaborators mathematically describe the algorithms behind two of these collections, the Riak map [2] and Yjs's `Y.Array` [17]. This work serves to document the CRDTs' semantics, verify strong convergence, and bring them to the attention of researchers.

3.1.2 Semidirect Product of CRDTs

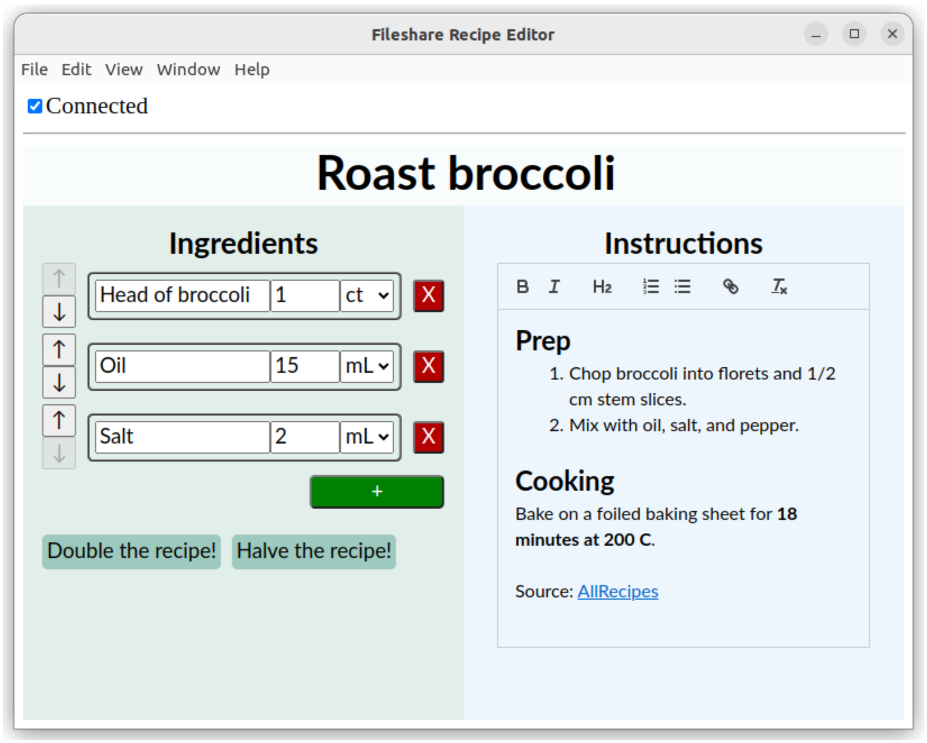
I and collaborators also describe a novel composition technique, the *semidirect product of CRDTs*, in an ICFP paper [48]. This technique lets an op-based CRDT support operations that do not naturally commute. Instead, the CRDT designer chooses:

1. An *arbitration order*, stating that in the face of concurrent non-commuting operations, one operation should be applied before the other.

2. An *action*, which is used to emulate the arbitration order on replicas that receive the operations in the opposite order. The action must satisfy some algebraic rules.[4]

For example, a collaborative set has operations $\mathsf{add}(x)$ and $\mathsf{remove}(x)$ to add or remove a value x from the set. These operations don't naturally commute: if two users concurrently perform

[4]The algebraic rules can be viewed as a simple special case of operational transformation [38], an approach to collaborative data structures that predates CRDTs.



Roast broccoli

Head of broccoli	1	ct
Oil	15	mL
Salt	2	mL

Prep

1. Chop broccoli into florets and 1/2 cm stem slices.
2. Mix with oil, salt, and pepper.

Cooking

Bake on a foiled baking sheet for **18 minutes at 200 C.**

Source: AllRecipes

Figure 3: A collaborative recipe editor built on top of Collabs.

operations add(x) and remove(x), then replicas who apply add(x) before remove(x) will get a different result (x not in the set) than replicas who apply remove(x) before add(x) (x in the set). However, one can choose either arbitration order, find a corresponding action, and form the semidirect product. This process gives valid CRDTs: the classic remove-wins and add-wins set CRDTs [40].

3.1.3 Example Apps

In informal work targeted at programmers, I use composition to describe novel CRDT algorithms for several collaborative apps. These examples demonstrate that composition is a useful way to create new CRDTs, hence it increases their flexibility (at least in theory).

One example app is a collaborative recipe editor [44]. I have implemented this example using Collabs (Section 3.2), with the GUI shown in Figure 3. Its CRDT has the following composed design, which mirrors the GUI's HTML structure:

- Each "ingredient CRDT" is the object composition of three primitive CRDTs: a text-editing CRDT [47] for the description, and last-writer-wins registers for the amount and units [40]. Here *object composition* is a composition technique from prior work (e.g., Bloom's lattice map [8]) that combines multiple independent CRDTs side-by-side, encapsulated in an object-oriented class.

- The ingredients list is a collection of CRDTs modeled after Yjs's `Y.Array` (Section 3.1.1) and Kleppmann's list-with-move [21], with ingredient CRDT elements.

- The overall recipe is the object composition of the ingredients list, a last-writer-wins register for the title, and a Peritext rich-text CRDT [32] for the instructions.

- The “Double the recipe!” and “Halve the recipe!” buttons were originally implemented using a semidirect product (Section 3.1.2), but currently use a simpler object composition: there is a last-writer-wins register controlling the global scale (i.e., how many servings of the recipe to display); each ingredient CRDT stores the amount-per-serving; and the GUI renders their product.

The other example apps are a collaborative spreadsheet [45] and a block-based rich-text editor [44]. For these, I describe compositional CRDT algorithms but do not give implementations.

3.2 Collabs: A Flexible and Performant CRDT Collaboration Framework

The work described so far concerns CRDT algorithms. In other completed work, I lead the development of Collabs, a CRDT collaboration framework that prioritizes flexibility [50]. Unlike existing collaboration frameworks, Collabs allows application programmers to implement custom CRDTs for their own apps through composition, and it allows researchers to implement and publish arbitrary CRDT algorithms as library extensions—including new composition techniques.

Collabs is written in TypeScript and published on npm.[5] It is open source on GitHub,[6] fully documented,[7] and comes with several demo collaborative apps. It also provides practical tools like a testing server and React hooks.

In addition to these programmer-facing features, I hope that Collabs can serve as a “CRDT laboratory” for academic research: researchers can implement and publish new CRDTs as extensions to Collabs. That would let app programmers use new CRDT algorithms immediately, mix-and-match approaches, and even optimize existing approaches. Extensions can be used in concert with the rest of Collabs’s CRDTs and practical plugins, instead of requiring a new collaboration framework per research paper.

3.2.1 Collabs’s Design

Collabs is designed to meet three key requirements.

Flexibility Collabs is flexible, allowing any programmer to implement new CRDTs alongside the framework. They might do so to choose custom semantics for their app in the face of concurrent edits, or to optimize performance for their app. Each CRDT is implemented as its own class with its own strongly-typed API; thus Collabs looks more like the Java Collections framework than like a NoSQL database, in contrast to prior work.

Composition Collabs supports composing existing CRDTs to create new ones. In particular, it is possible to implement the example applications from Section 3.1.3 in a similar style to traditional (non-collaborative) object-oriented apps.

Practicality Collabs aims to be practical enough to use in real applications. In particular, it has state-of-the-art performance in our benchmarks, and it is compatible with existing JavaScript tools like WebSockets and the Quill rich-text editor.

[5]`https://www.npmjs.com/package/@collabs/collabs`
[6]`https://github.com/composablesys/collabs`
[7]`https://collabs.readthedocs.io/`

To illustrate these requirements, consider Collabs's recipe editor demo.[8] It implements the composed design from Section 3.1.3 with the GUI shown in Figure 3. To validate Collabs's usefulness for local-first apps, the demo is a desktop app that saves its state to local files. Collaboration occurs through Dropbox; the files are written in such a way that there are no sync conflicts, using state-based CRDT states.

The app's CRDT code has the following object-oriented outline, mirroring the composed design:

```
// "extends CObject" indicates object composition.
class CIngredient extends CObject {
  // A text-editing CRDT.
  text: CText;
  // A custom CRDT that wraps a last-writer-wins
  // register, linking it to the global scale.
  amount: CScaleNum;
  // A last-writer-wins register with enum values.
  units: CVar<Unit>;

  ...
}

class CRecipe extends CObject {
  // A last-writer-wins register with string values.
  recipeName: CVar<string>;
  // A list-with-move collection of CRDTs. Note that
  // the elements are our custom CIngredient CRDT.
  private _ingrs: CList<CIngredient>;
  // A Peritext rich-text CRDT.
  instructions: CRichText;

  // CObjects can encapsulate their component CRDTs:
  addIngredient(index: number) {
    this._ingrs.insert(index);
  }

  ...
}
```

Internally, Collabs translates this outline into a tree of CRDTs, as shown in Figure 4. It uses the tree to (de-)multiplex op-based messages and state-based states, so that each CRDT can follow a traditional algorithm that assumes exclusive network access. The tree is carefully designed to maximize flexibility; for example, we can store our custom `CIngredient` CRDT as elements in Collabs's built-in `CList` collection, unlike previous libraries that only support nesting of built-in CRDTs [17, 6].

Besides composition, the recipe editor demo demonstrates Collabs's flexibility, by implementing several nuanced and app-specific semantics. For example, Figure 5 shows what happens if one user edits an ingredient's amount, while concurrently, another user clicks the "Halve the recipe!" button: the new amount is also halved, so that the recipe stays in proportion.

[8] Source code is available at `https://github.com/mweidner037/fileshare-recipe-editor/`.

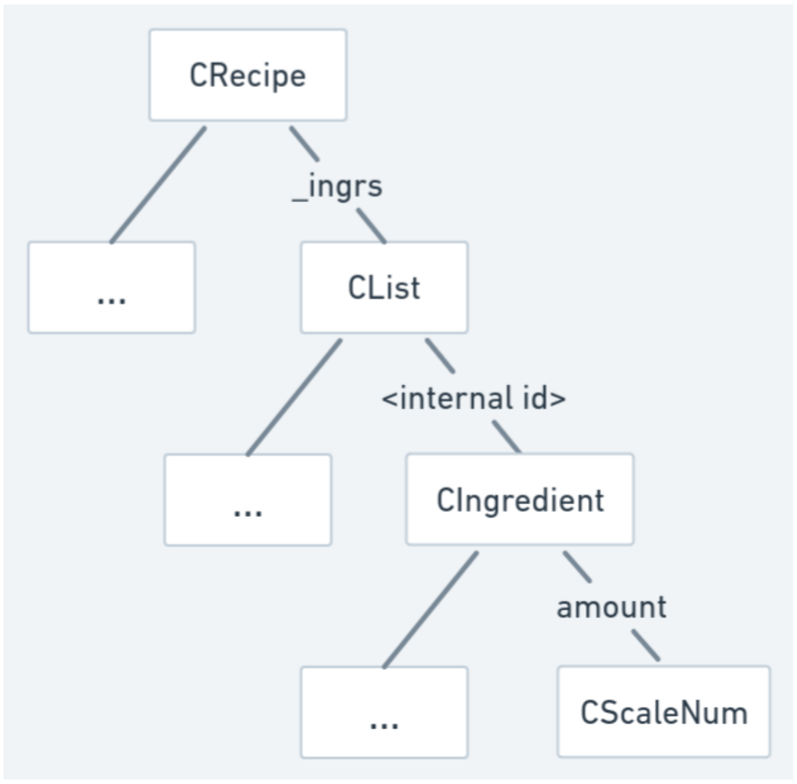


Figure 4: The tree of CRDTs corresponding to a `CRecipe`.

3.2.2 Evaluation

We evaluated Collabs by comparing its capabilities and performance to existing collaboration frameworks, including non-CRDT frameworks.

Capabilities Table 1 summarizes the capabilities of the compared frameworks. These capabilities determine the kinds of apps that are possible or easy to program on top of a given framework. A ✓ indicates full support for a given capability; ✗ indicates no support; and $\frac{1}{2}$ indicates limited support. Briefly, the capabilities are:

Local-first The framework is local-first in the sense of Kleppmann et al. [25].

Rich-text editing The framework has built-in support for collaborative rich-text editing.

Nested data The framework supports arbitrarily nested data.

List-with-move The framework supports moving elements in lists [21].

Encapsulated data models The framework lets an app define encapsulated, type-safe data models for portions of its own state.

Semantic flexibility & performance flexibility The framework lets an app customize its collaborative data structures' semantics (behavior under concurrency) and low-level performance.

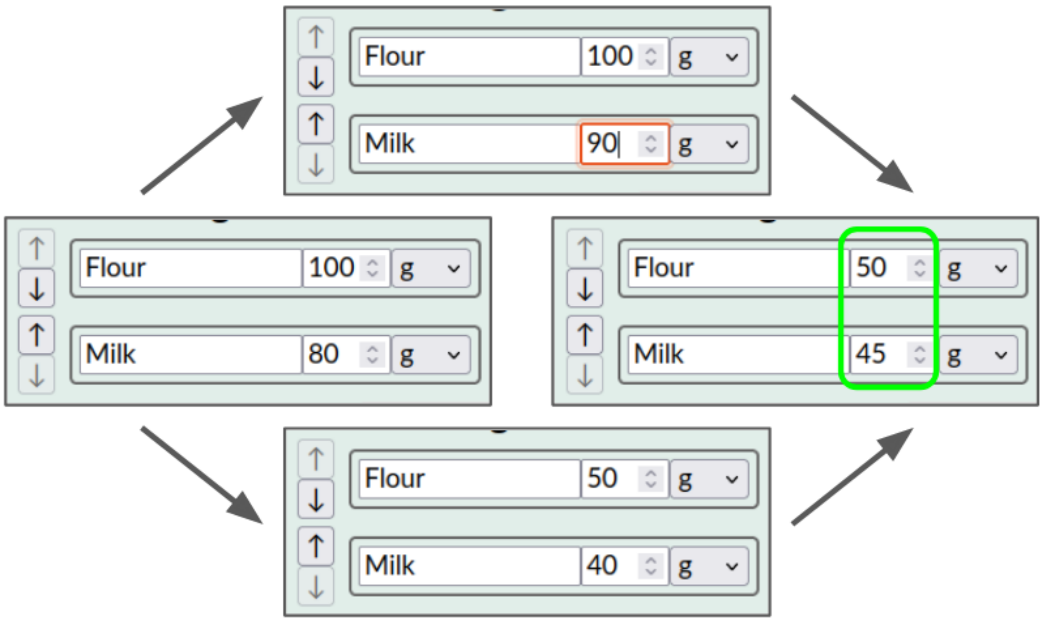


Figure 5: Behavior when one user adjusts a recipe (top) while another halves it concurrently (bottom).

	Collabs	Yjs [17]	Automerge [6]	Legion [51]	ShareDB [42]	OWebSync [18]	Replicache [39]
Local-first	✓	✓	✓	✓	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
Rich-text editing	✓	✓	✓	✗	✓	✗	✗
Nested data	✓	✓	✓	✗	✓	✓	✗
List-with-move	✓	$\frac{1}{2}$	$\frac{1}{2}$	✗	✗	✗	✗
Encapsulated data models	✓	$\frac{1}{2}$	✗	N/A	✗	✗	N/A
Semantic flexibility & performance flexibility	✓	✗	✗	✗	$\frac{1}{2}$	✗	✓

Table 1: Collabs's capabilities compared to other collaboration frameworks.

Performance We evaluated Collabs's performance using a collaborative rich-text editing benchmark.[9] This is a popular but difficult collaborative app. It is especially difficult to scale to a large number of users. For example, Google Docs limits a document to 100 simultaneous editors [12], and experiments with the 2016 version revealed high end-to-end latencies well before reaching the then-limit of 50 editors [9].

Specifically, we implemented a collaborative rich-text editor using Collabs's built-in CRDTs, then asked: how many simultaneous users can this editor support before the user experience breaks down, for realistic workloads? How does that compare to similar editors built on top of other collaboration frameworks, and what are the bottlenecks to further scaling? Our experiments used simulated users running on AWS containers, connected to a server in a different region.

Table 2 shows our main results. We found that Collabs scaled to 112 simultaneous active users, while Google Docs scaled to less then 32—despite its documented 100 editor limit. Collabs's

[9] Source code is available at `https://github.com/composablesys/collabs-rich-text-benchmarks`.

Framework	Max Users
Collabs	112
CollabsNoVC	112
Yjs [17]	96
Automerge [6]	16
ShareDB [42]	112
GDocs	16

Table 2: The maximum number of users for which each framework was fully functional. *CollabsNoVC* is Collabs with an option set that disables explicit causal-order delivery enforcement for op-based CRDTs (cf. Section 2.1); this is safe in networks that use a central server for op-based messages.

Framework	E2E latency, median (ms)	E2E latency, 95th percentile (ms)	Server CPU (%)	Client CPU (%)	Client memory (MiB)	Client network (KiB/sec)
Collabs	864	1436	10	80	345	187
CollabsNoVC	810	1373	9	79	345	39
ShareDB	2529	5367	101	81	333	36

Table 3: Performance comparison for Collabs and ShareDB with 112 users.

scalability was matched only by ShareDB, a non-CRDT collaboration framework that requires a central server.[10] Even then, Collabs had better end-to-end latency—the time from when one user performs an edit until other users see it—and much lower server CPU usage, with only a modest penalty in client-side resource usage; see Table 3.

4 Proposed Work: A CRDT-Inspired Distributed Version Control System

Local-first software [25] is a programming paradigm that combines the user-centric benefits of traditional desktop apps with the collaborative features of cloud-based apps. The Git distributed version control system is a prominent example: it stores all of its data in local files, but it also supports sharing changes with remote collaborators.

However, Git has two key shortcomings identified by Kleppmann et al. [25]. First, Git does not support real-time, conflict-free collaboration in the style of Google Docs. Second, its histories, diffs, and merges only understand line-based updates to plain text documents; they do not track finer-grained edits (e.g., per-character) or updates to non-text content types.

For my last thesis project, I propose to design and implement a CRDT-inspired distributed version control system that overcomes these shortcomings. From the end-user's perspective, my proposed system would have the following features:

1. Real-time, conflict-free collaboration in the style of Google Docs.

2. Versions, diffs, forks, and merges in the style of Git.

[10]Collabs provides a central server plugin, but it is not mandatory to use.

3. The ability to support arbitrary content types with nuanced, type-specific merge semantics: spreadsheets, recipes, digital art, etc.

To see why such a system is desirable, consider a team working on their company's quarterly report spreadsheet. The spreadsheet includes both modeling formulas made by financial analysts and per-quarter data entered by interns. My proposed system would allow separate branches for the analysts (to improve their formulas), interns (to enter the next quarter's data), and the manager (to maintain a published copy that accepts only urgent revisions). The manager could preview and merge the other two branches when ready, and the analysts could make an additional branch to test their draft formulas against the next quarter's data.

Internally, the system will be based on a flexible generalization of CRDTs that I propose below. It uses branches that are each an ordered log of application-defined updates. This generalization is a research contribution in itself, and it opens the door to research on collaborative operations that are difficult to incorporate into CRDTs. On the systems research side, existing version control systems apply numerous interesting techniques to enable time- and disk-efficient operations: Git's Patch format, Mercurial's revlogs, and RCS's forward and reverse deltas [3]. I am interested to explore and benchmark these techniques with the added challenges of fine-grained, application-defined updates and states. Finally, future work could investigate the system's user experience, i.e., how to expose its features in a usable way.

4.1 Design

I have already begun exploring an approach that I presented at the Programming Local-First workshop [46]; the description below incorporates changes based on feedback at the workshop. The design takes inspiration from Git [5], OpSets/Automerge [23, 6], Replicache [39], and others.

4.1.1 Branches

The system's central concept is a *branch*. Like in Git, a branch is an ordered log of updates to a document, stored on a single device. Unlike in Git, a branch's updates are fine-grained and application-specific: e.g., individual cell edits and formatting operations in a spreadsheet. Updates are written and read directly by a user-facing application.

The format of updates is arbitrary and chosen by the application, like an ordinary file's contents. The current state of a branch is given by applying an application-specific *reducer function*

$$\mathsf{reduce} : (\text{current state}) \times (\text{next update}) \to (\text{new state})$$

to the branch's updates in order; this programming pattern is already familiar to web programmers from Redux [7]. The state and updates might use an op-based CRDT's state and messages (especially for text editing—see Figure 7), but this is not a requirement: branches meant to converge to the same state will have the same updates *in the same order*, so there is no formal need for concurrent updates to commute.[11]

For example, a spreadsheet app could use updates and a reducer function like the following pseudocode:

[11]This idea comes from OpSets [23]. In contrast to OpSets' Lamport-timestamp order, though, I plan to give users some control over the update order via the semantics of merging, cherry-picking, and push-pull.

```
// Update to set a cell.
{
  type: "set",
  rowID: <UUID>,
  columnID: <UUID>,
  value: "= 3 * 5"
}

// Update to find & replace.
{
  type: "find-replace",
  find: "2023",
  replace: "2024"
}

// Reducer function
function reduce(state: Map<[RowID, ColumnID], string>, update) {
  switch (update.type) {
    case "set":
      return state.set([update.row, update.column], update.value);
    case "find-replace":
      ...
  }
}
```

Separating updates from state mutations via the reducer function has the practical benefit that it makes schema migrations easier: if an app developer wishes to change the state's schema in a new release of the app, they can leave existing documents' updates as-is, instead changing the reducer function. This contrasts with existing CRDT frameworks whose update histories remember only the operations on an underlying CRDT state [6], making future schema migrations possible but more difficult [33].

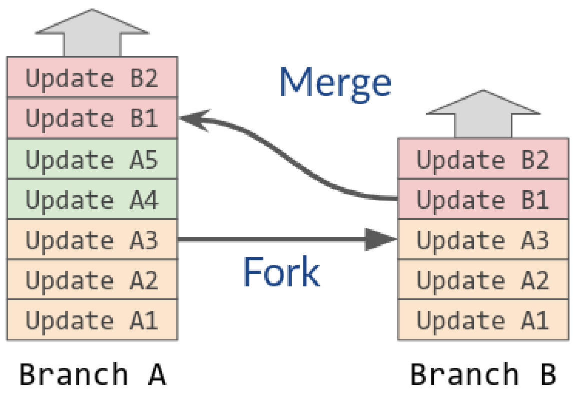
4.1.2 Collaboration

Since each branch is local to a device, collaboration involves copying updates between branches. There are several forms of collaboration:

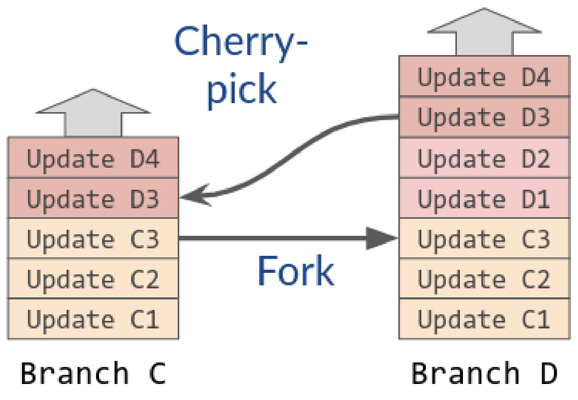
Forking and merging At any time, a user can *fork* a branch that they have read access to. This creates a new branch whose initial update log is a copy of the base branch's. Later, a user can *merge* the new branch into the base branch, copying over its new updates on top of the base update log (Figure 6(a)). Or, a user can *cherry-pick* a subset of updates from the new branch to the base branch (Figure 6(b)). Either way, users may choose to preview a *diff* between the base branch's current and new states, and manually fix issues, before committing the operation.

In the above example of a quarterly report spreadsheet, diff-and-merge is how the manager merges changes into the published spreadsheet. If one sheet is ready but others need further work, the manager could choose to cherry-pick only that sheet's updates.

Cross-device sync Branches meant to be replicas on different devices are kept in-sync by explicit "push" and "pull" operations. Users can treat one device's branch as canonical by always pushing updates on top of its update log—treating that branch as append-only—and



(a) Merging branch B into branch A.



(b) Cherry-picking a subset of updates from branch D to branch C.

Figure 6: Scenarios where updates move between branches in an interesting way.

then copying its log order to other branches, which automatically "rebase" their recent updates on top of the pulled updates. However, a canonical branch is only a social convention, not a technical requirement.[12]

In the quarterly report example, the analysts and interns would each maintain a canonical branch on the company's server. An analyst updating a complex formula could try out several variations and only push their updates to the canonical branch once it is stable. In the meantime, they put off pulling their peers' updates to avoid distractions.

Live collaboration Multiple users can set up their branches to automatically sync updates, either for a temporary live collaboration session, or to do cross-device sync without explicit push/pull. Here the system must take care to ensure that all branches converge to the same order of updates, like in OpSets [23]. Even when concurrent updates do not commute, their precise order is not important because unintended results can be noticed and fixed live.

In the quarterly report example, the analysts could collaborate live during a meeting, so that they can see and discuss each others' changes in real time—without the need to push and pull.

4.2 Relation to CRDTs

I mentioned above that it is not a requirement to use op-based CRDT messages as updates. At first, this sounds like it would throw away the guarantees that CRDTs work so hard for (e.g., strong eventual consistency). However, I believe that this relaxation is both necessary and useful.

First, it is not a given that users will want all branches to be eventually consistent with each other. In git repos, it is common to maintain a "downstream" branch that applies specialized patches on top of a generic "upstream" branch—e.g., for a more specialized version of a piece of software, or in forks where developers intend to take a different approach than upstream. In creative design, practitioners often maintain old versions as a palette of ideas without intending to merge them [43]. These situations challenge the utility of eventual consistency's guarantees.

[12]This contrasts with my original workshop paper, which required a canonical "homeserver" to enforce an append-only log order for each branch [46].

Second, there are situations where one branch may want to omit changes from a merged-in branch. For example, a reviewer may choose to reject changes, or they may find that the branch is only partially complete—e.g., one sheet in a spreadsheet is ready but the others are not. It is then natural to cherry-pick only the desired updates, as illustrated in Figure 6(b): updates D3 and D4 are copied from branch D, but not the (causally) related updates D1 and D2. However, deliberately dropping causal dependencies is a fundamental difference from the CRDT approach; most CRDT algorithms assume that all updates eventually arrive, and that there exists a causal order between them. Missing updates, as in the cherry-picking case, break a core assumption of CRDTs, but they are nonetheless a desirable outcome in some scenarios.

Third, CRDTs' guarantees are somewhat orthogonal to what we really care about when merging: getting a merged result that roughly matches users' intentions and requires minimal manual fixing. Indeed, strong eventual consistency technically allows "consistent nonsense", in which merging leads to useless or even errored states [53]. Dually, it is hard to make a commutative, CRDT version of a find-and-replace operation, but this operation is still useful for computing merges: e.g., merging in a branch that replaced "2023" with "2024" should likely apply the same transformation to the target branch.

CRDT ideas are still useful when formulating updates, and that is why I consider this system "CRDT-inspired". For example, updates involving text editing should use list CRDT positions (e.g., Fugue node IDs [47]) instead of array indices, to avoid bad merge results like in Figure 7. Likewise, my completed work on nuanced, app-specific semantics in the face of concurrency extends to semantics in the face of merging.

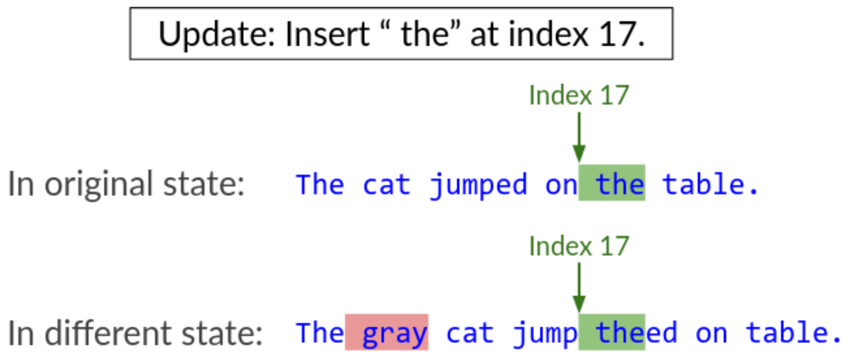

Update: Insert " the" at index 17.
Index 17
In original state:
The cat jumped on the table.
Index 17
In different state:
The gray cat jump theed on table.

Figure 7: Updates that use ordinary list indices may not "make sense" when applied to a different state—e.g., if they are applied on top of a more advanced branch during merging (Figure 6(a)). List CRDT positions solve this problem.

4.3 Challenges

I anticipate the following challenges while implementing the above design.

Performance A naive implementation of the above design would store every branch as a literal log of updates, and compute the state of a branch by applying the reducer function to every update. This may lead to impractical performance in terms of disk storage and branch switching times. To evaluate this performance, I plan to simulate long-lived document histories and benchmark the time and disk space needed for these tasks. Specifically, I will track the size of storing n branches with a shared history, which should be much less than n

times the single-branch size; and the time needed to switch between branches with various degrees of related-ness, which should not exceed several seconds in any case. As needed, I will pursue investigate optimizations inspired by Git, Mercurial, and RCS's internals, such as de-duplicating branches with shared histories, caching state snapshots, and computing inverses of updates.

API Applications built on top of the proposed distributed version control system will need novel APIs to interact with branches. To evaluate the APIs I develop, I plan to implement 3–5 simple prototype apps, including basic spreadsheet and drawing apps, and keep them updated as I evolve the underlying API, to evaluate how the developer experience changes. Here I can draw on my experience developing Collabs's API.

Keeping things simple To minimize risk and stay on track for completion, I plan to omit many useful but difficult features, such as Google Docs-style rejectable changes and text-editing specific optimizations.

5 Timeline

Feb–Mar VCS project: prototype branching and merging with trivial app. Writing: Completed Work chapters.

Apr–May VCS project: investigate optimized storage and branch switching. Writing: Background and Introduction chapters.

Jun–Jul VCS project: evaluation and write-up.

Aug–Sep Finishing touches and thesis defense.

References

[1] Carlos Baquero, Paulo Sérgio Almeida, Alcino Cunha, and Carla Ferreira. Composition of state-based CRDTs. Technical report, HASLab, INESC TEC, May 2015. `http://haslab.uminho.pt/cbm/files/crdtcompositionreport.pdf`.

[2] Basho. Riak datatypes, 2015. `http://github.com/basho`.

[3] Petr Baudiš. Current concepts in version control systems, 2014. `http://arxiv.org/abs/1405.3496`.

[4] Jim Bauwens and Elisa Gonzalez Boix. Nested Pure Operation-Based CRDTs. 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 2:1–2:26, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[5] Scott Chacon and Ben Straub. *Pro Git*. Apress, 2 edition, 2014.

[6] Automerge contributors. Automerge, 2023. `https://automerge.org/`.

[7] Redux contributors. Redux. GitHub repository, 2023. `https://github.com/reduxjs/redux`.

[8] Neil Conway, William R. Marczak, Peter Alvaro, Joseph M. Hellerstein, and David Maier. Logic and lattices for distributed programming. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, New York, NY, USA, 2012. Association for Computing Machinery.

[9] Quang-Vinh Dang and Claudia-Lavinia Ignat. Performance of real-time collaborative editors at large scale: User perspective. In *2016 IFIP Networking Conference (IFIP Networking) and Workshops*, pages 548–553, 2016.

[10] 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), oct 2021.

[11] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, jun 2002.

[12] Google. Share files from google drive, 2023. `https://support.google.com/docs/answer/2494822`.

[13] Claudia-Lavinia Ignat, Luc André, and Gérald Oster. Enhancing rich content wikis with real-time collaboration. *Concurrency and Computation: Practice and Experience*, 33(8):e4110, 2021. e4110 cpe.4110.

[14] Abdessamad Imine, Pascal Molli, Gérald Oster, and Michaël Rusinowitch. Proving correctness of transformation functions in real-time groupware. In Kari Kuutti, Eija Helena Karsten, Geraldine Fitzpatrick, Paul Dourish, and Kjeld Schmidt, editors, *ECSCW 2003*, pages 277–293, Dordrecht, 2003. Springer Netherlands.

[15] Kevin Jahns. Are CRDTs suitable for shared editing?, August 2020. `https://blog.kevinjahns.de/are-crdts-suitable-for-shared-editing/`.

[16] Kevin Jahns. Crdt benchmarks, 2020. `https://github.com/dmonad/crdt-benchmarks`.

[17] Kevin Jahns. Yjs, 2023. `https://docs.yjs.dev/`.

[18] Kristof Jannes, Bert Lagaisse, and Wouter Joosen. Owebsync: Seamless synchronization of distributed web clients. *IEEE Transactions on Parallel and Distributed Systems*, 32(9):2338–2351, 2021.

[19] Gowtham Kaki, Swarn Priya, KC Sivaramakrishnan, and Suresh Jagannathan. Mergeable replicated data types. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[20] M. Kleppmann and A. R. Beresford. A conflict-free replicated json datatype. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):2733–2746, 2017.

[21] Martin Kleppmann. *Moving Elements in List CRDTs*. Association for Computing Machinery, New York, NY, USA, 2020.

[22] Martin Kleppmann. Assessing the understandability of a distributed algorithm by tweeting buggy pseudocode. Technical Report UCAM-CL-TR-969, University of Cambridge, Computer Laboratory, May 2022. https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-969.pdf.

[23] Martin Kleppmann, Victor B. F. Gomes, Dominic P. Mulligan, and Alastair R. Beresford. Opsets: Sequential specifications for replicated datatypes (extended version), 2018. https://arxiv.org/abs/1805.04263.

[24] Martin Kleppmann and Heidi Howard. Byzantine eventual consistency and the fundamental limits of peer-to-peer databases, 2020. http://arxiv.org/abs/2012.00472.

[25] 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, page 154–178, New York, NY, USA, 2019. Association for Computing Machinery.

[26] Ali Koç and Abdullah Uz Tansel. A survey of version control systems. In *The 2nd International Conference on Engineering and Meta-Engineering*. International Institute of Informatics and Systemics, 2011.

[27] Christian Kuessner, Ragnar Mogk, Anna-Katharina Wickert, and Mira Mezini. Algebraic Replicated Data Types: Programming Secure Local-First Software (Artifact). *Dagstuhl Artifacts Series*, 9(2):26:1–26:4, 2023.

[28] Lindsey Kuper and Ryan R. Newton. Lvars: Lattice-based data structures for deterministic parallelism. In *Proceedings of the 2nd ACM SIGPLAN Workshop on Functional High-Performance Computing*, FHPC '13, page 71–84, New York, NY, USA, 2013. Association for Computing Machinery.

[29] Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, and Joseph M. Hellerstein. Katara: Synthesizing crdts with verified lifting. *Proc. ACM Program. Lang.*, 6(OOPSLA2), oct 2022.

[30] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, jul 1978.

[31] Adriaan Leijnse, Paulo Sérgio Almeida, and Carlos Baquero. Higher-order patterns in replicated data types. In *6th Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC 2019. ACM, March 2019.

[32] Geoffrey Litt, Sarah Lim, Martin Kleppmann, and Peter van Hardenberg. Peritext: A CRDT for collaborative rich text editing. *Proc. ACM Hum.-Comput. Interact.*, 6(CSCW2), nov 2022.

[33] Geoffrey Litt, Peter van Hardenberg, and Orion Henry. Project cambria: Translate your data with lenses. Technical report, Ink & Switch, October 2020. https://www.inkandswitch.com/cambria/.

[34] Karissa Rae McKelvey, Scott Jensen, Eileen Wagner, Blaine Cook, and Martin Kleppmann. Upwelling: Combining real-time collaboration with version control for writers, 2023. `https://www.inkandswitch.com/upwelling/`.

[35] T. Mens. A state-of-the-art survey on software merging. *IEEE Transactions on Software Engineering*, 28(5):449–462, 2002.

[36] MirageOS Project. Irmin, 2020. `https://irmin.org/`.

[37] Brice Nédelec, Pascal Molli, Achour Mostefaoui, and Emmanuel Desmontils. Lseq: An adaptive structure for sequences in distributed collaborative editing. In *Proceedings of the 2013 ACM Symposium on Document Engineering*, DocEng '13, page 37–46, New York, NY, USA, 2013. Association for Computing Machinery.

[38] Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhäuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *Proceedings of the 1996 ACM Conference on Computer Supported Cooperative Work*, CSCW '96, pages 288–297, New York, NY, USA, 1996. Association for Computing Machinery.

[39] Rocicorp. Replicache, 2023. `https://replicache.dev/`.

[40] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, Inria – Centre Paris-Rocquencourt ; INRIA, January 2011. `https://hal.inria.fr/inria-00555588`.

[41] 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*, pages 386–400, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[42] Nate Smith and Joseph Gentle. ShareDB, 2023. `https://share.github.io/sharedb/`.

[43] Sarah Sterman, Molly Jane Nicholas, and Eric Paulos. Towards creative version control. *Proc. ACM Hum.-Comput. Interact.*, 6(CSCW2), nov 2022.

[44] **Matthew Weidner**. CRDT survey, part 2: Semantic techniques. Blog post, 2022. `https://mattweidner.com/2023/09/26/crdt-survey-2.html`.

[45] **Matthew Weidner**. Designing data structures for collaborative apps. Blog post, 2023. `https://mattweidner.com/2022/02/10/collaborative-data-design.html`.

[46] **Matthew Weidner**. Proposal: Versioned collaborative documents. In *Programming Local-first Software Workshop*, PLF '23, 2023. `https://mattweidner.com/assets/pdf/versioned_collaborative_documents.pdf`.

[47] **Matthew Weidner** and Martin Kleppmann. The art of the Fugue: Minimizing interleaving in collaborative text editing, 2023. `http://arxiv.org/abs/2305.00583`.

[48] **Matthew Weidner**, Heather Miller, and Christopher Meiklejohn. Composing and decomposing op-based CRDTs with semidirect products. *Proc. ACM Program. Lang.*, 4(ICFP):1–27, August 2020.

[49] **Matthew Weidner**, Ria Pradeep, Benito Geordie, and Heather Miller. For-each operations in collaborative apps. In *Proceedings of the 10th Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '23, page 56–62, New York, NY, USA, 2023. Association for Computing Machinery.

[50] **Matthew Weidner**, Huairui Qi, Maxime Kjaer, Ria Pradeep, Benito Geordie, Yicheng Zhang, Gregory Schare, Xuan Tang, Sicheng Xing, and Heather Miller. Collabs: A flexible and performant CRDT collaboration framework, 2023. `http://arxiv.org/abs/2212.02618`.

[51] Albert van der Linde, Pedro Fouto, João Leitão, Nuno Preguiça, Santiago Castiñeira, and Annette Bieniusa. Legion: Enriching internet services with peer-to-peer interactions. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, pages 283–292, Republic and Canton of Geneva, Switzerland, 2017. International World Wide Web Conferences Steering Committee. `https://doi.org/10.1145/3038912.3052673`.

[52] Albert van der Linde, João Leitão, and Nuno Preguiça. δ-crdts: making δ-crdts delta-based. In *Proceedings of the 2nd Workshop on the Principles and Practice of Consistency for Distributed Data*, page 12. ACM, 2016.

[53] Peter van Hardenberg. The path to local-first software, 2022. Programming Local-First (PLF) workshop '22.

[54] Stephane Weiss, Pascal Urso, and Pascal Molli. Logoot: A scalable optimistic replication algorithm for collaborative editing on P2P networks. In *29th IEEE International Conference on Distributed Computing Systems*, ICDCS 2009, pages 404–412. IEEE, 2009.

[55] Elena Yanakieva, Philipp Bird, and Annette Bieniusa. A study of semantics for CRDT-based collaborative spreadsheets. In *Proceedings of the 10th Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '23, page 37–43, New York, NY, USA, 2023. Association for Computing Machinery.

[56] Pierre Étienne Meunier and Florent Becker. Pijul. `https://pijul.org/`, 2023.