Undo and Redo Support for Replicated Registers

Leo Stewen
lstwn@mailbox.org
Technical University of Munich
Germany

Martin Kleppmann
martin@kleppmann.com
University of Cambridge
United Kingdom

Abstract

Undo and redo functionality is ubiquitous in collaboration software. In single user settings, undo and redo are well understood. However, when multiple users edit a document, concurrency may arise, leading to a non-linear operation history. This renders undo and redo more complex both in terms of their semantics and implementation.

We survey the undo and redo semantics of current mainstream collaboration software and derive principles for undo and redo behavior in a collaborative setting. We then apply these principles to a simple CRDT, the Multi-Valued Replicated Register, and present a novel undo and redo algorithm that implements the undo and redo semantics that we believe are most consistent with users' expectations.

CCS Concepts: • **Human-centered computing** → **Collaborative and social computing systems and tools**; • **Computing methodologies** → *Distributed algorithms*; • **Information systems** → *Remote replication*; Data structures.

Keywords: undo, redo, CRDT, eventual consistency, collaborative editing

1 Introduction

Collaborative editing is an essential feature in modern software. Conflict-Free Replicated Data Types (CRDTs) [9] are gaining interest because they enable collaboration in local-first software [5], that works with, but does not require the cloud to function. They allow for concurrent editing of a shared document without requiring central coordination while still guaranteeing strong eventual consistency [11].

As much as collaboration is a common feature today, so is undo and redo support. Integrating it with CRDTs is not trivial. Although there are some published algorithms for undo and redo in CRDTs (see Section 7), their semantics do not match the behavior of current mainstream applications as detailed in Section 2. We believe the semantics of mainstream applications are more in line with user expectations for an undo feature, and therefore we propose an undo/redo algorithm for CRDTs that follows this mainstream behavior. It works on top of a Multi-Valued Replicated Register (MVR) [11], a core CRDT that is often used as a basic building block to build more complex CRDTs via composition. The intended environment for this algorithm is the Automerge [13] library, which is a JSON CRDT that is based on MVRs.

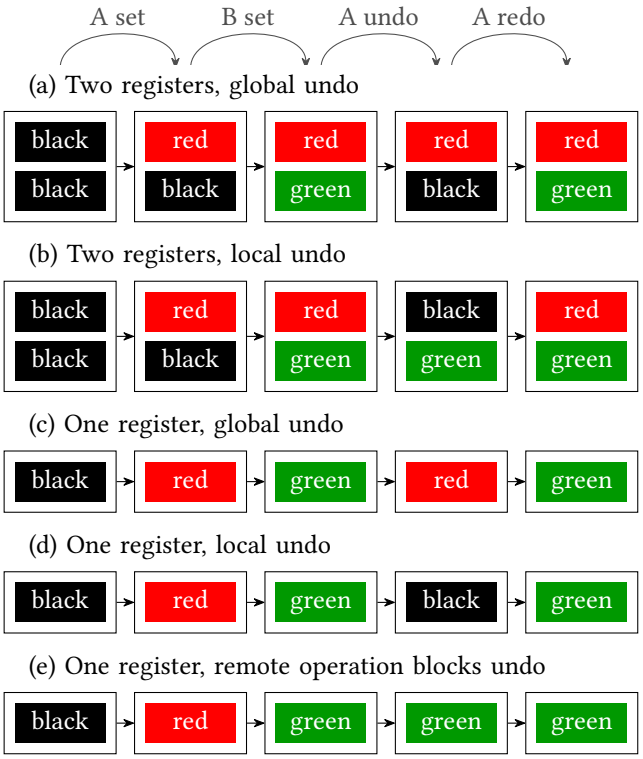

Figure 1. Different semantics of undo and redo with two users *A* and *B* collaboratively editing one (two) register(s).

2 Semantics of Undo/Redo with Multiple Users

To illustrate the possible semantics of undo and redo with multiple users, we consider replicated registers that are used to store the fill color of a rectangle, for instance, on a slide of a presentation deck. Figure 1 illustrates possible behaviors of undo and redo. Cases (a) and (b) deal with an application with two registers (rectangles) and Cases (c) to (e) show an application with one register. We assume that all operations are instantly synced to the other peer and the operations run sequentially, disregarding concurrency for now. We distinguish between the following undo behaviors:

Global undo. In Figure 1 (a), user *A* changes the upper rectangle's color to red, then user *B* sets the lower rectangle's color to green. When *A* subsequently performs an undo, it undoes the most recent operation *by any user*, i.e., it undoes *B*'s operation by setting the lower rectangle back to black. *A*'s redo operation restores the lower rectangle to green.

Case (c) shows the analogous scenario in which user A and B update the same rectangle. When A performs an undo, it undoes the most recent operation again *by any user*, changing the rectangle from green back to red.

Local undo. In Figure 1 (a), user A changes the upper rectangle to red, then user B sets the lower rectangle to green. Afterwards, user A performs an undo, which undoes the most recent operation *by A itself*, i.e., changing the upper rectangle back to black. A's subsequent redo restores the upper rectangle back to red.

In Case (d), there is only one rectangle. First, user A changes it to red, then user B changes it to green. Subsequently, A performs an undo, and like in the two-register case, this restores the value of the register to the state before the most recent operation *by A itself*, i.e., changing it back to black. When then A performs its redo, the state prior to the undo is restored, i.e., green.

Remote operation blocks undo. In Figure 1 (e), if user A tries to perform an undo in a state where the most recent operation is by a different user, undo is disabled for A.

Local undo restores to the state prior to the last operation performed by the same user who issues the undo. In contrast, *global undo* restores to the state prior the last operation performed by any user. However, a redo always restores to the state prior to its corresponding undo.

We tested the behavior of undo and redo in Google Sheets, Google Slides, Microsoft Excel Online, Microsoft PowerPoint Online, Figma and Miro Boards. In spreadsheets (Google Sheets and Microsoft Excel Online) we test a register by setting the value of a single cell instead of recoloring a rectangle. All applications we tested exhibit local undo behavior as shown in Figure 1 (b) and (d), except for Miro Boards, which blocks undo every time an operation by another user on the same register is received, as illustrated in Figure 1 (e). The most common implementations of multi-user undo for a register can be characterized by the following principles:

- **Local undo**: An undo by a user undoes her own last operation on the register, thereby possibly also undoing the effects of operations by other users, if they lie temporally in between her last operation and the time of issuing the undo.
- **Undo Redo Neutrality** [15]: Assume a register is in state s. A sequence of n undo operations followed by a sequence of n redo operations should restore state s.

The last principle captures a common usage pattern of undo and redo. Often users want to look at a past version of a document and then restore to the most recent version without changing anything.

We think that most applications settled for local undo behavior because global undo implicitly assumes users being aware of remote changes, too, while editing a document, and this is a fairly strong assumption to make. Figure 1 (a) demonstrates this issue, assuming that user A has only the upper but not the lower rectangle in her view, and vice-versa for user B. With global undo, A may be surprised to see no immediate effect of her undo because it reverted B's green coloring on a different page of the document. B may be surprised to learn that her last change was undone for no obvious reason. With more users editing collaboratively, the problem exacerbates, as the share of remote operations tends to increase. Hence, we only want to assume that users are aware of their own changes. Since we think that users favor predictable and consistent undo semantics, undo behavior with one register should follow local undo behavior as well.

Furthermore, global undo in a CRDT context suffers from additional problems. It is possible that multiple users concurrently undo or redo overlapping subsets of operations, leading to potentially confusing states. Local undo does not suffer from this problem, since users can only undo their own operations. Finally, global undo presumes that users agree on a total order of operations. If the total order is determined by (logical) timestamps on operations, this order cannot be final. Then, it is possible that a user undoes the operations it knows in some timestamp interval $[t_1, t_2]$, and subsequently receives a remote operation with a timestamp t_{op} that falls within the interval ($t_1 < t_{op} < t_2$). It is unclear how an algorithm should handle this situation without leading to further unexpected states. The fact that most non-CRDT software chooses local undo over global undo, despite not being affected by this problem, makes another case for following their lead.

3 Background on MVRs

A Multi-Valued Replicated Register (MVR) [11] is a data type from the family of CRDTs [9] that can be assigned a value, overwriting any previous value. When multiple values are concurrently assigned, the MVR retains all values that have not been overwritten. All values currently held by a register are called its *siblings* and we require that all replicas return siblings in the same order, e.g., sorted by a logical timestamp. For undo and redo behavior, an undo (or a redo) operation may reintroduce multiple siblings that have been overwritten in the past. We assume an operation-based CRDT model [2].

Replicas store their own copy of the MVR and can perform updates on it, generating an operation which the replica can apply locally, as well as broadcast to other replicas, which in turn apply it to their own copy of the MVR. We call an operation *local to a replica* if it originated from that same replica. Otherwise, we call it *remote*.

Each operation has a unique identifier (a logical timestamp) which imposes a total order over all operations. This order is a linear extension of the happens-before relation [6]. We call this identifier the *Operation Id (OpId)*. An example of such an identifier is pair of a local counter and a unique identifier for each replica. We write op_3^A to denote an operation by replica A with a local counter value of 3. Whenever a new

operation is generated, its counter value is set to one plus the greatest counter value of any operation known to the generating replica. This essentially yields a Lamport clock [6].

When multiple replicas concurrently update the register, the operation history becomes non-linear. We model an operation history as a directed acyclic graph where each node represents an operation and each edge from node op_2 to node op_1 represents a causal dependency of op_2 on op_1, that is, if op_2 overwrites the register value previously assigned by op_1. We call op_1 a *predecessor* of op_2 and op_2 a *successor* of op_1. Furthermore, we call op_1 an *ancestor* of op_2 and op_2 a *descendant* of op_1 if there exists a directed path from op_2 to op_1. We call op_1 and op_2 *concurrent* if neither is an ancestor of the other. Operations without any successors are called *heads* and operations without any predecessors are called *roots*. An operation may have multiple predecessors and multiple successors. Whenever an operation has multiple predecessors, we call it a *merge* operation.

Operations can be delivered in any order and/or multiple times because a replica can ensure idempotence by keeping track of the applied operations through their OpIds. Each replica buffers operations it receives and delivers them once they are causally ready, i.e., once all ancestors of an operation have been delivered.

4 Generating Operations for Undo/Redo

All operations carry an OpId and its set of predecessor OpIds, that represent their causal dependencies. The values produced by these predecessors are the ones that the operation overwrites. Next to the *SetOp*, which sets the MVR to the supplied value, we introduce a second type of operation that we call *RestoreOp*. The payload of this operation (in addition to its OpId and predecessors) is the OpId of an ancestor operation which we call its *anchor* operation. The effect of this operation is to restore the state of the register to the state prior to the anchor operation by searching through the operation history. *RestoreOp*s are used to implement both undo and redo.

Moreover, we require additional state besides the operation history. We introduce two stacks, one for undo and one for redo, with the usual last-in-first-out semantics. Both stacks exclusively contain operations generated on the local replica and ignore remote operations.

When a local *SetOp* is generated, it is pushed onto the undo stack and the redo stack is cleared, which causes the loss of the ability to redo after some sequence of undo operations followed by a *SetOp*. This behavior is in line with what most mainstream software does and makes the undo and redo semantics easier to comprehend. Clearing the redo stack ensures that redo has a clear next choice of what to redo. Without clearing the stack, redo (and undo) would become a tree to navigate: after a sequence of undo operations followed by a *SetOp* which is undone again, a subsequent redo would have the choice of redoing either the previous undo operation or the later *SetOp*. This complexity also exists in a single-user setting. The vim text editor supports undo trees[1] but it is unusual in this regard. To avoid the user-facing complexity caused by undo trees, we follow the mainstream approach of redo stack clearing.

Whenever a user performs an undo, we generate a *RestoreOp* that references the *SetOp* on top of the undo stack as its anchor and pop it off the undo stack. The generated *RestoreOp* is pushed onto the redo stack to allow it to be redone later. Whenever a user performs a redo, we generate a *RestoreOp* that references the *RestoreOp* on top of the redo stack as its anchor and pop it off the redo stack. Its anchor is resolved to a *SetOp* and pushed onto the undo stack to allow it to be undone later for another time. For redo operations, its *RestoreOp*'s anchor is another *RestoreOp*, hence requiring the algorithm to follow one indirection to resolve to a *SetOp*.

This algorithm ensures that the undo stack contains only *SetOp*s and the redo stack contains only *RestoreOp*s. That basically renders a redo as an "undo of an undo".

5 Applying Operations

Whenever a replica receives a remote operation, it buffers it until all its ancestors have been processed, before adding the new operation to its operation history. We now introduce the algorithm that determines the current value(s) of the register given an operation history containing *SetOp*s and *RestoreOp*s. We split the discussion into three smaller units: First, Algorithm 1 resolves the heads of the operation history to some *terminal heads* (defined below). Second, Algorithm 2 maps the resulting terminal heads to the value(s) of the register. To sort the values in the register appropriately, Algorithm 3 defines a comparison function for the terminal heads that Algorithm 2 utilizes.

Algorithm 1 Resolve Heads to Terminal Heads

```
1:  function resolveHeads(heads)
2:      todo ← ([head, ()] for each head ∈ heads)
3:      termHeads ← ()
4:      while todo is not empty do
5:          [nextOp, opIdTrace] ← todo.shift()
6:          opIdTrace ← (... opIdTrace, nextOp.opId)
7:          if isSetOp(nextOp) then
8:              termHeads.push([nextOp, opIdTrace])
9:          else                          ▷ nextOp is a RestoreOp
10:             anchor ← nextOp.anchor
11:             for each pred ∈ anchor.predecessors do
12:                 todo.push([pred, opIdTrace])
13:     return getValues(termHeads)
```

Algorithm 1 takes the current set of heads of the operation history and returns a list of (*SetOp*, *OpIdTrace*) pairs which

[1] https://vimhelp.org/undo.txt.html#undo-branches

we refer to as *terminal heads*. The *OpIdTrace* is a list of OpIds from the operations that have been visited along the path from a head to a *SetOp*. We track the *OpIdTrace* so that the register's values contributed by *RestoreOp*s can be sorted by the OpIds of the undo/redo operations that restore these values, rather than the OpIds of the *SetOp*s that originally assigned the values to the register. The *shift()* method on a list pops the first element off the list and returns it. The *push()* method on a list appends an element to the list.

The algorithm proceeds as follows: If a head is a *SetOp*, it is a terminal head and added to the terminal heads list together with the *OpIdTrace* that contains only the OpId of the *SetOp*. If a head is a *RestoreOp*, the algorithm traverses the operation history by considering its anchor's predecessors iteratively until *SetOp*s are encountered. While traversing the operation history, every encountered OpId is added to the *OpIdTrace* which is passed along to the next iteration. When the search stops at a *SetOp*, the *SetOp* and the *OpIdTrace* are added to the terminal heads list as a pair. The *OpIdTrace*'s last element is always an OpId from a *SetOp* and any preceding element is an OpId from a *RestoreOp*. The processing order of heads and predecessors in the *todo* list is not relevant for correctness as every terminal head is sorted later in Algorithms 2 and 3.

The reason for handling *SetOp*s and *RestoreOp*s differently is that a *RestoreOp* may contribute multiple values to the register due to concurrency in the operation history. For instance, if at any iteration step a merge *RestoreOp* having k predecessors is processed, all k predecessors are considered by the algorithm and they may produce k or more siblings.

Algorithm 2 Resolve Terminal Heads to Value(s)

```
1: function getValues(termHeads)
2:     sortedHeads ← sort(termHeads) desc using Alg. 3
3:     values ← ()
4:     for each head in sortedHeads do
5:         headOp ← head[0]
6:         if headOp.value ≠ None then
7:             values.push(headOp.value)
8:     return values
```

Algorithm 2 determines the value(s) of the register given a list of terminal heads consisting of pairs of *SetOp*s and *OpIdTrace*s. On line 2, it sorts the terminal heads by their *OpIdTrace* in descending order using Algorithm 3. Then, it filter maps the sorted terminal heads to their value(s) in lines 3-7. In case of a deletion (*SetOp* with no value supplied), the terminal head is skipped and no value is produced.

Algorithm 3 provides the comparison logic of two *OpIdTrace*s when sorting the list of terminal heads in line 2 of Algorithm 2. The *zip()* function takes two lists and returns a list of pairs where the first element of the pair is from the first list and the second element is from the second list. If the lists are of different lengths, the longer list is truncated to

Algorithm 3 Comparison Function

```
1: function compare(aOpIdTrace, bOpIdTrace)
2:     zipped ← zip(aOpIdTrace, bOpIdTrace)
3:     for each (aOpId, bOpId) in zipped do
4:         if aOpId < bOpId then
5:             return Less
6:         else if aOpId > bOpId then
7:             return Greater
```

the length of the shorter list. The iteration in lines 3-7 always terminates before finishing with its last iteration because of the nature of the *OpIdTrace*s. If two terminal heads stem from different heads they already differ in their first element (the OpId of the head). If two terminal heads stem from the same head (which then must be a *RestoreOp*), they might share a common path but their paths eventually diverge for some elements. The *zip()* function's truncation does not harm this property as an *OpIdTrace*'s last element is always an OpId from a *SetOp* which cannot occur on the same position of another, *longer OpIdTrace*.

Example. Figure 2 shows an operation history that is replicated between two replicas A and B that contains undo and redo operations. Table 1 shows the internal node states during the time steps from Figure 2. It demonstrates the stack maintenance introduced in Section 4 and shows the register's values produced by Algorithms 1 to 3 which are applied on top of the respective operation histories at the time steps. Steps (2a) and (2b) point out how two concurrent undo operations are resolved. $undo_5^A(\binom{A}{3})$ produces the value 2 with an *OpIdTrace* of $[\binom{A}{5},\binom{B}{2}]$. $undo_5^B(\binom{B}{4})$ produces the values 3 with an *OpIdTrace* of $[\binom{B}{5},\binom{B}{3}]$ and 4 with an *OpIdTrace* of $[\binom{B}{5},\binom{A}{3}]$. Sorting by these *OpIdTrace*s yields the register's values [3,4,2] after syncing in (2b).

The concurrent $undo_7^B(\binom{B}{2})$ and $set_7^A(6)$ operations at (4) highlight the necessity of the *OpIdTrace*. The $undo_7^B(\binom{B}{2})$ operation produces the value 1 with an *OpIdTrace* of $[\binom{B}{7},\binom{A}{1}]$. The $set_7^A(6)$ operation immediately yields the value 6 with an *OpIdTrace* of $[\binom{A}{7}]$. If the values were sorted by taking only their *SetOp*s into account, the values from a *RestoreOp* would always be sorted after the values from concurrent *SetOp*s (yielding $[6, 1]$ in the example). To fix this unfairness, the *OpIdTrace* sorts values produced by *different heads* by their heads' OpIds and values produced by the *same head* by their first deviation in their *OpIdTrace*s (resulting in $[1, 6]$ here).

Finally, at (4) A is giving up its ability to redo because its $set_7^A(6)$ operation clears the redo stack, rendering A unable to redo its previously undone $set_3^A(4)$ operation.

Optimization. Note that the predecessors of an operation and the anchor of a *RestoreOp* do not change as further operations are applied. Hence, we can cache the result of the traversal from a given head to improve the runtime at the cost of an increased memory overhead. For each *RestoreOp*

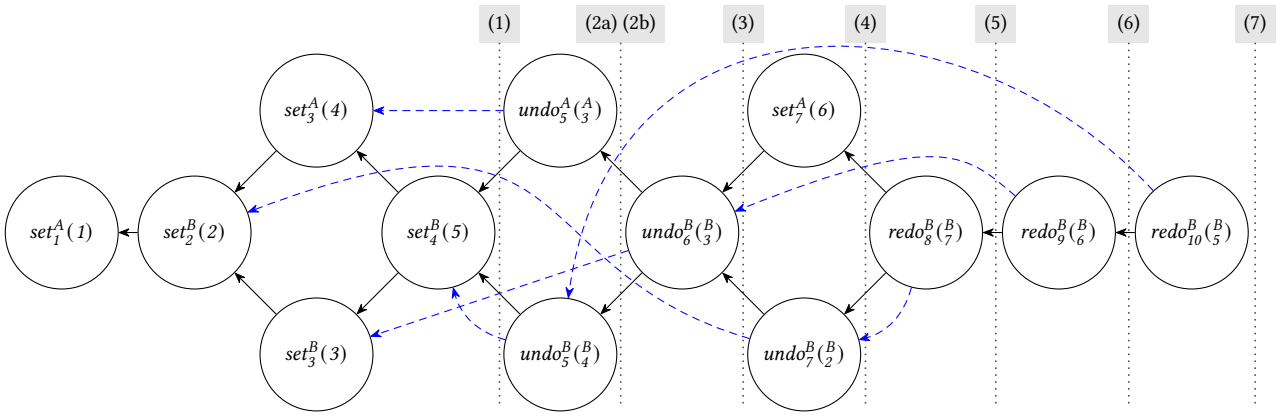

Figure 2. An operation history which is replicated between two replicas *A* and *B* with undo and redo operations. For clarity, undo and redo operations are not labeled as *RestoreOp*s. Their respective anchor operations are indicated by the dashed blue arrows and denoted in parentheses. The dotted lines indicate time steps during which we show the nodes' states in Table 1.

Time Step	(1)	(2a)	(2b)	(3)	(4)	(5)	(6)	(7)
$undoStack^A$	$[s_1^A(1),s_3^A(4)]$	$[s_1^A(1)]$	(2a)	(2a)	$[s_1^A(1),s_7^A(6)]$	(4)	(4)	(4)
$redoStack^A$	[]	$[rs_5^A(_3^A)]$	(2a)	(2a)	[]	(4)	(4)	(4)
$register^A$	[5]	[2]	[3,4,2]	[2]	[1,6]	[2]	[3,4,2]	[5]
$undoStack^B$	$[s_2^B(2),s_3^B(3),s_4^B(5)]$	$[s_2^B(2),s_3^B(3)]$	(2a)	$[s_2^B(2)]$	[]	(3)	(2a)	(1)
$redoStack^B$	[]	$[rs_5^B(_4^B)]$	(2a)	$[rs_5^B(_4^B),rs_6^B(_3^B)]$	$[rs_5^B(_4^B),rs_6^B(_3^B),rs_7^B(_2^B)]$	(3)	(2a)	(1)
$register^B$	[5]	[3,4]	[3,4,2]	[2]	[1,6]	[2]	[3,4,2]	[5]

Table 1. The internal states of the nodes' undo and redo stacks and the values of the register at the time steps from Figure 2. A set operation is abbreviated by *s* and a restore operation by *rs* with its anchor operation denoted within the parentheses. In case a cell contains a reference to another time step, the value is exactly the same as in that respective step. The difference between (2a) and (2b) is that in (2a) the replicas have not yet synced, whereas in (2b) they have exchanged their operations. At all other time steps the operations are synced.

we cache the values produced by it in sorted order. Then, we can additionally drop the values' *OpIdTrace*s and rely on a *stable* sort algorithm to include the sorted values of a *RestoreOp* at the right position among the siblings contributed by other heads, as the cached values' sorting w.r.t. the other siblings is sufficiently determined by the *OpIdTrace* starting from a head until the cached *RestoreOp*.

6 Evaluation

Correctness. Two replicas that have received the same set of operations converge to the same state because our algorithm is deterministic, traverses a finite and cycle-free predecessor relationship, and does not depend on the order in which operations were received.

Our implementation of the algorithm is written in Typescript and consists of around 350 lines of code (excluding comments and blank lines), implementing a MVR with undo and redo. We test it against a suite of unit tests that covers important scenarios, both in single-user and multi-user cases. There are 30 unit tests, comprising over 600 lines of test code.

Performance. The runtime cost of our algorithm is determined by the graph traversal that resolves *RestoreOp*s at the heads into the corresponding register values. In case of a *SetOp* at the head, the runtime of Algorithm 1 is constant as the while loop does not add an element, but only removes the *SetOp* from the *todo* list. With caching, the traversal in case of *RestoreOp*s is constant, too, because whenever an anchor's predecessor is a *RestoreOp*, its values are already cached in sorted order, rendering a deeper traversal redundant. The cost of sorting in Algorithm 2 is negligible, as we expect the number of heads to be small for a single register,

Integration with Automerge. Our implementation is currently a standalone prototype, but we intend to integrate our algorithm into the Automerge CRDT library in the future. Since Automerge stores a monotonically growing set of operations, efficient compression of the serialized operation set

is an important property. Our approach models both undo and redo with a single operation type whose extra payload consists only of a single OpId. Since every undo and redo creates a new *RestoreOp* that is added to the set, the undo and redo stacks are implicitly stored within the operation set and can be reconstructed upon loading the document to support undo and redo across sessions without additional persistency overhead.

If an application developer does not require undo and redo functionality, the value resolution algorithm will be the same as without it, making users of the library only pay for what they use. In terms of space complexity, the additional overhead are the undo and redo stacks which can be disabled if undo is not required or bounded if arbitrary undo depth support is not wanted.

Finally, we remark a possibility to prune the history under a special circumstance. Imagine a user undoing a few times and then redoing just as many times to see the document in the past and then returning to the original state. We call such a sequence of undo and redo effect-free. Provided that all replicas have seen the same operations and made no changes to the register within the effect-free sequence, the sequence can be pruned (garbage collected) from the history. However, before doing so causal stability [2] must be reached, that is, all replicas must have sent an operation whose predecessor set contains the highest OpId of the effect-free sequence.

7 Related Work

The terms local and global undo were introduced by the Operational Transformation literature in the context of collaborative text editing [1, 10, 12], but this distinction is less established in the context of CRDTs.

A common approach to undo and redo in the CRDT literature is to attach a counter (sometimes called *degree* [16] or *undo length* [3, 18]) to each operation. Some approaches [7, 16] initialize the counter to 1 and decrement it on each undo and increment it on each redo. If the counter is positive, the operation is visible, otherwise it is invisible. Other approaches [3, 18] initialize the counter to 0 and increment it on each undo and redo. If it is even, the operation is visible, otherwise it is not. When merging conflicting counters of an operation, the maximum is used. Yet, counter-based approaches handle undo only for a single operation and do not allow skipping over remote operations, which is required for local undo as outlined in Section 2. Figure 3 resembles the single-register scenario from Figure 1 but instead of A redoing in the last step, B issues an undo. It shows the correct outcomes with local undo. Applying counter-based approaches here poses some challenges. A's undo would render both A's and B's *SetOp* invisible to achieve the black coloring. B's subsequent undo requires to turn A's *SetOp* visible again to obtain the red coloring. To achieve this, B's *undo* must cause the effect of a *redo* of A's *SetOp*. Due to these

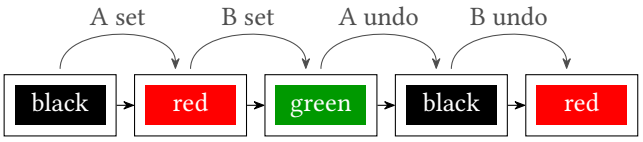

Figure 3. A difficult scenario for counter-based approaches.

counterintuitive semantics, we believe that counter-based approaches are not suitable for implementing local undo.

Yu et al. [17] deal with selective undo in the context of strings. Selective undo is a form of undo which allows a user to undo any operation, regardless of *where and when* it was generated. In contrast, global undo allows undoing a remote operation as well but only in reverse chronological order. A later work of Yu et al. [18] describes a generic undo mechanism, but it only applies to state-based CRDTs, assumes global undo behavior, and focusses on handling concurrent undo and redo of a single operation. It does not address the issue of (totally) ordering operations on the undo stack with global undo and a CRDT context.

Theoretical work by Dolan [4] demonstrates that any undoable CRDT is composed of just counters, contradicting our results at first glance. Their definition of undoable demands that an undo restores the CRDT to a previous state. We circumvent this by always advancing our internal state and only restoring the external state. Martin et al. [7] discuss undo in the context of XML-like documents. To the best of our knowledge, Brattli et al. [3] is the only work that also discusses undo and redo in the context of replicated registers, but they assume global undo behavior.

Yjs [8, 14] is another popular CRDT library with undo support[2]. Since it uses Last-Write-Wins Registers instead of MVRs, concurrent updates are lost. Consequently, an undo in Yjs does not recover siblings unlike our approach. Per default, Yjs implements global undo. While it allows the changes by a selected replica to be undone by filtering for "origins", it cannot be used for local undo, as the ability to undo is blocked upon receiving a change from a non-filtered origin[3]. Yjs's implementation uses inverse operations instead of pointers into the operation history.

8 Conclusion

We tested the semantics of undo and redo of existing collaboration software, and found that almost all use local undo. We then presented a novel algorithm for local undo on an MVR by traversing operation histories represented as directed acyclic graphs. We believe that our algorithm can also be generalized from a single MVR to more complex CRDTs, e.g., by treating every key in a map and every element in a list as a MVR, and by using shared undo and redo stacks for all MVRs in the entire data structure.

[2] https://docs.yjs.dev/api/undo-manager
[3] See Appendix A for documentation of this result.

Acknowledgments

This work was done while Martin Kleppmann was at the Technical University of Munich, funded by the Volkswagen Foundation and crowdfunding supporters including Mintter and SoftwareMill.

References

[1] Gregory D Abowd and Alan J Dix. 1992. Giving undo attention. *Interacting with computers* 4, 3 (1992), 317–342.

[2] Carlos Baquero, Paulo Sérgio Almeida, and Ali Shoker. 2017. Pure operation-based replicated data types. *arXiv preprint arXiv:1710.04469* (2017).

[3] Eric Brattli and Weihai Yu. 2021. Supporting Undo and Redo for Replicated Registers in Collaborative Applications. In *18th International Conference on Cooperative Design, Visualization, and Engineering (CDVE 2021)*. Springer LNCS volume 12983, 195–205. https://doi.org/10.1007/978-3-030-88207-5_19

[4] Stephen Dolan. 2020. Brief Announcement: The Only Undoable CRDTs Are Counters. In *39th Symposium on Principles of Distributed Computing (PODC 2020)*. ACM, 57–58. https://doi.org/10.1145/3382734.3405749 arXiv:2006.10494

[5] 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*. 154–178.

[6] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (jul 1978), 558–565. https://doi.org/10.1145/359545.359563

[7] Stéphane Martin, Pascal Urso, and Stéphane Weiss. 2010. Scalable XML Collaborative Editing with Undo. In *On the Move to Meaningful Internet Systems (OTM)*. Springer LNCS volume 6426, 507–514. https://doi.org/10.1007/978-3-642-16934-2_37 arXiv:1010.3615

[8] Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. 2016. Near real-time peer-to-peer shared editing on extensible data types. In *Proceedings of the 2016 ACM International Conference on Supporting Group Work*. 39–49.

[9] Nuno Preguiça. 2018. Conflict-free replicated data types: An overview. *arXiv preprint arXiv:1806.10254* (2018).

[10] Matthias Ressel and Rul Gunzenhäuser. 1999. Reducing the problems of group undo. In *Proceedings of the 1999 ACM International Conference on Supporting Group Work*. 131–139.

[11] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A comprehensive study of convergent and commutative replicated data types.* Technical Report.

[12] Chengzheng Sun. 2000. Undo any operation at any time in group editors. In *Proceedings of the 2000 ACM conference on Computer supported cooperative work*. 191–200.

[13] The Automerge Contributors. 2024. *Automerge CRDT Library.* https://github.com/automerge.

[14] The YJS Contributors. 2024. *YJS Library.* https://github.com/yjs/yjs.

[15] Evan Wallace. 2019. *How Figma's multiplayer technology works.* https://web.archive.org/web/20230904085717/https://www.figma.com/blog/how-figmas-multiplayer-technology-works/

[16] Stephane Weiss, Pascal Urso, and Pascal Molli. 2010. Logoot-Undo: Distributed Collaborative Editing System on P2P Networks. *IEEE Transactions on Parallel and Distributed Systems* 21, 8 (Aug. 2010), 1162–1174. https://doi.org/10.1109/TPDS.2009.173

[17] Weihai Yu, Luc André, and Claudia-Lavinia Ignat. 2015. A CRDT Supporting Selective Undo for Collaborative Text Editing. In *15th IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems (DAIS 2015)*. Springer LNCS volume 9038, 193–206. https://doi.org/10.1007/978-3-319-19129-4_16

[18] Weihai Yu, Victorien Elvinger, and Claudia-Lavinia Ignat. 2019. A Generic Undo Support for State-Based CRDTs. In *23rd International Conference on Principles of Distributed Systems (OPODIS 2019)*. Dagstuhl LIPIcs, Article 14. https://doi.org/10.4230/LIPIcs.OPODIS.2019.14

A Reproducibility

We provide all artifacts of this research on GitHub[4]. This gives access to our prototype implementation, its test suite, LaTeX sources, and documents the surveys of other software we have conducted. Yjs' undo behavior is recorded with unit tests.

B Experimental Evaluation

To support the theoretical runtime analysis from Section 6 we benchmarked our reference implementation. The benchmarking was conducted on a 2019 MacBook Pro with a 2.6 GHz 6-Core Intel Core i7 processor and Node version 18.17.1.

Figure 4 demonstrates that caching preserves the constant runtime of the algorithm even in the worst case scenario of an alternating undo-redo sequence. A length of an alternating undo-redo sequence of n translates into a linear operation history of one *SetOp* (to enable undoing), followed by n pairs of an undo and a redo. The figure shows the mean runtime to resolve the head(s) to the register's values over 1024 runs for such sequences of lengths 200, 400, 600, and 800, if the head is

- a *RestoreOp* from an undo (blue),
- a *RestoreOp* from a redo (red),
- and a *RestoreOp* from a redo but with caching (brown).

[4] https://github.com/lstwn/undo-redo-replicated-registers/tree/papoc-handin

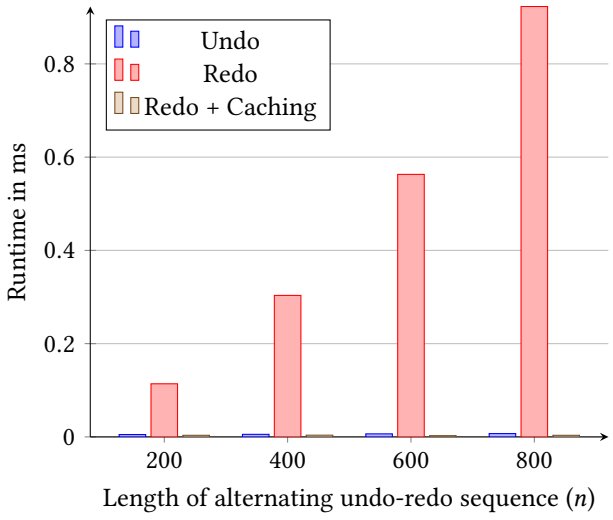

Figure 4. Runtime of the last undo/redo operation in a sequence of alternating undo-redo operations of length n (counting one undo-redo-pair as one).

The findings show that without caching there is a linear runtime w.r.t. n. Enabling caching turns the algorithm into a constant time algorithm, irrespective of the length of the operation history. Furthermore, it indicates that if space is more precious than runtime, the price to pay for the traversal may be acceptable, as even for alternating undo-redo sequences of length 800, which we believe to be rare in practice, the runtime is below a millisecond.