Proposal: Versioned Collaborative Documents

Matthew Weidner
maweidne@andrew.cmu.edu
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA

Abstract

I propose a design for *versioned collaborative documents*. Versioned collaborative documents combine Google Docs-style real-time collaboration with git-style fork-merge collaboration. I sketch the architecture of a local-first platform for versioned collaborative documents, outline the programming model that programmers would use to support new document types on top of the platform, and discuss the tools required to implement the proposal.

1 Introduction

Local-first software [16] 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. First, it 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.

We propose *versioned collaborative documents* that overcome these shortcomings. Our proposal has 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.
3. The ability to support arbitrary document types with nuanced, type-specific merge semantics: rich text documents, spreadsheets, recipes, etc.

Put another way, our proposal generalizes Upwelling [20]—a rich-text editor with features from version control systems—to handle arbitrary document types and branching patterns.

There are two parts to our proposal. Section 2 describes the user-facing concepts involved in versioned collaborative documents and a platform to manage them. Section 3 describes the programming model that programmers use to support new document types on top of the platform.

This paper is a design document; implementing it as real software is the next step. Section 4 discusses the challenges in realizing versioned collaborative documents and how existing tools measure up.

2 Concepts and Platform

We begin by discussing the user-facing concepts involved in versioned collaborative documents. They are a blend of concepts from Google Docs and git.

For each concept, we describe the user's experience and also sketch its underlying implementation. The implementations together form a *platform* for versioned collaborative documents, consisting of independent "homeservers" and a local-first client on each user device. Both the concepts and the platform apply to arbitrary document types.

2.1 Documents

A *document* is a "unit of collaboration": something that a group of users can collaborate on in real time. Each document is identified by a UUID that we call its *docID*. A document is mutable: the content identified by a docID changes over time.

When a document is created, it is assigned several properties:

Homeserver A server chosen by the document's creator where the document lives.[1]

Location A URL on the homeserver that includes the docID. Homeservers follow a consistent protocol so that a client can connect to any document given just its location. A typical location could have the form `https://<homeserver>/<docID>`, like a hosted git repository.

Document type A type that indicates how the document is formatted, analogous to a file extension.

Collaborators' clients connect to the location to read and edit the document. The homeserver controls access to the document, i.e., who is allowed to read and edit it. These permissions are originally set by the document's creator, but the homeserver could also allow them to designate admins, transfer ownership, etc.

Concretely, a homeserver views a document as an append-only, totally-ordered log of*updates*. Each user input that changes the collaborative state generates an update describing that change: insert some text, change a spreadsheet cell, set an ingredient's amount, etc. The document's type determines how these updates are formatted. The user needs a type-specific *renderer* to view and edit the document; we discuss these renderers in Section 3.

[1]We borrow this term from Matrix [10], but use it differently: in Matrix, each *user* has a single homeserver, while in our proposal, each *document* does.

We later introduce optimizations that let clients avoid working with the entire update log (Section 3.6).

Optimistic updates. The homeserver's update log is the single source of truth for the document. Clients may optimistically apply updates to their own view of the document, but these updates are tentative until they are confirmed by the homeserver. In particular, a client always applies the local user's updates immediately, without waiting for confirmation from the homeserver: the document is local-first.

Updates can be logically concurrent (causally ordered), even though the homeserver eventually assigns them a total order. Thus we recommend using an op-based CRDT[2] [24] to generate and interpret updates. For example, a rich-text document could use the Peritext rich-text CRDT [17]. We discuss CRDTs further in Section 4.

Homeserver vs full decentralization. Using a homeserver has several advantages over fully decentralized collaboration:

- In confusing situations, the homeserver makes the final decision about which updates become part of the update log, using established techniques for server-based apps. For example, if one user edits the document concurrent to losing their edit privileges, the homeserver could choose to accept exactly the updates that it received before learning of the privilege change.
- Likewise, the homeserver decides how to handle confusing situations involving permissions, e.g., two admins who ban each other concurrently.
- Many CRDTs are vulnerable to *equivocation*: a user sends multiple messages with the same logical timestamp but different contents, putting other users in inconsistent states [15]. The homeserver could detect equivocation and only accept the first message with a given logical timestamp.

Despite our use of a homeserver, decentralized collaboration is still allowed: users can share optimistic updates without using the homeserver, e.g., via peer-to-peer connections or a backup server.

For example, two users who are offline but on the same LAN could continue collaborating. Later, the first user to come back online pushes both users' tentative updates to the homeserver. Note that the homeserver may reject some updates, e.g., if the second user lost edit access while offline; we discuss how to handle this sitation in Section 3.5.

One disadvantage of using a homeserver is that it is a single point of failure for a specific document. However, users can work around a failure by creating a fork (described later) on a different homeserver.

[2]Or an equivalent technology like decentralized Operational Transformation [23].

Document analogies: Google Docs document; git remote branch.

2.2 Versions

A *version* of a document is a snapshot of that document's state at a particular point in time. A version is immutable, and it is unambiguously and globally identified by a *version string* of the form `<docID>@<versionID>`. Any user with read access can request a version's state given its version string.

A typical version corresponds to a point in time on the homeserver: the state resulting from the first n updates in the homeserver's log, for some n. In principle, one could also expose versions that include tentative local updates, but these would require more complicated versionIDs (e.g., encoding a vector clock [9, 19]).

Version analogies: git commit; Google docs version in the version history.

2.3 Forks

At any time, a user can create a *fork* of a document version that they have read access to (the fork's *base version*). This creates a new document whose initial update log is a copy of the base version's.

The new document has a new docID, it is owned by the forking user, and it has a new location chosen by the forking user. Thus the forking user can edit and share the forked document even if they had more limited permissions on the original document. Forks, and edits to forks, have no influence on the original document, which might not even be aware of them.

Since a fork copies its base version's update log, it starts with the same version history as the base version. However, a fork is not affected by newer edits to the original document. In particular, a fork continues functioning even if the forking user loses read access to the original document.

Users may also create forks retrospectively. For example, if a user makes many edits to a document offline and wants to view the diff before updating the homeserver, the user's client could pretend that those edits were applied to a new fork of the original document. Likewise if some tentative updates are rejected by the homeserver, or if the homeserver fails.

All forks remember the docID of their *root document*: the originally created document, prior to any forks.

Fork analogies: git branching; GitHub fork; Google Docs "Make a copy" but preserving version history.

2.4 Diffs

A user can view the *diff* between any two versions that share a common root document, even if they come from different forks (i.e., they have different docIDs). A diff is unambiguously identified by a *diff string* of the form `<version string 1>..<version string 2>`. This indicates the changes that result from inverting the updates that are in version 1 but not version 2, then applying the updates that are in version 2 but not version 1.

A user can also view diffs involving "pseudo-versions" of the form `<docID>@current`, indicating the current, live-updating version of docID.

Diff analogies: git diff; Google Docs "show changes" in version history.

2.5 Merging

At any time, a version V may be *merged* into a different document D that has the same root document. This merges V's update log into D's, appending any updates that are present in V's update log but not D's.

Merging is intended for git-style branch-and-merge workflows, as follows:

1. A user creates a fork F of a document D that they have read access to.
2. That user, and possibly collaborators, edit F as a separate document, applying some coherent set of changes.
3. When finished, they propose to merge the current version V of F into D. Note that D may have updated independently in the meantime.
4. An editor of D reviews the diff that would result—the diff from D's current state to the merged state—and suggests changes to V.
5. Once the editor is happy with V (or a subsequent changed version), they merge it into D.
6. Usually, the forked document F is then deleted.

Note that F's updates are applied on top of any concurrent updates to D. It is important that F's updates have good merge semantics, i.e., they roughly preserve F's authors' intentions despite being applied to a different state. Thus we again recommend using an op-based CRDT to generate and interpret updates. Ideally, this CRDT's merge semantics are tuned to the particular document type.

Forking and merging can form arbitrary branching patterns, like in git. For example, an owner's manual could have a fork for the next product release, a fork on top of that for a new product feature, and a further fork for typo fixes. These forks would be merged back in reverse order. If the feature is pushed back to a later release, its fork could be merged into that release's document, even though it was not forked from that document.[3]

As another example, a creative practitioner could try out various ideas in different forks and keep them as a "palette of materials" [27]. If they later decide to move forward with an idea, they can merge its fork into their main document, even across long timescales.

[3]Even criss-cross merges are technically possible, but less confusing than in git: assuming that you use an op-based CRDT, merging D_1 into D_2 always gives the same state as merging D_2 into D_1.

Merging analogies: git merge commit; GitHub pull request.

3 Programming Model

A user interacts with a versioned collaborative document using a *renderer* specific to its document type. For example, a renderer for rich-text documents would center around a rich-text editor.

Any programmer can implement a new renderer, hence support a new document type.[4] This section describes the programming model for renderers.

3.1 Renderer Responsibilities

A renderer is only responsible for tasks specific to its document type: rendering the document in a GUI, updating the GUI in response to new updates, and generating new updates in response to user input.

We propose that all other tasks are handled by the platform's local-first client. In particular, the platform client (not the renderer) is responsible for UI elements that manage Section 2's concepts: commands to fork and merge documents; a display showing whether the current document is up-to-date with its homeserver; peer-to-peer networking options; etc.

One option is to make the platform client be a comprehensive app that includes the common UI elements alongside an embedded renderer. Another option is to make the client be a standalone app that interacts with renderers through local files or IPC, similar to git.[5] Either way, moving common tasks into the platform client reduces renderers' implementation difficulty and ensures a consistent user experience across document types.

The remaining subsections describe the specific tasks that a renderer must implement.

3.2 Document Editing

In normal operation, the platform client provides the renderer with a document's update log. The renderer must compute the current state from this log and render it in a GUI.

The platform client also delivers new updates as it receives them from local storage, the homeserver, or peer-to-peer connections; the renderer must update its GUI to reflect each new update.

Optimistic updates may cause the renderer to receive updates in a different order than the homeserver. In particular, the platform client will echo local updates back to the renderer immediately, possibly before concurrent updates that the homeserver orders first. The renderer must ensure that it computes the correct state regardless: the document state must depend only on the *set* of updates received, not their

[4]There can be multiple apps for a particular document type if they agree on a common update format.

[5]In fact, one can already use git as a rudimentary client: store the update log in a git-tracked text file with one update per line, and have the renderer read and write that file.

delivery order (strong eventual consistency [25]). To ensure this, we again recommend that the renderer uses an op-based CRDT to generate and interpret updates.

When the user has edit access to a document, the renderer must also allow editing. For each local edit, the renderer must give the platform client a new update describing the edit. The platform client echoes this update back to the renderer immediately (to emphasize that the document is local-first), saves it locally, and eventually distributes it to the homeserver and possibly peer-to-peer collaborators.

3.3 Diffs

To display the diff between two versions, the platform client provides the renderer with the updates in both versions, the updates in version 1 but not version 2, and the updates in version 2 but not version 1. The renderer must display these to the user in a reasonable type-specific way.

Recall from Section 2.4 that a diff can involve live-updating "pseudo-versions". In this case, the platform client delivers live updates to the renderer, which must update its GUI to reflect each new update.

Optionally, the renderer may also allow the user to make edits while viewing a diff. This is useful when reviewing the diff before merging V into D: the reviewer may make additional edits to V before approving the merge.[6] In particular, the renderer could allow selective undo operations, analogous to Google Docs' "Reject suggestion" feature.

3.4 Versions, Forks, and Merges

The renderer does not need to implement any additional functionality to support versions, forks, and merges:

Displaying a version The platform client provides the version's update log as if it were a read-only document.

Forking This does not involve document rendering.

Merging The platform client asks the renderer to display a diff between the target document's current state and the merged state. After merging, the platform client asks the renderer to display the merged document normally.

3.5 Rejected Updates

Recall that updates are tentative until accepted by the homeserver. This may lead to a confusing situation where the platform client optimistically applies an update but later learns that it has been rejected by the homeserver. In particular, this can occur if:

(1) The local user lost edit access; or
(2) The local user optimistically applied peer-to-peer updates from a user who lost edit access or performed equivocation.

[6]Technically, these edits would occur in a new fork whose starting state is the merge of V and D.

To avoid requiring additional functionality from the renderer (namely, the ability to revert updates), we propose that the platform client transparently switches to a new fork of the document that matches the renderer's state. The user may continue editing this fork, or in case (2), they could merge it back into the original document after checking for vandalism. If the renderer supports selective undo operations, it could offer to undo all rejected updates during the merge diff.

3.6 Optimization: Saved States

So far, we have defined a document solely in terms of its update log. However, there are situations when working with the update log alone is inefficient:

(1) When a user opens a document, it can take some time to apply the whole update log.
(2) When a user first downloads a document, they would prefer to see the current state quickly and download the whole update log (hence version history) in the background, if at all.
(3) When a merge occurs, the document gets many new updates all at once, which can take some time to process.

Situations (1) and (2) can be solved by saving the renderer's internal representation of a document, e.g., its op-based CRDT's state (including CRDT metadata). The renderer must provide functions that save and load its internal state. The platform client can call these to load documents in place of applying the corresponding updates.

Situation (3) requires more complex saved states that can be "merged", where merging is equivalent to taking the union of the corresponding updates. This is how state-based CRDTs behave [24], and so the renderer can use hybrid op-based/state-based CRDTs to optimize situations (1)–(3). We expect to make state-based merging optional because its benefits are modest.

4 State of Tooling

Implementing versioned collaborative documents will require a combination of techniques from version control systems, collaborative apps, and existing local-first software. We now describe the problems that we expect to be most difficult and how well existing tools address them.

Platform. The platform internals appears to be a "small matter of programming"—substantial engineering effort, but mostly straightforward. One exception is decentralized collaboration, which requires peer-to-peer networking, efficient causal ordering, and handling rejected updates. Luckily, that is optional.

Designing the platform's client UI will be challenging. It must expose all of the concepts from Section 2 in a way that is understandable to ordinary users, including forking and merging. Prior work generally trades off power and usability:

Google Docs suggestions are easy to use but limited; git is powerful but technical; Upwelling is in between [20].

Renderers. The main challenge in implementing a renderer is likely state management, i.e., processing and generating updates. Luckily, several libraries provide hybrid op-based/state-based CRDTs sufficient to implement document editing (Section 3.2) and saved states (Section 3.6):

- Yjs supports rich text, maps, and lists [13].
- Automerge supports JSON [3].
- Collabs supports various data structures and also permits custom merge semantics [30].

These can be paired with a reactive UI framework, such as React [6], Elm [7], or REScala [21], to render a live-updating document.

Existing CRDT libraries have limited support for diffs (Section 3.3). Yjs has undocumented support for "snapshots" (versions) and a demo that displays the diff between two rich-text snapshots [12]. In general, one could render a diff by delivering the differing updates to a CRDT library and recording the resulting local changes. However, this seems difficult to program against, especially when live updates are permitted. Future CRDT libraries could support native diff views, e.g., a list CRDT that tags each element with the versions containing it (version 1, version 2, or both).

Native diff views could also support rejecting changes. We believe it is sufficient to perform ordinary updates that undo a change—e.g., reject text by deleting it—but more nuanced semantics are possible [1].[7] Yjs has a selective Undo/Redo manager, but it only allows undoing in stack order, not at arbitrary points in the history.

Schema evolution is another difficult problem. The format of updates, and the renderer code generally, could change over the life of a document. Cambria [18] uses "bidirectional lenses" to address this problem, which translate updates between schemas. It remains to be seen whether the centralized homeserver makes schema evolution easier, e.g., by enforcing a linear sequence of schema upgrades (though this can break down during forking and merging).

5 Related Work

Versioned collaborative documents synthesize ideas from a number of sources. We have already mentioned links to Google Docs, git, and GitHub.

Upwelling [20] 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, with explicit change tracking (diffs). Versioned collaborative documents extend Upwelling's concepts to arbitrary document types. Also, we allow more general branching patterns: forked documents can themselves be forked, and we do not "rebase" forks on top of their merge target until the user requests to review a diff. Note that Upwelling already includes what we would call a renderer for rich-text documents, including live-updating diffs.

PushPin [28] describes a programming model for local-first collaborative documents that inspired our renderers. Specifically, it uses functional reactive programming to render and update CRDT documents, with each document identified by a URL. Unlike versioned collaborative documents, PushPin does not consider forking and merging, and it uses purely peer-to-peer networking instead of a homeserver.

Matrix-CRDT [8] and Automerge Repo [4] are document stores for CRDT-based documents. They provide storage and collaborative networking so that an individual collaborative app mostly only needs to implement rendering. They do not support forking and merging, and documents do not have a central homeserver.

Several works explore how to manage permissions for fully decentralized collaboration, including Matrix's authorization rules [11], DCGKA [29], and @localfirst/auth [2]. We believe that the homeserver's centralized source of truth makes their complex techniques unnecessary. It comes at the cost of introducing a single point of failure, which we mitigate using forks instead of full decentralization.

Irmin [5] and Pijul [32] are distributed version control systems that involve CRDTs. Irmin implements a key-value store that one can build CRDTs on top of [14, 26], while Pijul implements a specific CRDT for line-based text files. Unlike versioned collaborative documents, these systems do not support arbitrary CRDT updates, and we are not aware of real-time collaboration support.

6 Future Work

For future work, we would like to refine this proposal and eventually implement it. In particular, we aim to implement the proposed local-first platform and provide tools for writing renderers. These will hopefully let non-expert programmers create novel local-first apps.

Besides the implementation challenges from Section 4, future work could investigate extensions to this proposal:

Commit messages. A user may wish to provide a "commit message" summarizing a group of updates and its rationale [22]. The platform should allow adding a commit message to a group of updates (e.g., a merge) and display these messages in a history view. The renderer could also display these messages in a "git blame" view.

History hiding. Versioned collaborative documents retain their full version history. Users may instead wish to hide deleted content from future collaborators, or they may need

[7] In principle, one could undo an update by removing it from the update log directly. In practice, this might erase metadata needed by future updates, e.g., info used by a text CRDT to sort other characters.

to delete leaked secrets from the history. In particular, when merging, users may wish to perform a "squash merge" that omits irrelevant updates.

Cherry-picking. A user may wish to merge only a cherry-picked subset of changes from a fork—e.g., all changes to a specific paragraph. The platform can implement cherry-picking by appending only the desired updates to the log, but it seems challenging to design user interfaces and CRDTs that can support this.

Yu, Oster, and Ignat describe a CRDT-based Emacs extension with a similar feature [31].

Beyond CRDTs. Op-based CRDTs are designed for a fully decentralized setting, but our documents have a centralized source of truth on the homeserver. Can we leverage this to allow merge semantics beyond CRDTs? For example, no existing CRDT supports a find-and-replace-text operation, but it seems possible to perform such an operation in a fork and then run it again just after merging.

Subdocuments. Some documents contain sections with different permissions, e.g., body text vs comments. We can model each section as a separate document with its own permissions, but forking and merging these subdocuments together seems complicated.

Partially decentralized permissions. The homeserver enforces permissions in a fully centralized fashion. We could instead let clients enforce basic permissions on their own in a decentralized fashion, while still using the homeserver as a fallback in confusing situations.

Acknowledgments

I thank Florian Jacob, Gregory Schare, and the PLF workshop reviewers for providing feedback on drafts of this proposal. I was supported by an NDSEG Fellowship sponsored by the US Office of Naval Research.

References

[1] Eric Brattli and Weihai Yu. 2021. Supporting Undo and Redo for Replicated Registers in Collaborative Applications. In *Cooperative Design, Visualization, and Engineering*, Yuhua Luo (Ed.). Springer International Publishing, Cham, 195–205.

[2] Herb Caudill. 2023. @localfirst/auth. GitHub repository. https://github.com/local-first-web/auth

[3] Automerge contributors. 2023. Automerge. GitHub repository. https://github.com/automerge/automerge

[4] Automerge contributors. 2023. Automerge Repo. GitHub repository. https://github.com/automerge/automerge-repo

[5] Irmin contributors. 2023. Irmin. GitHub repository. https://github.com/mirage/irmin

[6] React contributors. 2023. React. GitHub repository. https://github.com/facebook/react

[7] Evan Czaplicki. 2021. Elm. https://elm-lang.org/

[8] Yousef El-Dardiry. 2023. Matrix CRDT. GitHub repository. https://github.com/YousefED/Matrix-CRDT

[9] Colin J. Fidge. 1988. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference* 10, 1 (1988), 56–66.

[10] Matrix.org Foundation. 2023. Matrix. https://matrix.org/

[11] Florian Jacob, Luca Becker, Jan Grashöfer, and Hannes Hartenstein. 2020. Matrix Decomposition: Analysis of an Access Control Approach on Transaction-Based DAGs without Finality. In *Proceedings of the 25th ACM Symposium on Access Control Models and Technologies* (Barcelona, Spain) *(SACMAT '20)*. Association for Computing Machinery, New York, NY, USA, 81–92. https://doi.org/10.1145/3381991.3395399

[12] Kevin Jahns. 2022. ProseMirror + Versions Demo. GitHub repository. https://github.com/yjs/yjs-demos/tree/main/prosemirror-versions

[13] Kevin Jahns. 2023. Yjs. GitHub repository. https://github.com/yjs/yjs

[14] Gowtham Kaki, Swarn Priya, KC Sivaramakrishnan, and Suresh Jagannathan. 2019. Mergeable Replicated Data Types. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 154 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360580

[15] Martin Kleppmann and Heidi Howard. 2020. Byzantine Eventual Consistency and the Fundamental Limits of Peer-to-Peer Databases. *CoRR* abs/2012.00472 (2020). arXiv:2012.00472 https://arxiv.org/abs/2012.00472

[16] Martin Kleppmann, Adam Wiggins, Peter van Hardenberg, and Mark McGranaghan. 2019. 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* (Athens, Greece) *(Onward! 2019)*. Association for Computing Machinery, New York, NY, USA, 154–178. https://doi.org/10.1145/3359591.3359737

[17] Geoffrey Litt, Sarah Lim, Martin Kleppmann, and Peter van Hardenberg. 2022. Peritext: A CRDT for Collaborative Rich Text Editing. *Proc. ACM Hum.-Comput. Interact.* 6, CSCW2, Article 531 (nov 2022), 36 pages. https://doi.org/10.1145/3555644

[18] Geoffrey Litt, Peter van Hardenberg, and Orion Henry. 2020. Project Cambria: Translate your data with lenses. https://www.inkandswitch.com/cambria/

[19] Friedemann Mattern. 1989. Virtual Time and Global States of Distributed Systems. In *Parallel and Distributed Algorithms*. North-Holland, 215–226.

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

[21] Ragnar Mogk, Lars Baumgärtner, Guido Salvaneschi, Bernd Freisleben, and Mira Mezini. 2018. Fault-tolerant Distributed Reactive Programming. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 109)*, Todd Millstein (Ed.). Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 1:1–1:26. https://doi.org/10.4230/LIPIcs.ECOOP.2018.1

[22] So Yeon Park and Sang Won Lee. 2023. Why "why"? The Importance of Communicating Rationales for Edits in Collaborative Writing. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems* (Hamburg, Germany) *(CHI '23)*. Association for Computing Machinery, New York, NY, USA, Article 616, 25 pages. https://doi.org/10.1145/3544548.3581345

[23] Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhäuser. 1996. 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* (Boston, Massachusetts, USA) *(CSCW '96)*. Association for Computing Machinery, New York, NY, USA, 288–297. https://doi.org/10.1145/240080.240305

[24] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A comprehensive study of Convergent and Commutative Replicated Data Types*. Research Report RR-7506. Inria – Centre Paris-Rocquencourt ; INRIA. 50 pages. https://hal.inria.fr/inria-00555588

[25] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems*, Xavier Défago, Franck Petit, and Vincent Villain (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 386–400.

[26] Vimala Soundarapandian, Adharsh Kamath, Kartik Nagar, and KC Sivaramakrishnan. 2022. Certified Mergeable Replicated Data Types. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 332–347. https://doi.org/10.1145/3519939.3523735

[27] Sarah Sterman, Molly Jane Nicholas, and Eric Paulos. 2022. Towards Creative Version Control. *Proc. ACM Hum.-Comput. Interact.* 6, CSCW2, Article 336 (nov 2022), 25 pages. https://doi.org/10.1145/3555756

[28] Peter van Hardenberg and Martin Kleppmann. 2020. PushPin: Towards Production-Quality Peer-to-Peer Collaboration. In *Proceedings of the 7th Workshop on Principles and Practice of Consistency for Distributed Data* (Heraklion, Greece) *(PaPoC '20)*. Association for Computing Machinery, New York, NY, USA, Article 10, 10 pages. https://doi.org/10.1145/3380787.3393683

[29] Matthew Weidner, Martin Kleppmann, Daniel Hugenroth, and Alastair R. Beresford. 2021. Key Agreement for Decentralized Secure Group Messaging with Strong Security Guarantees. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 2024–2045. https://doi.org/10.1145/3460120.3484542

[30] Matthew Weidner, Heather Miller, Huairui Qi, Maxime Kjaer, Ria Pradeep, Ignacio Maronna, Benito Geordie, and Yicheng Zhang. 2023. Collabs. GitHub repository. https://github.com/composablesys/collabs

[31] Weihai Yu, Gérald Oster, and Claudia-Lavinia Ignat. 2017. Handling Disturbance and Awareness of Concurrent Updates in a Collaborative Editor. In *Cooperative Design, Visualization, and Engineering*, Yuhua Luo (Ed.). Springer International Publishing, Cham, 39–47.

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