

The Art of the Fugue: Minimizing Interleaving in Collaborative Text Editing

Matthew Weidner 

Carnegie Mellon University, Pittsburgh, PA, USA

Joseph Gentle 

Independent

Martin Kleppmann 

Technical University of Munich, Germany

Abstract

Existing algorithms for replicated lists, which are widely used in collaborative text editors, suffer from a problem: when two users concurrently insert text at the same position in the document, the merged outcome may interleave the inserted text passages, resulting in corrupted and potentially unreadable text. The problem has gone unnoticed for decades, and it affects both CRDTs and Operational Transformation. This paper presents Fugue, the first algorithm that guarantees maximal non-interleaving, our new correctness property for replicated lists. We present two variants of the Fugue algorithm, one based on a tree and the other based on a list, and prove that they are semantically equivalent. We also implement Fugue and demonstrate that it offers performance comparable to state-of-the-art CRDT libraries for text editing.

2012 ACM Subject Classification Theory of computation → Distributed algorithms; Human-centered computing → Collaborative and social computing systems and tools

Keywords and phrases distributed data structures, replica consistency, collaborative text editing, Conflict-free Replicated Data Types (CRDTs), operational transformation

Funding *Matthew Weidner*: Supported by an NDSEG Fellowship sponsored by the US Office of Naval Research

Martin Kleppmann: Supported by the Volkswagen Foundation and crowdfunding supporters including Mintter, David Pollak, and SoftwareMill

1 Introduction

Collaborative text editors such as Google Docs allow several users to concurrently modify a document, while ensuring that all users' copies of the document converge to the same state. This type of software is implemented as a replicated list of characters; a replicated list object is a distributed data structure whose semantics can be formally specified [1].

Even though algorithms for replicated lists and collaborative text editing have been studied for over three decades [6, 18, 31], a formal specification of the required behavior of a replicated list only appeared as recently as 2016 [1]. We argue that this specification is incomplete. In Section 2 we show that there is an additional correctness property that is important in practice, but which has been overlooked by almost all prior research on this topic: *non-interleaving*. Informally stated, this property requires that when sections of text are composed independently from each other (perhaps while the users are offline), and the edits are subsequently merged, those sections are placed one after another, and not intermingled in the final document. (We give a formal definition in Section 5.2.)

There are two main families of algorithms for replicated lists: Conflict-free Replicated Data Types (CRDTs) [28, 23], and Operational Transformation (OT) [6, 31]. Perhaps surprisingly, *all* existing text collaboration algorithms we surveyed, both CRDT and OT,

suffer from interleaving. The probability of interleaving occurring varies depending on the algorithm, but we are not aware of any existing algorithm that rules out interleaving entirely.

To address this situation, this paper makes the following contributions:

- We survey a selection of existing CRDT and OT algorithms for list replication and collaborative text editing, and highlight interleaving anomalies with all of them (Section 2.3).
- We demonstrate that the previous attempt to address the interleaving problem [16] is flawed: its definition of non-interleaving is impossible to satisfy, and the proposed algorithm in that paper does not converge (Section 2.4).
- We extend the formal specification of replicated lists by Attiya et al. [1] with a new property, which we call *maximal non-interleaving* (Section 5.2). This definition is subtle: we show that an alternative, simpler definition is also impossible to satisfy (Section 5.1).
- We introduce *Fugue*, a CRDT algorithm for replicated lists that (to our knowledge) is the first algorithm to satisfy maximal non-interleaving (Section 3). We present two different formulations of Fugue, one based on trees and one based on lists, and we prove that they are semantically equivalent (Section 6).
- We provide an optimized open source implementation of Fugue, and show that it achieves memory, network, and CPU performance comparable to the state-of-the-art Yjs library on a realistic text-editing trace (Section 4).

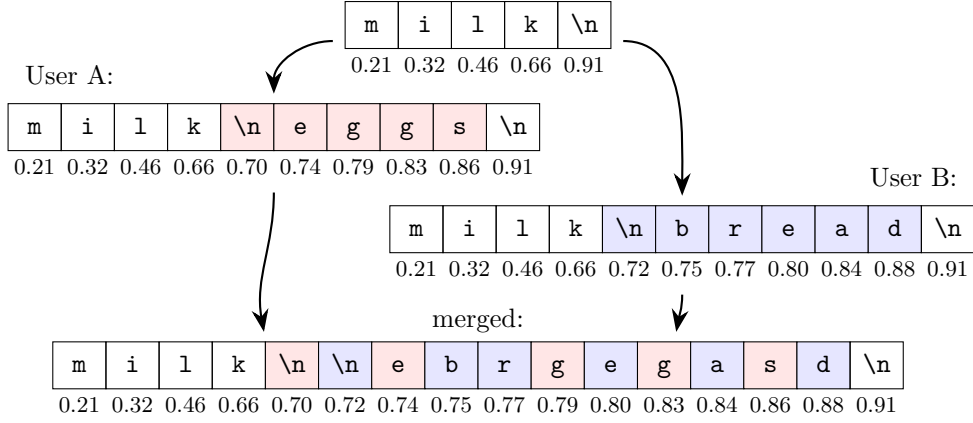
2 Background and Related Work

In collaborative text editors, each user session (e.g., in a web browser) maintains a replica of the list of characters. On user input, the user’s local replica of the document is updated by inserting or deleting characters in this list. Local edits are applied immediately, without waiting for network communication with any other nodes, in order to provide a responsive user experience independently of network latency. A user’s edits are then asynchronously propagated via the network to collaborators’ replicas, which integrate them into their local state. We can also generalize the model beyond text: instead of a list of characters, the replica could represent a list of other objects, such as items on a to-do list.

The expected behavior of such a replicated list was specified by Attiya et al. [1]; we summarize this specification in the proof of Theorem 1 in Section 3. Restated informally, it requires all replicas to converge to the same state; that state must reflect all edits made by users, and it must place the list elements (i.e., characters) in a valid order. The order is valid if list elements remain in the order in which a user inserted them; however, elements concurrently inserted on different replicas may be ordered arbitrarily.

Collaborative text editing originated with the work of Ellis and Gibbs [6], who also introduced Operational Transformation (OT) as a technique for resolving concurrent edits. This approach was formalized by Ressel et al. [25], and further developed by Sun et al. [30] and many other papers. The OT algorithm Jupiter [18] later became the basis for real-time collaboration in Google Docs [5]. Some OT algorithms, including Jupiter, assume a central server, while others allow more flexible network topologies.

Following bugs in several OT algorithms, which failed to converge in some situations [8, 21], Conflict-free Replicated Data Types (CRDTs) were developed as an alternative approach [28]. The first CRDT for a replicated list was WOOT [22], which was followed by Treedoc [24], Logoot [35], RGA [26], and several others. CRDTs do not assume a central server and therefore allow peer-to-peer operation. The algorithms differ in their performance characteristics, but all satisfy the strong list specification [1].



■ **Figure 1** Interleaving when character positions are taken from the rational numbers \mathbb{Q} .

2.1 The interleaving problem

Several replicated list CRDTs, including Treedoc [24], Logoot [35], and LSEQ [17], assign to each list element a unique identifier from a dense, totally ordered set. The sequence of list elements is then obtained by sorting the IDs in ascending order. To insert a new list element between two adjacent elements with IDs id_1 and id_2 respectively, the algorithm generates a new unique ID id_3 such that $id_1 < id_3 < id_2$, where $<$ is the total order on identifiers.

Say another user concurrently inserts an element with ID id_4 between the same pair of elements (id_1, id_2) such that $id_1 < id_4 < id_2$. The minimum requirement of $id_3 \neq id_4$ is easy to achieve (e.g., by including in each ID the unique name of the replica that generated it), but whether $id_3 < id_4$ or $id_3 > id_4$ is an arbitrary choice.

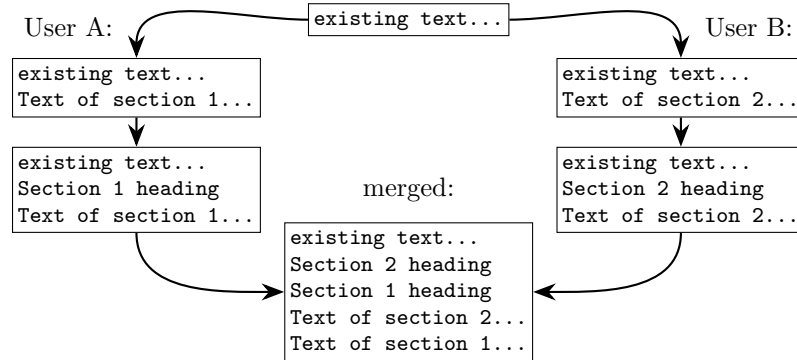
When two users concurrently insert several new elements in the same ID interval, the result is an effect illustrated in Figure 1. (The figure uses a rational number as each ID, whereas actual algorithms use a path through a tree, but the resulting behavior is similar.) The diagram shows the state of a text document containing a shopping list, initially containing the word “milk” and a newline character. User A inserts a line break and “eggs”, while concurrently user B inserts a line break and “bread”. The merged result contains “milk”, a blank line, and then the word “ebrgegasd”.

We argue that this behavior is obviously undesirable. Nevertheless, none of the affected papers even mention the issue, and although it had been informally known to people in the field for some time, it was not documented in the research literature until 2018 [15, 32]. Attiya et al.’s specification [1] allows interleaving like in Figure 1.

2.2 Interleaving of forward and backward insertions

In some replicated list algorithms, whether interleaving can occur or not depends on the order in which the elements are inserted into the list. In the common case, when a user writes text, they insert characters in forward direction: that is, “bread” is inserted as the character sequence “b”, “r”, “e”, “a”, “d”. However, not all writing is in forward direction: sometimes users hit backspace to fix a typo, or move their cursor to a different location in the document and continue typing there.

A particular editing pattern that is interesting from an interleaving point of view is insertion of list elements in backward direction. The extreme case of typing text in reverse



■ **Figure 2** Each user first types a section of text, then moves their cursor back to the start of the section, and adds a heading. When these edits are merged in an algorithm that allows interleaving of backward insertions, the merged result may place the headings and sections in an illogical order.

order character by character (typing “bread” as “d”, “a”, “e”, “r”, “b”) is unlikely to occur in practical text editing scenarios. However, a plausible scenario of backward insertion is illustrated in Figure 2. In this example, two users are working offline on a document. Each user appends a text passage to the document, moves the cursor back to the beginning of the passage they inserted, and then adds a heading for their new section. The insertion of the passage and the insertion of the heading occur in backward order.

When the two users in Figure 2 go back online and merge their changes, an algorithm that rules out interleaving of forward insertions but allows interleaving of backward insertions may place the headings and the text passages in a surprising order (for example, placing both headings before both text passages, perhaps in different orders). This behavior is less bad than the fine-grained character-by-character interleaving of Figure 1, but it is nevertheless not ideal. It would be preferable to keep all of each user’s insertions as one contiguous string, regardless of the order in which the elements were inserted.

Another reason for avoiding interleaving of backward insertions is that OT/CRDT algorithms for replicated lists are not only used for text, but also for other multi-user applications with ordered sequences, such as the rows of a spreadsheet, or the items in a to-do list. With these applications, backward insertion is more likely to occur: for example, in a spreadsheet or to-do list where new rows/items are regularly inserted at the top, one at a time. If we can avoid both forward and backward interleaving, we also improve the behavior of these multi-user applications.

2.3 Algorithms that exhibit interleaving

The interleaving problem was first noticed in CRDTs such as Logoot and LSEQ because they are particularly prone to the problem; experiments with implementations of these algorithms are easily able to trigger interleaving in practice [11]. However, when we started looking at the problem more closely, we found that interleaving is surprisingly prevalent among both OT and CRDT algorithms for collaborative text editing. Our findings are summarized in Table 1, and examples of each instance of interleaving are detailed in Appendix A.

Occurrence of interleaving is often nondeterministic, and the probability of exhibiting interleaving varies depending on the algorithm: for example, in some algorithms it depends on the exact order in which concurrently sent network messages are received, and in some it depends on random numbers generated as part of the algorithm. But we have not been able

■ **Table 1** Various algorithms’ susceptibility to interleaving anomalies. Key: ● = interleaving can occur; ○ = we have not been able to find examples of interleaving; ○✓ = proven not to interleave; ⚡ = algorithm may incorrectly reorder characters. Examples of anomalies appear in Appendix A.

Family	Algorithm	forward interleaving (one replica)	forward interleaving (multi-replica)	backward interleaving (one replica)	backward interleaving (multi-replica)
OT	adOPTed [25]	●	●	○	●
	Jupiter [18]	●	●	○	○
	GOT [31]	●	●	⚡	⚡
	SOCT2 [29]	●	●	●	●
	TTF [20]	●	●	○	●
CRDT	WOOT [22]	●	●	○	○
	Logoot [35]	●	●	●	●
	LSEQ [17]	●	●	●	●
	Treedoc [24]	●	●	●	●
	RGA [26]	○✓	○✓	●	●
	Yjs [10]	○✓	○✓	○	●
	Fugue (this work)	○✓	○✓	○✓	○✓

to find any published algorithm that rules out interleaving entirely.

In some algorithms, interleaving occurs only if multiple replicas participate in one of the concurrent editing sessions; this is indicated in the columns labeled “multi-replica”. This can happen, for example, if a user starts some work on one device and then continues on another device (producing an editing session that spans two devices), while independently another user is working offline on the same document on a third device. It can also occur in systems with ephemeral replica IDs, such as a web application that generates a fresh replica ID every time its browser tab is refreshed.

In the cases marked ○ in Table 1 we conjecture non-interleaving, but we have not proved it. Only in the cases marked ○✓ has non-interleaving been proven. RGA forward non-interleaving was proved by Kleppmann et al. [15], Yjs forward non-interleaving is proved in Appendix E of this paper, and our own algorithm Fugue is verified in Section 5.2.

2.4 Previous attempt to ensure non-interleaving

Kleppmann et al. [16] previously identified the interleaving problem. That work has two serious flaws:

1. The definition of non-interleaving in that paper cannot be satisfied by any algorithm.
2. The CRDT algorithm proposed in that paper, which aims to be non-interleaving, is incorrect – it does not converge. Appendix A.3 gives an example found by Chandrassery [4].

That paper defines non-interleaving as follows (paraphrased):

Suppose two sets of list elements X and Y satisfy:

- All elements in X were inserted concurrently to all elements in Y .
- The elements were inserted at the same location in the document, that is: after applying the insertions for $X \cup Y$ and their causal predecessors, $X \cup Y$ are contiguous in the list order.

Then either X appears before Y or vice-versa. That is, either $\forall x \in X, y \in Y. x < y$ or $\forall x \in X, y \in Y. y < x$, where $<$ is the order of elements in the final list.

To show that no replicated list algorithm can satisfy this definition, it is sufficient to give a counterexample. Starting from an empty list, suppose four replicas concurrently each insert one element. After applying these four insertions, the list state must be some ordering of these four elements; let the order be $abcd$. Then $X = \{a, c\}$ and $Y = \{b, d\}$ satisfy the two hypotheses, but they are interleaved. Since this situation could arise with any algorithm, it cannot be prevented.

In Section 5.2 we give a new definition of non-interleaving that can be implemented.

3 The Fugue algorithm based on trees

We now introduce *Fugue* (pronounced [fju:g]), the first non-interleaving algorithm for replicated lists and collaborative text editing. It is named after a form of classical music in which several melodic lines are interwoven in a pleasing way. We evaluate *Fugue* implementations in Section 4, and we analyze the algorithm’s non-interleaving properties in Section 5.

The internal structure of the *Fugue* algorithm in this section is a tree, so we also call it *Tree-Fugue*. In Section 6 we show that there is an alternative formulation of the algorithm based on lists, which we call *List-Fugue*. The two formulations have the same API and the same behavior, and differ only in their internal representation. We refer to both collectively as *Fugue* when the internal structure is not important.

We describe *Fugue* as an operation-based CRDT, although it can easily be reformulated as a state-based CRDT. The external interface of *Fugue* is an ordered sequence of values, e.g., the characters in a text document. Since the same value may appear multiple times in a list, we use *element* to refer to a unique instance of a value. Then the operations on the list are:

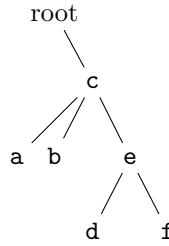
- **insert**(i, x): Inserts a new element with value x at index i , between the existing elements at indices $i - 1$ and i . All later elements (index $\geq i$) shift to an incremented index.
- **delete**(i): Deletes the element at index i . All later elements (index $\geq i + 1$) shift to a decremented index.

Note that we omit operations to mutate or move elements; these can be implemented by combining a replicated list with other CRDTs [13]. We also omit optimizations that compress consecutive runs of insertions or deletions; these can be added later without affecting the core algorithm. At a high level the algorithm works as follows:

State The state of each replica is a tree in which each non-root node is labeled with a unique ID and a value. Each non-root node is marked as either a *left* or *right* child of its parent, but the tree is not necessarily binary: a parent can have multiple left children or right children, as illustrated in Figure 3. The tree does not need to be balanced.

Each non-root node in the tree corresponds to an element in the list (e.g., a character in the text document). The list order is given by the depth-first in-order traversal over this tree: first recursively traverse a node’s left children, then visit the node’s own value, then traverse its right children. *Same-side siblings*—nodes with the same parent and the same side—are traversed in lexicographic order of their IDs; the exact construction of IDs is not important.

Insert To implement **insert**(i, x), a replica creates a new node, labeled with a new unique ID and value x , at an appropriate position in its local tree: if the element at index $i - 1$ has no right children, the new node becomes a right child of the element at index $i - 1$; otherwise, the new node is added as a left child of the next element. Figure 4 illustrates how



■ **Figure 3** One possible Tree-Fugue structure for the list `abcdef`. Observe that `a` and `b` are both left children of `c`; they are sorted lexicographically by their elements' IDs.



(a) Inserting a new element `g` between `a` and `b`. When `a` has no right children, making `g` a right child of `a` places it immediately after `a` in the traversal.

(b) Inserting a second new element `h` between `a` and `g`. When `g` has no left children, making `h` a left child of `g` puts it in the correct place.

■ **Figure 4** Cases for inserting a new element between two existing, adjacent elements.

this choice is made, and Theorem 1 shows that this approach results in the desired behavior. The replica then uses a causal broadcast protocol to send the new node, its parent, and its side (left or right child) to other replicas, which add the node to their own local trees.

A replica will not create a new node where it already has a same-side sibling, i.e., it will try to keep the tree binary. However, multiple replicas may concurrently insert nodes at the same position, creating same-side siblings like `a` and `b` in Figure 3.

Delete To implement `delete(i)`, a replica looks up the node at index i in the current list state, then causally broadcasts a message containing that element's ID. All replicas then replace that node's value with a special value \perp , flagging it as deleted (i.e., making it a *tombstone*). Nodes with this value are skipped when computing the external list state (e.g., for the purpose of computing indices of list elements); however, their non-deleted descendants are still traversed normally, and a deleted node may still be used as a parent of a new node.

We cannot remove a deleted element's node entirely: it may be an ancestor to non-deleted nodes, including nodes inserted concurrently. In Section 4 we discuss ways of mitigating memory usage from tombstones.

Pseudocode Algorithm 1 gives pseudocode for Tree-Fugue. Following the conventional notation for operation-based CRDTs [27], each operation is described in terms of a *generator* and an *effector*. The generator is called to handle user input on the user's local replica, and it returns a message to be broadcast to the other replicas. Each replica, including the sender, applies the operation by passing this message to the corresponding effector; the sender does

so atomically with the generator call. We assume that messages are received exactly once on each replica, and in causal order: if a replica received message m before generating message m' , then all replicas receive m before m' . This is a standard assumption for operation-based CRDTs [27], and is easily realized in practice using a causal broadcast protocol [2].

► **Theorem 1.** *Algorithm 1 satisfies the strong list specification of Attiya et al. [1].*

Proof. For any execution, we must show that there is a total order $<$ on all list elements (across all replicas), such that:

- (a) At any time, calling `values()` on a replica returns the list of values corresponding to all elements for which the replica received `insert` messages, minus the elements for which it received `delete` messages, in order $<$.
- (b) Suppose a replica's `values()` query yields values corresponding to elements $[a_0, a_1, \dots, a_{n-1}]$ just before the insert generator `insert(i, x)` is called. Then the inserted element e satisfies $a_0, a_1, \dots, a_{i-1} < e < a_i, a_{i+1}, \dots, a_{n-1}$.

Let $<$ be the total order given by the depth-first in-order traversal on the union of all replicas' local trees (with tombstone nodes overriding nodes with the originally inserted value). To show (a), note that by the causal order delivery assumption, a `delete` message is received after its corresponding `insert` message. Therefore, on any given replica, the set of tree nodes with $value \neq \perp$ are those nodes that have been inserted but not deleted on that replica. These are exactly the nodes whose values are returned by `values()`, in the same order as $<$ because the same traversal is used.

To show (b), note that `leftOrigin` and `rightOrigin` are consecutive elements in the tree traversal, and `leftOrigin` has no right children, inserting the new node as a right child of `leftOrigin` makes the new node the immediate successor of `leftOrigin` in the tree traversal. If `leftOrigin` does have right children, `rightOrigin` must be a descendant of `leftOrigin`, and `rightOrigin` must have no left children (since otherwise `leftOrigin` and `rightOrigin` would not be consecutive), and therefore inserting the new node as a left child of `rightOrigin` ensures the traversal visits the new child between `leftOrigin` and `rightOrigin`. In either case, the newly inserted element appears between a_{i-1} and a_i in the tree traversal, as required. ◀

4 Implementation and Evaluation

We implemented several variations of Fugue in TypeScript. Each is written as a custom CRDT for the Collabs library [34]; Collabs then provides causal order delivery and other utilities. All implementations are available as open-source software on GitHub.¹ The variations are:

Tree-Fugue An optimized implementation of Algorithm 1 in 1543 lines of code. It uses practical optimizations inspired by Yjs [9] and RGASplit [3]. In particular, it condenses sequentially-inserted tree nodes into a single “item” object instead of one object per node, and it uses Protocol Buffers to efficiently encode update messages and saved documents. Collabs v0.6.1 uses this implementation for its list CRDTs.

Tree-Fugue Simple A direct implementation of Algorithm 1 in 299 lines of code. It represents the state as a doubly-linked tree with one object per node, and it uses JSON encodings.

List-Fugue Simple A direct implementation of Algorithm 2 in 272 lines of code. It represents the state as a doubly-linked list with one object per element, and it uses JSON encodings.

¹ <https://github.com/mweidner037/fugue>


```

1  types:
2    RID, type of replica identifiers
3    ID := (RID × ℕ) ∪ {null}, type of element IDs
4    V, type of values
5    ⊥, a marker for deleted nodes
6    {L, R}, type of a child node's side (left or right)
7    NODE := ID × (V ∪ {⊥}) × ID × {L, R}, tree node tuples (id, value, parent, side)

8  per-replica CRDT state:
9    replicaID ∈ RID: the unique ID of this replica
10   tree ⊆ NODE: a set of tree nodes, initially {root} where root = (null, ⊥, null, null)
11   counter ∈ ℕ: a counter for generating element IDs, initially 0

12 query values() : V[]
13   function traverse(nodeID) : V[]
14     values ← []
15     node ← the unique node ∈ tree such that node.id = nodeID
16     for child ∈ {(id, v, p, s) ∈ tree | p = nodeID ∧ s = L} ordered by id do
17       values ← values + traverse(child.id)
18     if node.value ≠ ⊥ then
19       values ← values + [node.value]
20     for child ∈ {(id, v, p, s) ∈ tree | p = nodeID ∧ s = R} ordered by id do
21       values ← values + traverse(child.id)
22     return values
23   return traverse(null)

24 update insert
25   generator (i, x)
26     id ← (replicaID, counter); counter ← counter + 1
27     leftOrigin ← node for (i - 1)-th value in values(), or root if i = 0
28     if ∄id', v'. (id', v', leftOrigin.id, R) ∈ tree then
29       node ← (id, x, leftOrigin.id, R) // right child of leftOrigin; see Figure 4a
30     else
31       rightOrigin ← next node after leftOrigin in the tree traversal that includes
32         tombstones
33       node ← (id, x, rightOrigin.id, L) // left child of rightOrigin; see Figure 4b
34     return (insert, node)
35   effector (insert, node)
36     tree ← tree ∪ {node}

37 update delete
38   generator (i)
39     node ← node for i-th value in values()
40     return (delete, node.id)
41   effector (delete, id)
42     node ← the unique node ∈ tree such that node.id = id
43     node.value ← ⊥

```

■ **Algorithm 1** Pseudocode for the Tree-Fugue algorithm.

■ **Table 2** Saved document metrics. The plain text (without CRDT metadata or tombstones) is 105 kB in size.

Implementation	Save size (kB)	Save time (ms)	Load time (ms)
Automerge-Wasm	142 \pm 0	520 \pm 40	3,281 \pm 171
Yjs	160 \pm 0	20 \pm 1	79 \pm 8
Y-Wasm	160 \pm 0	2 \pm 0	13 \pm 1
Tree-Fugue	168 \pm 0	14 \pm 1	11 \pm 1
Tree-Fugue Simple	18,726 \pm 0	140 \pm 6	362 \pm 16
List-Fugue Simple	33,751 \pm 0	299 \pm 1	389 \pm 3

4.1 Benchmarks

We evaluated our implementations using Jahns’s crdt-benchmarks repository.² All benchmarks were run on a Dell Latitude 7400 with a 1.90GHz Intel i7-8665U processor, 16 GiB of RAM, and Ubuntu 22.04.1. The JavaScript environment was Node.js v16.13.1. For each metric, we performed 5 warmup trials followed by 10 measured trials; tables show mean \pm standard deviation for the 10 measured trials.

We also compared to existing implementations in the crdt-benchmarks repository:

Automerge-Wasm (v0.1.6) is a Rust implementation of the Automerge library,³ compiled to WebAssembly for web-based collaborative apps. Its list CRDT is based on RGA [26].

Yjs (v13.5.44) is a JavaScript library for web-based collaborative apps [10]. Its list CRDT is based on YATA [19] and it is known for its good performance [9].

Y-Wasm (v0.12.2) is a Rust-to-WebAssembly variant of Yjs.

Tables 2 and 3 show results from a benchmark that replays a real-world text-editing trace [12], in which every keystroke of the writing process for the L^AT_EX source of a 17-page paper [14] was captured. It consists of 182,315 single-character insert operations and 77,463 single-character delete operations, resulting in a final document size of 104,852 characters (not including tombstones). Each implementation processed the full trace sequentially on a single replica. Results for additional benchmarks, including microbenchmarks with concurrent operations, can be found in our code repository.

Table 2 considers the final saved document including CRDT metadata. In a typical collaborative app, this saved document would be saved (possibly on a server) at the end of each user session, and loaded at the start of the next session. Thus save size determines disk/network usage, while save/load time determines user-perceived save and startup latencies.

We see that Tree-Fugue is comparable to state-of-the-art Yjs on all three metrics, and the CRDT metadata is only 60% of the literal text’s size. Tree-Fugue Simple and List-Fugue Simple are worse but still usable in practice. The large save sizes of these implementations are due to their inefficient JSON encodings; GZIP compresses the saved documents $\approx 20\times$.

Table 3 shows performance metrics for live usage by a single user. Memory usage shows the increase in heap used⁴ from the start to the end of the trace and thus approximates each list CRDT’s in-memory size. Network bytes/op shows the average size of the per-op

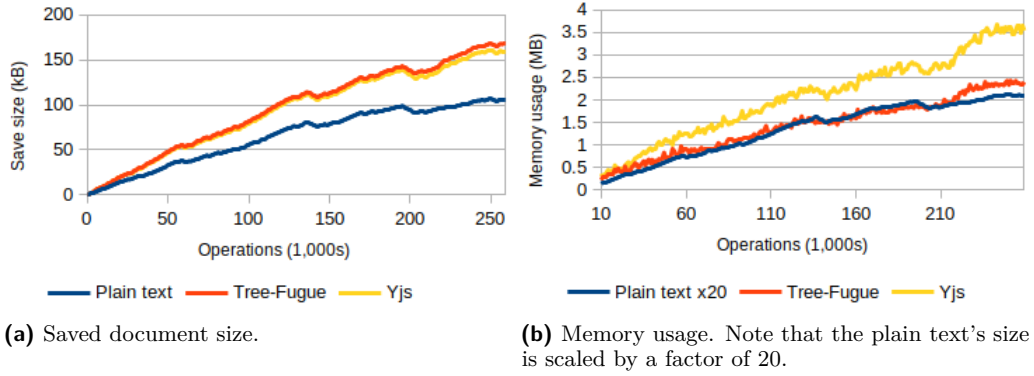
² <https://github.com/dmonad/crdt-benchmarks/>

³ <https://github.com/automerge/automerge>

⁴ Measured with Node.js’s `process.memoryUsage().heapUsed` after garbage collection. We exclude WebAssembly libraries because they do not use the JavaScript heap. While resident set size measures WebAssembly memory usage, it had frequent outliers and gave implausible values for Automerge-Wasm.

■ **Table 3** Metrics for replaying a character-by-character text editing trace with 260k operations.

Implementation	Memory usage (MB)	Network bytes/op	Ops/sec (1,000s)
Automerge-Wasm	–	224 ± 0	65 ± 2
Yjs	3.2 ± 0.2	29 ± 0	51 ± 0
Y-Wasm	–	29 ± 0	5 ± 0
Tree-Fugue	2.4 ± 0.0	39 ± 0	199 ± 5
Tree-Fugue Simple	64.9 ± 0.0	145 ± 0	19 ± 1
List-Fugue Simple	24.1 ± 0.0	178 ± 0	4 ± 0



■ **Figure 5** Metrics as a function of progress through the real text trace.

messages sent to remote collaborators. Ops/sec shows the average operation throughput; it reflects the time to process an op and encode a message for remote collaborators. For example, Tree-Fugue achieves 199,000 ops/sec, an average of 5 μ s per operation.

We again see that Tree-Fugue is practical and comparable to Yjs. In particular, its memory usage is only a few MB—about 23 bytes per character, or 13 bytes per characters-including-tombstones. This refutes a common criticism of CRDTs for collaborative text editing: namely, that they have too much per-character memory overhead [33]. The memory overhead is worse for Tree-Fugue Simple (619 bytes/char) and List-Fugue Simple (230 bytes/char), but the total is still well within modern memory limits. For all Fugue variants, the network usage and operation throughput are far from being bottlenecks, given that a typical user types at ≈ 10 chars/sec and a typical collaborative document has < 100 simultaneous users.

Figures 5a and 5b show how save size and memory usage vary throughout the text editing trace. The size of the plain text (without CRDT metadata or tombstones) is given for comparison. Observe that both metrics track the plain text’s size at a modest multiple, and save size even decreases when text is deleted, despite tombstones.

Finally, Table 4 shows selected metrics for the same text-editing trace but repeated 100

■ **Table 4** Metrics for the real text trace repeated 100 times sequentially. The literal text has size 10,485 kB. We exclude implementations that use excessive time or memory.

Implementation	Save size (kB)	Save time (ms)	Load time (ms)	Memory usage (MB)
Yjs	$15,989 \pm 0$	374 ± 26	$2,461 \pm 664$	288 ± 17
Tree-Fugue	$17,845 \pm 0$	$1,644 \pm 193$	695 ± 25	223 ± 0

times. The final document contains 10.5 million characters—far longer than any typical text document. Nonetheless, Tree-Fugue’s performance remains tolerable: 18MB save size, less than 2 seconds to save or load, and 223MB memory usage. Additionally, average network usage and throughput (not shown) remain within a $2\times$ factor of Table 3.

5 Non-Interleaving

We have proven that Tree-Fugue satisfies the strong list specification (Theorem 1). We now show that it also avoids the interleaving problem described in Section 2. Specifically, we prove that Tree-Fugue is *maximally non-interleaving*: it avoids interleaving of both forward and backward insertions, to the maximum extent possible. Intuitively, this holds because concurrent edits end up in different subtrees, which are traversed separately.

5.1 Impossibility Result

We already saw that the definition of non-interleaving by Kleppmann et al. [16] is impossible to satisfy (Section 2.4). In this section we show that a second, seemingly reasonable definition of non-interleaving is also impossible to satisfy. This second definition is the conjunction of forward and backward non-interleaving, which we define below.

Throughout this section, fix a replicated list satisfying the strong list specification [1]. Let $<$ be its (implicit global) total order on elements. In an execution using this replicated list, the *left origin* of an element is the element directly preceding the insertion position at the time of insertion. That is, if the element was inserted by an $\text{insert}(i, x)$ call, then its left origin was at index $i - 1$ at the time of this call. If there was no such element ($i = 0$), then its left origin is a special symbol *start* such that $\text{start} < e$ for every element e . This definition coincides with the *leftOrigin* variable in Algorithm 1, except using *start* instead of *root*.

► **Definition 2** (Weak forward non-interleaving). *Suppose distinct list elements a and b_1, \dots, b_n satisfy:*

- a and b_1 have the same left origin;
- b_1, \dots, b_n form a chain of left origins, i.e., the left origin of b_j is b_{j-1} for all $j \geq 2$; and
- a was inserted concurrently to all b_j .

Then an algorithm satisfies weak forward non-interleaving if it guarantees that in the final list order, all b_j are on the same side of a , i.e., either $a < b_1, \dots, b_n$ or $b_1, \dots, b_n < a$.

In the context of collaborative text editing, if one user types a while another user (or sequence of users) concurrently types $b_1 \dots b_n$ at the same position, then this mandates that a is not interleaved with $b_1 \dots b_n$. Instead, a must appear before or after the whole sequence.

Define the *left-origin tree* to be the tree of list elements in which each element’s parent is its left origin. Observe that the tree is rooted at *start*. When walking this tree, we always use the depth-first pre-order traversal: visit a node, then traverse its children in some order. This tree’s definition is similar to causal trees [7] and timestamped insertion trees [1].

► **Proposition 3.** *For any replicated list satisfying the strong list specification, the following statements are equivalent:*

- (1) *The replicated list satisfies weak forward non-interleaving as per Definition 2.*
- (2) *The replicated list satisfies forward non-interleaving as defined by Kleppmann et al. [15, §4]: if two sequences $a_1 \dots a_m$ and $b_1 \dots b_n$ are inserted from left to right concurrently at the same position, then in the final list order, all b_j are on the same side of all a_i .*

- (3) *The list order $<$ is a depth-first pre-order traversal over the left-origin tree. In other words, the replicated list is semantically equivalent to a variant of RGA [26], where the only allowed variation is the order of siblings in the tree.*

We refer to any of these equivalent conditions as *forward non-interleaving*. The proof appears in Appendix B.

We analogously define *right origin* as the element directly following the insertion position at the time of insertion, the special symbol *end* if no following element exists, and *weak backward non-interleaving* based on a chain of right origins. The right origin is a tombstone if the list element immediately following the left origin in the list is a tombstone, like the *rightOrigin* variable in Algorithm 1; this choice simplifies the analysis but is not essential.

► **Corollary 4.** *Weak backward non-interleaving is equivalent to the statement that the list order $<$ is a depth-first post-order traversal over the right-origin tree, which is defined analogously to the left-origin tree.*

We refer to either condition as *backward non-interleaving*.

Kleppmann et al. [15] show that RGA [26] satisfies forward non-interleaving. It follows that a “reversed” version of RGA satisfies backward non-interleaving. It is then tempting to define general non-interleaving as the conjunction of forward and backward non-interleaving. Nonetheless, it turns out that this is impossible to achieve:

► **Theorem 5.** *No replicated list algorithm satisfying the strong list specification satisfies both forward non-interleaving and backward non-interleaving.*

We prove this in Appendix B, by giving an execution trace in which forward and backward non-interleaving mandate contradictory behaviors.

5.2 Tree-Fugue is Maximally Non-Interleaving

We now present a new definition, *maximal non-interleaving*, which circumvents the previous section’s impossibility results.

Since forward insertions are more common in typical text editing behavior, we begin by mandating forward non-interleaving. Proposition 3 then implies that $<$ is a depth-first pre-order traversal over the left-origin tree. Our only remaining degree of freedom is in how we sort siblings within that tree, i.e., nodes with the same left origin. So, we mandate backward non-interleaving for those siblings, but not otherwise.

Formally, for a set of siblings S in the left-origin tree, define the rooted tree $T_R|_S$ by:

- The nodes of $T_R|_S$ are $S \cup \{\text{end}\}$, and its root is *end*.
- The parent of $s \in S$ in $T_R|_S$ is its right origin, unless that is not in S , in which case s ’s parent is *end*.

In other words, $T_R|_S$ is the graph restriction of the right-origin tree to S , except with extra *end* parent relationships to ensure that it is a tree instead of a forest.

► **Definition 6** (Maximal non-interleaving). *A replicated list algorithm satisfies maximal non-interleaving if:*

- (a) *$<$ is a depth-first pre-order traversal over the left-origin tree; and*
- (b) *for each set of siblings S in the left-origin tree, the restriction of $<$ to S is some depth-first post-order traversal of $T_R|_S$ (excluding *end* from the traversal).*

Observe that maximal non-interleaving almost completely determines $<$. The only degree of freedom is: for each set of siblings S in the left-origin tree, for each set of siblings S' within $T_R|_S$, choose an arbitrary total order on S' that is consistent across replicas.

► **Theorem 7.** *Tree-Fugue (Algorithm 1) is maximally non-interleaving (Definition 6).*

The proof appears in Appendix C.

6 The Fugue Algorithm based on Lists

In this section, we describe List-Fugue, our second formulation of Fugue. Unlike Tree-Fugue, it sorts elements without help from an explicit tree structure. Instead, each replica’s state is a list of elements, and a replica inserts a new element into its local list “at the correct location” using a for-loop. Nonetheless, we prove that List-Fugue is semantically equivalent to Tree-Fugue: they induce the same total order $<$ on elements.

List-Fugue is based on Yjs’s list CRDT implementation. In Appendix E, we perform the opposite list-to-tree conversion for Yjs. We use this conversion to give a new proof that Yjs’s underlying algorithm is correct and to characterize its semantics and interleaving properties.

6.1 Algorithm

As with Tree-Fugue, we describe List-Fugue as an operation-based CRDT, although it can easily be reformulated as a state-based CRDT. In particular, we assume that messages are received (effected) exactly once on each replica, and in causal order.

Algorithm 2 gives pseudocode. A replica’s state is the list of elements it has received (including tombstones), with metadata: unique ID, left origin, and right origin. The `insert` generator broadcasts the new element together with its metadata. The `insert` effector computes *left*—the existing element immediately to the left of the new element—then inserts the new element after it, shifting all later elements to an incremented index. Deletions are handled using tombstones, like in Tree-Fugue (not shown).

The core of List-Fugue is lines 23–36, which compute *left* for a newly received element. For any correct replicated list, we know that *elt* must be inserted between *elt.leftOrigin* and *elt.rightOrigin*; thus *left* must be in the half-open interval $[elt.leftOrigin, elt.rightOrigin)$. The `insert` effector starts with *left* = *elt.leftOrigin*, then loops over the remainder of this range from left to right, occasionally updating *left* to the current loop variable. Eventually, the loop ends and the last-set value of *left* is used.

6.2 Equivalence with Tree-Fugue

Algorithmically, List-Fugue bears little resemblance to Tree-Fugue. Although List-Fugue’s for-loop is easier to implement than an explicit tree, it is not obvious what total order it enforces or that the order is even consistent across replicas.

In spite of this, we claim that List-Fugue is semantically equivalent to Tree-Fugue. This equivalence is why we consider both algorithms to be formulations of Fugue.

► **Theorem 8.** *List-Fugue is semantically equivalent to Tree-Fugue. That is, in any execution, every List-Fugue replica orders its list of elements according to Tree-Fugue’s (implicit global) order $<$ on elements.*

The proof appears in Appendix D. Essentially, we relate the conditions in List-Fugue’s for-loop to properties of the left- and right-origin tree traversals. We then show on a case-by-case basis that List-Fugue’s decisions match those of Tree-Fugue.

► **Corollary 9.** *List-Fugue satisfies the strong list specification [1], and it is maximally non-interleaving.*

```

1 per-replica CRDT state:
2   replicaID ∈ RID: the unique ID of this replica
3   start, end ∈ ID: special symbols used as in Section 5.1
4   list: a list with elements
      (id ∈ ID, value ∈  $\mathbb{V} \cup \{\perp\}$ , leftOrigin ∈ ID, rightOrigin ∈ ID), initially empty.
5   counter ∈  $\mathbb{N}$ : a counter for generating element IDs, initially 0.

6 query values() :  $\mathbb{V}[]$ 
7   values ← []
8   for elt ∈ list do
9     if elt.value ≠  $\perp$  then
10       values ← values + [elt.value]
11   return values

12 update insert
13   generator (i, x)
14     id ← (replicaID, counter); counter ← counter + 1
15     leftOrigin ← node for (i − 1)-th value in values(), or start if i = 0
16     rightOrigin ← next node after leftOrigin in list including deleted nodes, or end
      if leftOrigin is the last element
17     return (insert, (id, x, leftOrigin.id, rightOrigin.id))

18 effector (insert, elt)
19   function rightParent(p) : ID
20     if p.rightOrigin = end or p.rightOrigin.leftOrigin ≠ p.leftOrigin then
21       return end
22     else return p.rightOrigin
23   left ← elt.leftOrigin
24   scanning ← false
25   for o in list from elt.leftOrigin to elt.rightOrigin, exclusive do
26     if o.leftOrigin < elt.leftOrigin then break
27     else if o.leftOrigin = elt.leftOrigin then
28       if rightParent(o) < rightParent(elt) then
29         scanning ← true
30       else if rightParent(o) = rightParent(elt) then // o and elt are double
        siblings
31         if o.id > elt.id then break
32         else scanning ← false
33       else // rightParent(o) > rightParent(elt)
        scanning ← false
34   if scanning = false then
35     left ← o
36   Insert elt into list immediately after left
37

```

■ **Algorithm 2** Pseudocode for the List-Fugue algorithm. Types and the delete operation are the same as for Tree-Fugue (Algorithm 1) and are omitted. We abuse notation and use IDs (e.g. *elt.leftOrigin*) to mean the element with that ID. Except for *o.id* > *elt.id*, all comparisons reference the existing list order, *not* the lexicographic order on IDs, with the added rule *start* < *p* < *end* for all *p* ∈ *list*.

7 Conclusion

Interleaving of concurrent insertions at the same position is an undesirable but largely ignored problem with many replicated list algorithms that are used for collaborative text editing. Indeed, all CRDT and OT algorithms that we surveyed exhibit interleaving anomalies. We also found that existing definitions of non-interleaving are impossible to satisfy.

In this paper, we proposed a new definition, maximal non-interleaving, and the Fugue list CRDT that satisfies it. We described two formulations of Fugue, Tree-Fugue and List-Fugue, and proved that they are semantically equivalent. Our optimized implementation of Tree-Fugue has performance comparable to the state-of-the-art Yjs library.

References

- 1 Hagit Attiya, Sebastian Burckhardt, Alexey Gotsman, Adam Morrison, Hongseok Yang, and Marek Zawirski. Specification and complexity of collaborative text editing. In *2016 ACM Symposium on Principles of Distributed Computing*, PODC '16, page 259–268. ACM, 2016. doi:10.1145/2933057.2933090.
- 2 Kenneth P Birman, André Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272 – 314, August 1991. doi:10.1145/128738.128742.
- 3 Loïck Briot, Pascal Urso, and Marc Shapiro. High responsiveness for group editing CRDTs. In *19th International Conference on Supporting Group Work*, GROUP 2016, pages 51–60. ACM, November 2016. doi:10.1145/2957276.2957300.
- 4 Adithya Rajesh Chandrassery. Personal communication, 2021.
- 5 John Day-Richter. What’s different about the new Google Docs: Making collaboration fast, September 2010. URL: <https://drive.googleblog.com/2010/09/whats-different-about-new-google-docs.html>.
- 6 C A Ellis and S J Gibbs. Concurrency control in groupware systems. In *ACM International Conference on Management of Data*, SIGMOD 1989, pages 399–407, 1989. doi:10.1145/67544.66963.
- 7 Victor Grishchenko. Citrea and Swarm: Partially ordered op logs in the browser: Implementing a collaborative editor and an object sync library in JavaScript. In *1st Workshop on Principles and Practice of Eventual Consistency*, PaPEC '14. ACM, 2014. doi:10.1145/2596631.2596641.
- 8 Abdessamad Imine, Pascal Molli, Gérald Oster, and Michaël Rusinowitch. Proving correctness of transformation functions in real-time groupware. In *8th European Conference on Computer Supported Cooperative Work*, ECSCW 2003, pages 277–293, 2003. doi:10.1007/978-94-010-0068-0_15.
- 9 Kevin Jahns. Are CRDTs suitable for shared editing? Blog post, August 2020. URL: <https://blog.kevinjahns.de/are-crdts-suitable-for-shared-editing/>.
- 10 Kevin Jahns. Yjs. GitHub repository, December 2022. URL: <https://github.com/yjs/yjs>.
- 11 Martin Kleppmann. Insertion interleaving test, February 2018. URL: <https://github.com/ept/insert-interleaving>.
- 12 Martin Kleppmann. Benchmarking resources for Automerge, 2020. URL: <https://github.com/automerge/automerge-perf>.
- 13 Martin Kleppmann. Moving elements in list CRDTs. In *7th Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC '20. ACM, 2020. doi:10.1145/3380787.3393677.
- 14 Martin Kleppmann and Alastair R Beresford. A conflict-free replicated JSON datatype. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):2733–2746, April 2017. arXiv:1608.03960, doi:10.1109/TPDS.2017.2697382.

- 15 Martin Kleppmann, Victor B. F. Gomes, Dominic P. Mulligan, and Alastair R. Beresford. Opsets: Sequential specifications for replicated datatypes (extended version), 2018. [arXiv:1805.04263](#).
- 16 Martin Kleppmann, Victor B. F. Gomes, Dominic P. Mulligan, and Alastair R. Beresford. Interleaving anomalies in collaborative text editors. In *6th Workshop on Principles and Practice of Consistency for Distributed Data*, PaPoC 2019. ACM, March 2019. doi:10.1145/3301419.3323972.
- 17 Brice Nédelec, Pascal Molli, Achour Mostefaoui, and Emmanuel Desmontils. LSEQ: An adaptive structure for sequences in distributed collaborative editing. In *ACM Symposium on Document Engineering*, DocEng '13, pages 37–46. ACM, 2013. doi:10.1145/2494266.2494278.
- 18 David A Nichols, Pavel Curtis, Michael Dixon, and John Lamping. High-latency, low-bandwidth windowing in the Jupiter collaboration system. In *8th Annual ACM Symposium on User Interface and Software Technology*, UIST 1995, pages 111–120, 1995. doi:10.1145/215585.215706.
- 19 Petru Nicolaescu, Kevin Jahns, Michael Derntl, and Ralf Klamma. Near real-time peer-to-peer shared editing on extensible data types. In *19th International Conference on Supporting Group Work*, GROUP 2016, pages 39–49. ACM, November 2016. doi:10.1145/2957276.2957310.
- 20 Gérard Oster, Pascal Molli, Pascal Urso, and Abdessamad Imine. Tombstone transformation functions for ensuring consistency in collaborative editing systems. In *9th IEEE International Conference on Collaborative Computing*, 2006. doi:10.1109/colcom.2006.361867.
- 21 Gérard Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Proving correctness of transformation functions in collaborative editing systems. Technical Report RR-5795, INRIA, 2005. URL: <https://hal.inria.fr/inria-00071213/>.
- 22 Gérard Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Data consistency for P2P collaborative editing. In *ACM Conference on Computer Supported Cooperative Work*, CSCW 2006, pages 259–268, 2006. doi:10.1145/1180875.1180916.
- 23 Nuno Preguiça, Carlos Baquero, and Marc Shapiro. *Encyclopedia of Big Data Technologies*, chapter Conflict-Free Replicated Data Types CRDTs. Springer, 2018. doi:10.1007/978-3-319-63962-8_185-1.
- 24 Nuno Preguiça, Joan Manuel Marques, Marc Shapiro, and Mihai Letia. A commutative replicated data type for cooperative editing. In *29th IEEE International Conference on Distributed Computing Systems*, ICDCS 2009, pages 395–403. IEEE, 2009. doi:10.1109/ICDCS.2009.20.
- 25 Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhäuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *ACM Conference on Computer Supported Cooperative Work*, CSCW 1996, pages 288–297, 1996. doi:10.1145/240080.240305.
- 26 Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. Replicated abstract data types: Building blocks for collaborative applications. *Journal of Parallel and Distributed Computing*, 71(3):354–368, 2011. doi:<https://doi.org/10.1016/j.jpdc.2010.12.006>.
- 27 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of convergent and commutative replicated data types. Research Report RR-7506, INRIA, January 2011. URL: <https://hal.inria.fr/inria-00555588>.
- 28 Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *13th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, SSS 2011, pages 386–400, 2011. doi:10.1007/978-3-642-24550-3_29.
- 29 Maher Suleiman, Michèle Cart, and Jean Ferrié. Serialization of concurrent operations in a distributed collaborative environment. In *ACM International Conference on Supporting Group Work*, GROUP '97, pages 435–445. ACM, November 1997. doi:10.1145/266838.267369.
- 30 Chengzheng Sun and Clarence Ellis. Operational transformation in real-time group editors: Issues, algorithms, and achievements. In *ACM Conference on Computer Supported Cooperative Work*, CSCW 1998, pages 59–68, 1998. doi:10.1145/289444.289469.

- 31 Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction*, 5(1):63–108, March 1998. doi:10.1145/274444.274447.
- 32 Chengzheng Sun, David Sun, Agustina, and Weiwei Cai. Real differences between OT and CRDT for co-editors, October 2018. URL: <https://arxiv.org/abs/1810.02137>.
- 33 David Sun, Chengzheng Sun, Agustina Ng, and Weiwei Cai. Real differences between OT and CRDT in correctness and complexity for consistency maintenance in co-editors. *Proceedings of the ACM on Human-Computer Interaction*, 4(CSCW1), May 2020. doi:10.1145/3392825.
- 34 Matthew Weidner, Heather Miller, Huairui Qi, Maxime Kjaer, Ria Pradeep, Ignacio Maronna, Benito Geordie, and Yicheng Zhang. Collabs. GitHub repository, July 2022. URL: <https://github.com/composablesys/collabs>.
- 35 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*, pages 404–412, 2009. doi:10.1109/ICDCS.2009.75.

A

 Problems with various algorithms

A.1 Examples of interleaving

We give brief justifications of the interleaving claims (●) in Table 1. Terminology and notation are as in the source cited for each algorithm. For each algorithm we give a minimal example, in each case starting with an empty text document:

- One user inserts a followed by b , while concurrently another user inserts x . An algorithm exhibits weak forward interleaving (Definition 2) if a possible merged outcome is axb .
- One user inserts b , and then prepends a before b ; concurrently another user inserts x . An algorithm exhibits weak backward interleaving if a possible merged outcome is axb .

A.1.1 adOPTed and TTF forward interleaving

adOPTed [25] and TTF [20] use essentially the same transformation function for concurrent insertions. We use the notation from TTF: an insertion operation is denoted $ins(p, c, s)$, where p is the index at which to insert, c is the inserted character, and s is the ID of the site (i.e., replica) on which the operation was generated.

To demonstrate forward interleaving, replica A generates $ins(1, a, A)$ followed by $ins(2, b, A)$. Concurrently, replica B generates $ins(1, x, B)$. Assume $A < B$ and consider the execution at B :

1. B executes $ins(1, x, B)$, so the document is x .
2. When B receives $ins(1, a, A)$, it computes the transformation

$$T(ins(1, a, A), ins(1, x, B)) = ins(1, a, A) \quad \text{because } 1 = 1 \wedge A < B$$

so a is inserted before x , yielding ax .

3. When B receives $ins(2, b, A)$, it computes the transformation

$$T(ins(2, b, A), ins(1, x, B)) = ins(3, b, A) \quad \text{because } 2 > 1$$

so b is inserted after x , yielding axb .

A.1.2 adOPTed and TTF backward interleaving (multi-replica)

To demonstrate backward interleaving, we use three replicas, A , B , and C with $A < B < C$. First C generates $ins(1, b, C)$, which is sent to A , and then A generates $ins(1, a, A)$. Concurrently, B generates $ins(1, x, B)$. Consider the execution at replica B :

1. B executes $ins(1, x, B)$, so the document is x .
2. When B receives $ins(1, b, C)$, it computes the transformation

$$T(ins(1, b, C), ins(1, x, B)) = ins(2, b, C) \quad \text{because } 1 = 1 \wedge C > B$$

so b is inserted after x , yielding xb .

3. When B receives $ins(1, a, A)$, it computes the transformation

$$T(ins(1, a, A), ins(1, x, B)) = ins(1, a, A) \quad \text{because } 1 = 1 \wedge A < B$$

so a is inserted before x , yielding axb .

A.1.3 Jupiter forward interleaving

Jupiter [18] is based on a single server that determines a total order of all operations, and it always transforms a client operation and a server operation with respect to each other. When there are multiple clients and an operation from one client has been transformed and applied by the server, that transformed operation is considered a server operation from the point of view of the other clients (even though the operation actually originated on a client).

The Jupiter paper [18] does not explicitly specify the transformation function for concurrent insertions, it only describes it verbally as: “We arbitrarily chose to put server text first if both [i.e. server and client] try to insert at the same spot.” We show that even this informal definition implies that the algorithm exhibits forward interleaving.

Starting with an empty document, assume that client A generates $ins(1, a)$ followed by $ins(2, b)$, and concurrently client B generates $ins(1, x)$. Consider the execution at the server:

1. Assume that client A ’s $ins(1, a)$ is the first operation to reach the server, so the server simply applies it, resulting in the document a .
2. Next, client B ’s $ins(1, x)$ reaches the server. Since $ins(1, a)$ was concurrently applied by the server, we have two concurrent insertions at the same position. Per the rule for such insertions, the server’s character, that is a , is placed first, and the client’s x second. This means B ’s operation is transformed to $ins(2, x)$ and the server’s document now reads ax .
3. Finally, client A ’s $ins(2, b)$ reaches the server. Concurrently, the server has now performed $ins(2, x)$, which is the transformed form of B ’s operation. Again we have two concurrent insertions at the same position, in this case at index 2. Per the rule above we place the server’s character x first, and the client’s character b second. This means A ’s operation is transformed to $ins(3, b)$ and the server’s document now reads axb , exhibiting forward interleaving.

If the rule is changed to place the client’s insertion first and the server’s insertion second in the case of concurrent insertions at the same position, the algorithm exhibits backward interleaving instead of forward interleaving.

A.1.4 GOT forward interleaving

Strictly speaking, GOT is an OT control algorithm that is not specific to text; here we use GOT in conjunction with the transformation functions for text editing that are specified in the same paper [31]. For insertions, those functions are an inclusion transformation IT_II and an exclusion transformation ET_II .

To demonstrate forward interleaving, replica A generates $Insert[a, 1]$ followed by $Insert[b, 2]$, while concurrently replica B generates $Insert[x, 1]$. Let the total order on these operations be $Insert[a, 1] < Insert[x, 1] < Insert[b, 1]$; this order is possible in the GOT timestamping scheme. Consider the execution of operations at a replica that receives them in ascending order, so that we can skip the undo/do/redox process. That replica will go through the following steps:

1. Receive $Insert[a, 1]$. We now have $HB = [Insert[a, 1]]$, and the document is a .
2. Receive $Insert[x, 1]$. This operation is concurrent with all operations in HB , so it is transformed as follows:

$$EO_{new} = LIT(Insert[x, 1], HB[1, 1]) = IT_II(Insert[x, 1], Insert[a, 1]) = Insert[x, 2]$$

resulting in $HB = [Insert[a, 1], Insert[x, 2]]$ and the document ax .

3. Receive $Insert[b, 2]$. This operation is dependent on $HB[1]$ and concurrent with $HB[2]$, so it is transformed as follows:

$$EO_{new} = LIT(Insert[b, 2], HB[2, 2]) = IT_II(Insert[b, 2], Insert[x, 2]) = Insert[b, 3]$$

resulting in $HB = [Insert[a, 1], Insert[x, 2], Insert[b, 3]]$ and the document axb .

We discuss behavior of backward insertions in GOT in Appendix A.2.

A.1.5 SOCT2, Logoot, and LSEQ

Forward and backward interleaving in implementations of SOCT2 [29], Logoot [35], and LSEQ [17] are demonstrated in an open source repository [11].

A.1.6 WOOT forward interleaving

The WOOT algorithm derives a partial order from the left and right origins of each inserted character, and then defines the document to be a unique linear extension of this partial order [22]. The notation $ins(a < x < b)$ means that the character x is inserted between a and b . c_b marks the beginning and c_e marks the end of the document. When a replica receives an operation from another replica, a recursive function `IntegrateIns` determines where the insertion should be placed.

To demonstrate forward interleaving, replica A generates $ins(c_b < a < c_e)$ to insert a , followed by $ins(a < b < c_e)$ to insert b . Concurrently, replica B generates $ins(c_b < x < c_e)$ to insert x . Assume the ordering on the inserted characters' IDs is $a <_{id} x <_{id} b$. Consider the execution at replica B :

1. To apply $ins(c_b < x < c_e)$ we call `IntegrateIns(x, c_b, c_e)`, resulting in the document x .
2. To apply $ins(c_b < a < c_e)$, calling `IntegrateIns(a, c_b, c_e)` computes $S' = x$ and $L = c_b x c_e$. The loop stops at $i = 1$ because $x >_{id} a$, so we recursively call `IntegrateIns(a, c_b, x)`, resulting in the document ax .
3. To apply $ins(a < b < c_e)$, calling `IntegrateIns(b, a, c_e)` computes $S' = x$ and $L = c_b x c_e$. The loop stops at $i = 2$ because $x <_{id} b$, so we recursively call `IntegrateIns(b, x, c_e)`, resulting in the document axb .

A.1.7 Treedoc forward and backward interleaving

To demonstrate forward interleaving in Treedoc [24], replica *A* generates $\text{insert}(p_a, a)$ followed by $\text{insert}(p_b, b)$, while concurrently replica *B* generates $\text{insert}(p_x, x)$. These operations are assigned position identifiers $p_a = [(1 : d_a)]$, $p_b = [1(1 : d_b)]$, and $p_x = [(1 : d_x)]$, where d_a , d_b , and d_x are disambiguators. Assume $d_a < d_x$; then the order on position identifiers is $p_a < p_x < p_b$, resulting in the document axb .

To demonstrate backward interleaving in Treedoc, replica *A* generates $\text{insert}(p_b, b)$ and then prepends $\text{insert}(p_a, a)$, while concurrently replica *B* generates $\text{insert}(p_x, x)$. These operations are assigned position identifiers $p_b = [(0 : d_b)]$, $p_a = [0(0 : d_a)]$, and $p_x = [(0 : d_x)]$, where d_b , d_a , and d_x are disambiguators. Assume $d_x < d_b$; then the order on position identifiers is $p_a < p_x < p_b$, resulting in the document axb .

A.1.8 RGA backward interleaving

To demonstrate backward interleaving in RGA [26], replica *A* generates $\text{Insert}(1, b)$ followed by prepending $\text{Insert}(1, a)$, while concurrently replica *B* generates $\text{Insert}(1, x)$. The insertion of b is assigned an s4vector of $\vec{v}_b = \langle 0, A, 1, 0 \rangle$, the insertion of a an s4vector of $\vec{v}_a = \langle 0, A, 2, 0 \rangle$, and the insertion of x an s4vector of $\vec{v}_x = \langle 0, B, 1, 0 \rangle$. Assuming $A < B$, the order on these s4vectors is $\vec{v}_b < \vec{v}_x < \vec{v}_a$. The left cobject (i.e., left origin) of all three operations is nil, and therefore the three elements are placed in their list in descending s4vector order, resulting in the document axb .

A.1.9 Yjs backward interleaving (multi-replica)

Yjs [10] exhibits interleaving only in a limited case: when insertions occur in backward order across multiple replica IDs. The following code demonstrates such interleaving in Yjs version 13.5.44, and it is analogous to the example used in Proposition 16 below. Our code repository contains a runnable copy of this code.

```
// Set up three replicas
const Y = require('yjs')
let doc1 = new Y.Doc(), doc2 = new Y.Doc(), doc3 = new Y.Doc()
doc1.clientID = 1; doc2.clientID = 2; doc3.clientID = 3

// Replica 3 inserts 'b'
doc3.getArray().insert(0, ['b'])

// Replica 1 inserts 'a' before 'b'
Y.applyUpdateV2(doc1, Y.encodeStateAsUpdateV2(doc3, Y.encodeStateVector(doc1)))
doc1.getArray().insert(0, ['a'])

// Replica 2 concurrently inserts 'x'
doc2.getArray().insert(0, ['x'])

// Prints the merged document: "axb"
Y.applyUpdateV2(doc1, Y.encodeStateAsUpdateV2(doc2, Y.encodeStateVector(doc1)))
console.log(doc1.getArray().toArray().join(''))
```

A.2 GOT character reordering

We found that in the case of concurrent backward insertions, GOT exhibits an anomaly that is worse than interleaving: it reorders characters so that they appear in a different order from what the user typed, violating the replicated list specification. In the following example, replica *A* inserts ab in backward order, while *B* concurrently inserts x , and the resulting

document reads xba — the a and b are reordered! Like in Appendix A.1.4, we refer to the combination of the GOT control algorithm with the transformation functions specified in the same paper [31].

Assume replica A generates $Insert[b, 1]$ followed by $Insert[a, 1]$, while concurrently replica B generates $Insert[x, 1]$. Let the total order on these operations be $Insert[x, 1] < Insert[b, 1] < Insert[a, 1]$, which is consistent with causality. Consider the execution of operations at a replica that receives them in ascending order. That replica will go through the following steps:

1. Receive $Insert[x, 1]$. We now have $HB = [Insert[x, 1]]$, and the document is x .
2. Receive $Insert[b, 1]$. This operation is concurrent with all operations in HB , so it is transformed as follows:

$$EO_{new} = LIT(Insert[b, 1], HB[1, 1]) = IT_II(Insert[b, 1], Insert[x, 1]) = Insert[b, 2]$$

resulting in $HB = [Insert[x, 1], Insert[b, 2]]$ and the document xb .

3. Receive $Insert[a, 1]$. Now $HB[1]$ is concurrent to the new operation, but $HB[2]$ causally precedes it. Therefore $EOL = [Insert[b, 2]]$ and

$$\begin{aligned} EOL' &= [LET(Insert[b, 2], HB[1, 1]^{-1})] \\ &= [ET_II(Insert[b, 2], Insert[x, 1])] \\ &= [Insert[b, 1]] \end{aligned}$$

$$\begin{aligned} O'_{new} &= LET(Insert[a, 1], EOL'^{-1}) \\ &= ET(Insert[a, 1], Insert[b, 1]) \\ &= Insert[a, 1] \end{aligned}$$

$$\begin{aligned} EO_{new} &= LIT(O'_{new}, HB[1, 2]) \\ &= IT_II(IT_II(Insert[a, 1], Insert[x, 1]), Insert[b, 2]) \\ &= IT_II(Insert[a, 2], Insert[b, 2]) \\ &= Insert[a, 3] \end{aligned}$$

When EO_{new} is applied, we obtain the incorrect document state xba .

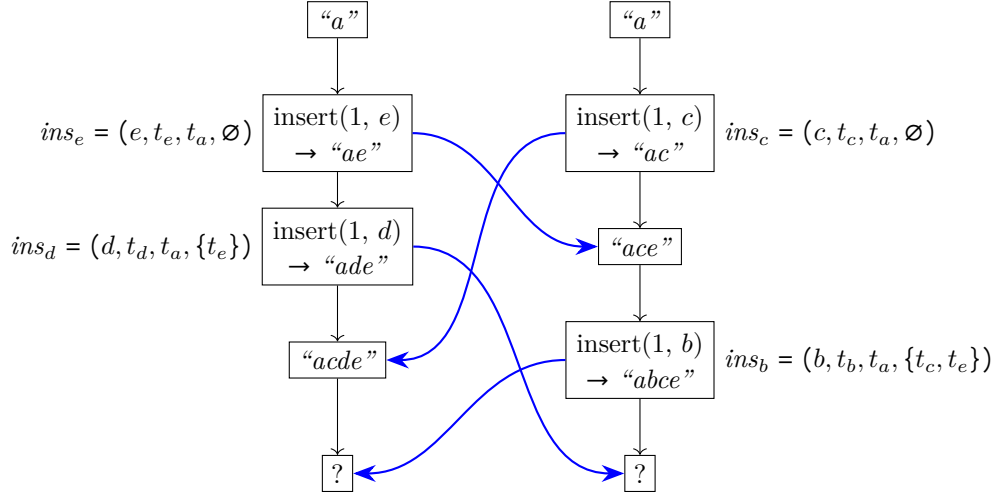
The GOT paper [31, Definition 9] specifies that the transformation functions must satisfy a reversibility requirement, $IT(ET(O_a, O_b), O_b) = O_a$. However, the insert/insert transformation functions in Section 9 of the same paper do not meet this requirement. Let $O_a = Insert[a, 1]$ and $O_b = Insert[b, 1]$. Then

$$\begin{aligned} IT(ET(O_a, O_b), O_b) &= IT_II(ET_II(Insert[a, 1], Insert[b, 1]), Insert[b, 1]) \\ &= IT_II(Insert[a, 1], Insert[b, 1]) \\ &= Insert[a, 2] \neq O_a \end{aligned}$$

We believe that this issue is not a typo or a similarly easy fix, but rather a deeper conceptual problem with the GOT algorithm.

A.3 Non-interleaving RGA does not converge

As explained in Section 2.4, Kleppmann et al. [16] previously attempted to design a non-interleaving text CRDT, but this algorithm does not converge. We now give an example of this problem, which was found by Chandrassery [4].



■ **Figure 6** Execution that leads to divergence in Kleppmann et al. [16]'s non-interleaving variant of RGA.

The algorithm is a variant of Attiya et al. [1]'s timestamped insertion tree (which is a reformulation of RGA [26]). Each insertion operation includes the ID of a *reference element* (after which the new character should be inserted), and additionally this algorithm includes the set of existing characters with the same reference element. When determining the order of characters with the same reference element, this additional metadata is taken into account.

The correctness of the algorithm depends on a strict total ordering relation $<$ that determines the order in which characters with the same reference element should appear in the document. However, it is possible to construct executions in which three insertion operations x , y , and z are ordered $x < y$, $y < z$, and $z < x$, violating the asymmetry property of the total order. If $<$ is not a total order, the order of characters in the document is ambiguous, and so the algorithm cannot guarantee that replicas converge to the same state.

Figure 6 shows an execution that triggers the problem. Each insertion operation is a tuple (a, t, r, e) where a is the character being inserted, t is the timestamp (unique ID) of the operation, r is the timestamp of the reference character (immediate predecessor at the time the operation was generated), and e is the set of siblings (operations with the same reference character at the time the operation was generated).

Assume $t_e < t_c < t_d < t_b$. To determine the total order of operations, we first apply the rule that $op_1 < op_2$ if op_1 's timestamp appears in op_2 's siblings, which gives us:

$$\begin{aligned} ins_e &= (e, t_e, t_a, \emptyset) < (d, t_d, t_a, \{t_e\}) = ins_d \\ ins_c &= (c, t_c, t_a, \emptyset) < (b, t_b, t_a, \{t_c, t_e\}) = ins_b \\ ins_e &= (e, t_e, t_a, \emptyset) < (b, t_b, t_a, \{t_c, t_e\}) = ins_b \end{aligned}$$

Next, we compare ins_b and ins_d using the rule for concurrent operations:

$$\begin{aligned} \min(\{t_b\} \cup \{t_c, t_e\} - \{t_e\}) &= t_c \\ \min(\{t_d\} \cup \{t_e\} - \{t_c, t_e\}) &= t_d \\ t_c < t_d, \text{ therefore: } ins_b &< ins_d \end{aligned}$$

Finally, we compare ins_c and ins_d using the rule for concurrent operations:

$$\min(\{t_c\} \cup \emptyset - \{t_e\}) = t_c$$

$$\min(\{t_d\} \cup \{t_e\} - \emptyset) = t_e$$

$$t_e < t_c, \text{ therefore: } ins_d < ins_c$$

We now have $ins_c < ins_b$, $ins_b < ins_d$, and $ins_d < ins_c$. This violates the requirement that $<$ is a total order. Therefore, the order of characters in the document is not uniquely determined, and we cannot guarantee that replicas will converge to the same state.

B Proofs for Impossibility Result

We first prove the equivalence between (1) weak forward interleaving, (2) forward interleaving as defined by Kleppmann et al. [15, §4], and (3) the condition that the list order is a depth-first pre-order traversal over the left-origin tree.

Proof of Proposition 3. (3) \implies (2): This follows by the same proof as for RGA [15]. Briefly: sequences inserted concurrently in the forward direction end up in different subtrees, each rooted at the sequence's first element, so the traversal visits one entire sequence before the other.

(2) \implies (1): Take $a_1 \dots a_m$ to be the single element a .

(1) \implies (3): First, we prove the following claim. Let a and b_1, \dots, b_n be as in the definition of weak forward non-interleaving, except that a was not necessarily inserted concurrently to all b_j . Then the conclusion still holds: in the final list order, either $a < b_1, \dots, b_n$ or $b_1, \dots, b_n < a$.

If $a < b_1$, then the claim is obvious, since $b_1 < \dots < b_n$ by the chain of left origins.

Else $b_1 < a$. Since $a.\text{leftOrigin} = b_1.\text{leftOrigin} < b_1 < a$, a must have been unaware of b_1 when a was inserted. Thus a is concurrent to or causally prior to b_1 . Since all b_j are causally greater than b_1 (by the chain of left origins), a is likewise concurrent to or causally prior to all b_j .

Now there must be an index j (possibly 0 or n) such that a is concurrent to b_1, \dots, b_j and causally prior to b_{j+1}, \dots, b_n . By weak forward non-interleaving, $b_1, \dots, b_j < a$. Next, when b_{j+1} was inserted, it was aware of both b_j and a , but chose b_j as its left origin instead of a ; thus it was inserted to the left of a , i.e., $b_{j+1} < a$. The same holds for the rest of b_{j+2}, \dots, b_n , proving the claim.

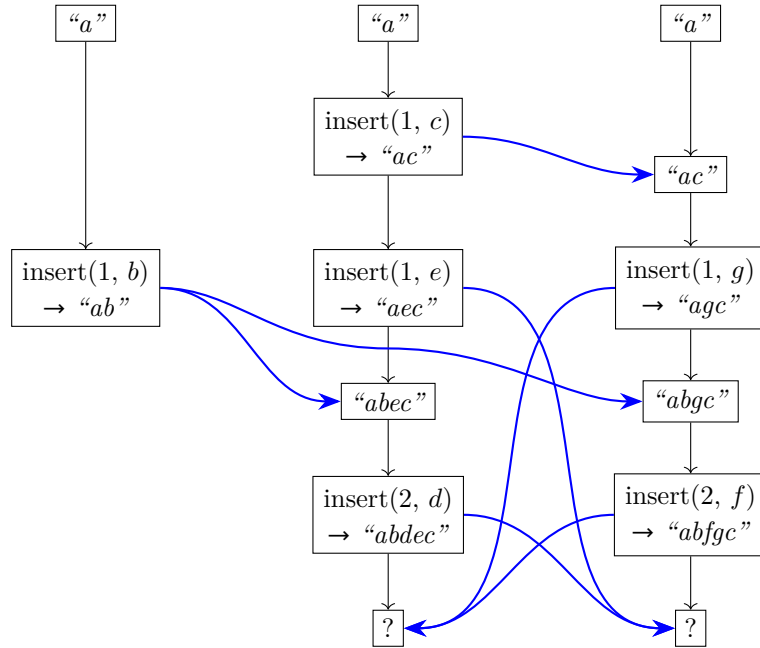
Next, to prove that the list order is a depth-first pre-order traversal over the left-origin tree, it suffices to prove:

- (a) Each element is greater than its parent in the tree.
- (b) If a and b are siblings in the tree and $b < a$, then the entire subtree rooted at b is less than a .

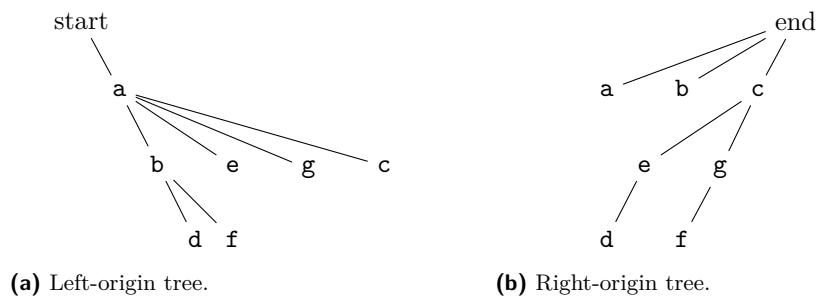
Statement (a) holds because each element's parent is its left origin, which is lesser by definition. Statement (b) follows from the above claim with $b_1 = b$: as siblings, a and b have the same left origin; and any descendant d of b is part of a chain left origins $b = b_1, b_2, \dots, b_n = d$, hence is on the same side of a as b ($d < a$). ◀

Next, we prove that it is impossible to satisfy both forward and backward interleaving.

Proof of Theorem 5. We give a counterexample. Figure 7 shows the flow of operations in the example, and Figure 8 shows the final left-origin and right-origin trees.



■ **Figure 7** Visualization of the example in the proof of Theorem 5.



■ **Figure 8** Left- and right-origin trees for the example in the proof of Theorem 5.

The document starts as a . Concurrently, one user types b after a (yielding ab), while another types c after a (ac). WLOG assume that $b < c$ in the final document order.

After receiving c but not b (state ac), one user types e between a and c (aec), while another types g (agc). Both users then receive b . Note that b and c have the same right origins (end), $b < c$, and e, g have right origin c ; thus by backward non-interleaving, $b < e, g$. So, the two users see $abec$ and $abgc$.

Next, the user with $abec$ types d between b and e ($abdec$), while concurrently, the user with $abgc$ types f between b and g ($abfgc$).

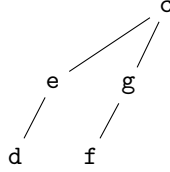
Finally, they merge their changes. Note that b and e have the same left origins (a), $b < e$, and d, f have left origin b ; thus by forward non-interleaving, $d, f < e$. Likewise, $d, f < g$. So, we have

$$a < b < (d, f) < (e, g) < c.$$

The allowed final orders according to forward non-interleaving are then

$$abdfegc \quad abdfgec \quad abfdegc \quad abfdgec$$

All of these orders interleave de with fg . But this is forbidden by backward non-interleaving: the right-origin tree contains the subtree



and so the final order on $\{d, e, f, g\}$ according to backward non-interleaving must be either $defg$ or $fgde$. Hence we cannot satisfy both forward and backward non-interleaving at the same time. ◀

C Proof that Tree-Fugue is Maximally Non-Interleaving

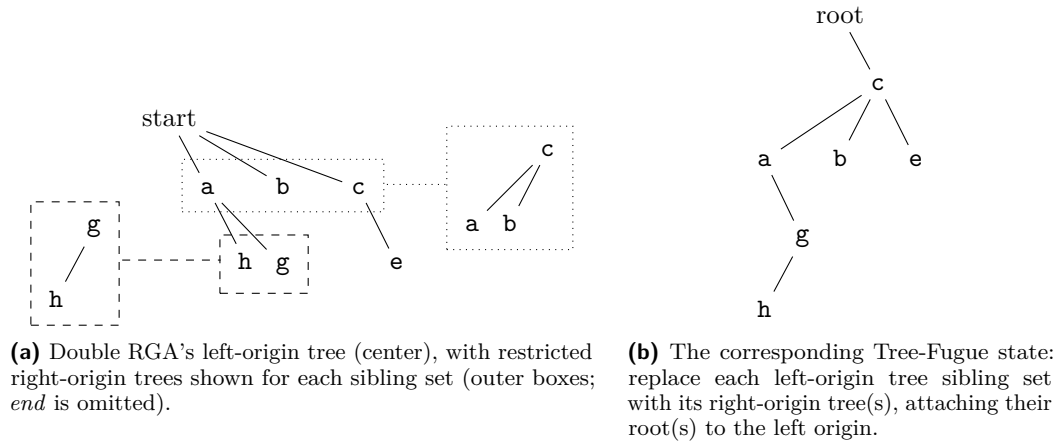
Let S be a set of siblings in the left-origin tree, and let $T_R|_S$ be the rooted tree as defined in Section 5.2. Recall from Section 5.2 that maximal non-interleaving almost completely determines $<$, the total order on list elements. The only degree of freedom is: for each set of siblings S in the left-origin tree, for each set of siblings S' within $T_R|_S$, choose an arbitrary total order on S' that is consistent across replicas.

To formalize this idea, define *double siblings* to be members of the same S' , i.e., nodes that are siblings in both the left-origin tree and in the restricted right-origin tree $T_R|_S$. Define a *tiebreaker* to be an algorithm that assigns a consistent total order to each set of double siblings. For example, the *lexicographic tiebreaker* sorts elements lexicographically by ID.

► **Definition 10** (Double RGA). *Given a tiebreaker B , Double RGA with tiebreaker B is the list CRDT defined by:*

State *The set of all previously-inserted elements including tombstones, each labeled by a unique ID, its left origin, and its right origin.*

Total order *The depth-first pre-order traversal of the left origin tree. Siblings S are sorted using the depth-first post-order traversal of $T_R|_S$, the right origin tree restricted to S . Siblings within $T_R|_S$ (i.e., double siblings) are sorted using the tiebreaker B .*



■ **Figure 9** Example conversion from Double RGA to Tree-Fugue.

Insertions The generator broadcasts a new element together with its id, left origin, and right origin; the effector adds the element to the state.

Deletions Implemented using tombstones, like in *Tree-Fugue* (Algorithm 1).

We choose the name “Double RGA” because it traverses the left-origin tree like RGA, except instead of sorting siblings by descending s4vector or Lamport timestamp, it sorts them via a second, reversed RGA.

► **Lemma 11.** *For any tiebreaker B , Double RGA with tiebreaker B is maximally non-interleaving. Conversely, any maximal non-interleaving replicated list is semantically equivalent to Double RGA with some tiebreaker. That is, in any execution, they induce the same (implicit global) order $<$ on elements.*

Proof. This is obvious from the definition of maximal non-interleaving. ◀

The choice of tiebreaker is arbitrary, hence we do not restrict it. Indeed, non-deleted double siblings must all have been inserted concurrently at the same position. There is no clear semantic reason to prefer any order of such elements, although an application could use a custom tiebreaker if it has a need for ensuring a particular order on concurrent insertions at the same position.

► **Lemma 12.** *Tree-Fugue is semantically equivalent to Double RGA with the lexicographic tiebreaker. That is, in any execution, they induce the same (implicit global) order $<$ on elements.*

Proof. Given the left- and right-origin trees corresponding to an execution, we can construct a Tree-Fugue ordered tree as follows (see Figure 9 for an example).

Start with the left-origin tree. Replace each sibling set S with $T_R|_S$ (the right-origin tree restricted to S), except using S ’s left origin as the root instead of *end*. In other words, turn each child of the root in $T_R|_S$ into a child of S ’s parent. In the resulting tree, edges inherited from the left-origin tree become right child relationships (left parents), while edges inherited from the right-origin tree become left child relationships (right parents).

Consider the in-order traversal of this tree in which siblings are sorted lexicographically. One can check that this yields nodes in the same order as Double RGA with the lexicographic tiebreaker. Furthermore, by induction on insert operations, one can check that this tree is identical to Tree-Fugue’s implicit global tree. ◀

More generally, we can use any tiebreaker B to sort same-side siblings within Tree-Fugue. That gives a list CRDT equivalent to Double RGA with tiebreaker B . Indeed, Double RGA's double siblings correspond exactly to Tree-Fugue's same-side siblings.

It follows that Tree-Fugue is maximally non-interleaving:

Proof of Theorem 7. Follows immediately from Lemma 11 and Lemma 12. ◀

D Proof that List-Fugue is Equivalent to Tree-Fugue

Proof of Theorem 8. We will actually prove that List-Fugue is semantically equivalent to Double RGA with the lexicographic tiebreaker, where Double RGA is the list CRDT defined in Appendix C. This suffices by Lemma 12.

So, let $<$ be Double RGA's total order when using the lexicographic tiebreaker. When we use the words “greater” or “lesser”, we are referring to this order. Let $list$ be the state of some List-Fugue replica at some point during the execution, and let elt be the next element received by this replica (i.e., delivered to its insert effector). Inductively, we may assume that $list$ is already ordered by $<$; we must prove that the replica inserts elt into $list$ at the location specified by $<$.

Specifically, let $left'$ denote the element immediately preceding elt in $list \cup \{elt\}$ when sorted by $<$. We must prove that by the last line of the insert effector, $left = left'$.

Observe that by the inductive hypothesis, elt 's left and right origins are the same in the Double RGA and List-Fugue executions. One can also assume that elt has the same ID in both executions. Thus we can refer to these unambiguously.

Like in the definition of maximal non-interleaving (Section 5.2), let S be the set of left-origin-siblings of elt that are present in $list \cup \{elt\}$, and let $T_R|_S$ be the tree on $S \cup \{end\}$ derived from the right-origin tree. Also partition $S = S_< \sqcup \{elt\} \sqcup S_>$, where $S_<$ and $S_>$ are the elements less and greater than elt , respectively.

Observe that for $p \in S$, $rightParent(p)$ returns the parent of p in $T_R|_S$, which may differ from p 's right origin. We will refer to this as p 's *right parent* below.

Case $S_< = \emptyset$: Recall that $<$ is a depth-first pre-order traversal over the left-origin tree. Thus in this case, $left' = elt.leftOrigin$. We must verify that the for-loop in the insert effector terminates before it ever sets $left \leftarrow o$.

If $S_> = \emptyset$, then the first element o_1 that the for-loop visits is the next element after $elt.leftOrigin$'s left-origin-subtree. So in the left-origin tree, o_1 is a sibling of $elt.leftOrigin$ or a sibling of an ancestor of $elt.leftOrigin$. Then its parent is a left-origin ancestor of $elt.leftOrigin$, implying $o_1.leftOrigin < elt.leftOrigin$. Hence the loop ends at the first break statement (line 26).

Else $S_> \neq \emptyset$. Then the first element o_1 that the for-loop visits is the least element of $S_>$. Recall that S is ordered by a depth-first post-order traversal of $T_R|_S$; thus o_1 the next element after elt in $T_R|_S$. This implies that o_1 is either (1) elt 's right parent, (2) a greater sibling of elt in $T_R|_S$, or (3) a right-origin-descendant of such a sibling. In these cases:

- (1) The for-loop's range ends before o_1 , since $elt.rightOrigin \leq rightParent(elt) = o_1$.
- (2) o_1 is a double sibling of elt and is greater in Double RGA's list order. By the definition of double sibling, the loop enters the block on lines 31–32. Also, since $o_1 > elt$ and Double RGA sorts double siblings lexicographically by ID, we must have $o_1.id > elt.id$. Hence the for-loop ends at the second break statement (line 31).
- (3) We have $rightParent(o_1) < rightParent(elt)$, since $rightParent(elt)$ is an ancestor of $rightParent(o_1)$ in $T_R|_S$. Also, $o_1.leftOrigin = elt.leftOrigin$ because they are left-origin siblings, so the for-loop sets $scanning \leftarrow \text{true}$ on line 29. Then this iteration does *not* set

$left \leftarrow o_1$ at the end. Subsequent iterations visit left-origin-descendants o of o_1 ; these iterations have $o.leftOrigin \geq o_1 > elt.leftOrigin$ and $scanning = \text{true}$, so they do nothing. Eventually, the for-loop visits o_1 's parent in $T_R|_S$, which either falls into case (2) or (3). Repeat the argument until reaching case (2).

Case $S_< \neq \emptyset$: Let $pred$ be the greatest element of $S_<$. In this case, $left'$ is the greatest left-origin-descendant of $pred$. We must verify that:

- (a) The for-loop in the insert effector does not terminate before or during the $left'$ iteration.
- (b) When $o = left'$, the for-loop sets $left \leftarrow o$.
- (c) After $left'$, the for-loop terminates before it sets $left \leftarrow o$ again.

Claim (c) is proved similarly to the case $S_< = \emptyset$, except that o_1 is the *next* element visited after $left'$ instead of the *first* element visited.

Claim (a) essentially follows by reversing the argument for Claim (c): one can check that whenever the for-loop terminates, necessarily $o > elt > left'$.

To prove claim (b), we first argue that when $o = pred$, the for-loop sets $scanning \leftarrow \text{false}$. Indeed, we have $pred.leftOrigin = elt.leftOrigin$ and $pred < elt$, so $pred$ precedes elt in the depth-first post-order traversal of $T_R|_S$. Thus either:

- $pred$ is a lesser sibling of elt in $T_R|_S$, i.e., a lesser double sibling. In this case, the loop enters the block on lines 31–32 and $pred.id < elt.id$, so line 32 sets $scanning \leftarrow \text{false}$.
- Or, elt has no lesser siblings in $T_R|_S$. In this case, $\text{rightParent}(pred) > \text{rightParent}(elt)$, so line 34 sets $scanning \leftarrow \text{false}$.

Next, iterations immediately after $pred$ traverse the left-origin-descendants o of $pred$. Such descendants have $o.leftOrigin \geq pred > elt.leftOrigin$, so their iterations leave $scanning = \text{false}$ and set $left \leftarrow o$. In particular, the iteration with $o = left'$ sets $left \leftarrow left'$. ◀

E Characterizing Yjs

The Yjs library's list CRDT implementation is used in a number of production apps and has state-of-the-art performance [10, 9]. In this section, we characterize its underlying algorithm's semantics and interleaving guarantees.

Specifically, we show that Yjs's total order is a depth-first pre-order traversal over the left-origin tree—similar to RGA but with a different sibling sort order. Using this result, we give a new proof that Yjs is correct, and we give the first proof that it is forward non-interleaving. We also give a counterexample to show that it is *not* maximally non-interleaving.

The results in this section are adapted from our proof that List-Fugue is equivalent to Tree-Fugue (Appendix D).

E.1 Yjs Algorithm

Algorithm 3 shows the algorithm underlying Yjs's list CRDT; we will refer to this algorithm also as Yjs. The algorithm is paraphrased from the Yjs library's source code, omitting optimizations and implementation details.

Except for the insert effector, Yjs is identical to List-Fugue. In particular, each replica's state is a list of elements, and a replica inserts a new element into its local list “at the correct location” using a for-loop. Deletions use tombstones.


```

1  update insert
2    effector (insert, elt)
3      left  $\leftarrow$  elt.leftOrigin
4      conflicts  $\leftarrow$   $\emptyset$ ; before  $\leftarrow$   $\emptyset$ 
5      for o in list from elt.leftOrigin to elt.rightOrigin, exclusive do
6        before  $\leftarrow$  before  $\cup$  {o}
7        conflicts  $\leftarrow$  conflicts  $\cup$  {o}
8        if elt.leftOrigin = o.leftOrigin then
9          if o.id.replicaID < elt.id.replicaID then
10             left  $\leftarrow$  o; conflicts  $\leftarrow$   $\emptyset$ 
11          else if elt.rightOrigin = o.rightOrigin then
12             break
13          else if o.leftOrigin  $\in$  before then // Equiv., o.leftOrigin > elt.leftOrigin
14             if o.leftOrigin  $\notin$  conflicts then
15                 left  $\leftarrow$  o; conflicts  $\leftarrow$   $\emptyset$ 
16             else // o.leftOrigin < elt.leftOrigin
17                 break
18      Insert elt into list immediately after left

```

■ **Algorithm 3** Yjs insert effector; the rest of the algorithm is as in List-Fugue (Algorithm 2). Paraphrased from `Item.js` lines 427–481 in commit 31b4ab.

E.2 Tree-Based Total Order

We claim that Yjs’s list order on each replica is a specific depth-first pre-order traversal of the left-origin tree.

First, we must specify the total order that a Yjs replica uses to sort siblings in the left-origin tree. Unfortunately, this order is hard to describe in a global (replica-independent) way. Instead, Algorithm 4 gives an algorithm that a specific replica can use during an execution of Yjs to determine its own sort order on the children of any given element *r* (or *r* = *start*); we call this the *Yjs sibling sort*. We later prove that all replicas converge to the same Yjs sibling sort (Lemma 14).

```

1  per-replica state for parent r:
2    sibs: a list of the elements with left origin r
3  when effecting Yjs message (insert, elt)
4    p  $\leftarrow$  greatest element of sibs s.t. p < elt.rightOrigin and
       p.id.replicaID < elt.id.replicaID, or null if there is no such element
5    Insert elt into sibs immediately after p, or at the beginning if p = null

```

■ **Algorithm 4** Algorithm that recursively defines Yjs’s sibling sort on the left-origin-children of a list element *r* (or *r* = *start*) during an execution of Yjs, for a specific replica. Note that *p* < *elt.rightOrigin* recursively references the total order < on the replica’s entire list (not just *sibs*), since *elt.rightOrigin* might not be in *sibs*.

In the context of Algorithm 4, when inserting an element *elt* into *sibs*, we call *p* on line 4 the *placeholder* for *elt*. We call all the values considered on line 4 *candidate placeholders*. Note that the placeholder is the greatest candidate placeholder.

We now give our tree-based description of Yjs’s total order.

► **Proposition 13.** *Each Yjs replica sorts elements according to the total order $<$ defined by the depth-first pre-order traversal of the left-origin tree using that replica's Yjs sibling sort.*

Proof. Fix a replica. Let $<$ be the described total order. When we use the words “greater” or “lesser”, we are referring to this order. Let *list* be the state of the Yjs replica at some point during the execution, and let *elt* be the next element received by this replica (i.e., delivered to its insert effector). Inductively, we may assume that *list* is already ordered by $<$; we must prove that the replica inserts *elt* into *list* at the location specified by $<$.

Specifically, let $left'$ denote the element immediately preceding *elt* in $list \cup \{elt\}$ when sorted by $<$. We must prove that by the last line of the insert effector, $left = left'$.

For each element *o* visited by the loop, we can characterize the top-level if-...-else block's cases as follows.

Case 1 (Line 8): The first case, $o.leftOrigin = elt.leftOrigin$, occurs if and only if *o* is a (left-origin-tree) sibling of *elt*. We claim that this case sets $left \leftarrow o$; $conflicts \leftarrow \emptyset$ if and only if *o* is a candidate placeholder for *elt* (during *elt*'s insertion into *list*). Since the loop terminates at $elt.rightOrigin$, we know $o < elt.rightOrigin$; thus *o* is a candidate placeholder if and only if $o.id.replicaID < elt.id.replicaID$. This matches the inner if-statement, proving the claim.

Case 3 (Line 16): The third case is easily seen to be equivalent to $o.leftOrigin < elt.leftOrigin$, as noted in the comment. The depth-first pre-order traversal's first element after *elt*'s subtree satisfies this, since it is a sibling of $elt.leftOrigin$ or one of its ancestors. Meanwhile, all (left-origin-tree) descendants *d* of *elt* have $d.leftOrigin \geq elt.leftOrigin$. Thus this case is triggered precisely when *o* is the first element in the traversal after *elt*'s subtree, at which point it breaks out of the loop.

Case 2 (Line 13): The second case, $o.leftOrigin \in before$, occurs if and only if *o* is a (left-origin-tree) child of a previously-visited value. By the previous paragraph, all previously-visited values were descendants of $o.leftOrigin$. Hence this case occurs if and only if *o* is a descendant of $o.leftOrigin$ that is not a sibling of *elt*. Let *s* be the unique sibling of *elt* that is an ancestor of *o*. One verifies by induction and Case 1 that $o.leftOrigin \notin conflicts$ (hence $left \leftarrow o$; $conflicts \leftarrow \emptyset$) if and only if *s* is a candidate placeholder for *elt*.

In summary, we have proven that the loop sets $left \leftarrow o$ precisely when *o* is a candidate placeholder or its descendant. Meanwhile, the correct value $left'$ is the greatest descendant of the greatest candidate placeholder, or $elt.leftOrigin$ if there are no candidate placeholders. One checks that $left'$ is indeed returned unless the loop terminates before it is visited.

It remains to show that when the loop terminates, $left'$ has already been visited. The three termination scenarios are:

- Upon reaching $elt.rightOrigin$, which is always greater than $left'$.
- Upon reaching Case 3's break statement (line 17). We proved above that this only happens after visiting the entire subtree rooted at $elt.leftOrigin$, which includes $left'$.
- Upon reaching the break statement on line 12. In this case, *o* is a sibling with $o.rightOrigin = elt.rightOrigin$ and $o.id.replicaID \geq elt.id.replicaID$. Then *o*'s set of candidate placeholders contains all of *elt*'s candidate placeholders, implying $left' < o$.

◀

E.3 Yjs Properties

Finally, we prove properties of the Yjs algorithm.

► **Lemma 14.** *The Yjs sibling sort (Algorithm 4) is eventually consistent. That is, for any execution of Yjs, for any element r (or $r = \text{start}$), two replicas that have effected the same insert messages have identical states sibs for r .*

Proof. Let σ be a state (i.e., a sibs value), and let a and b be concurrently-inserted left-origin-children of r . WLOG $a.\text{id.replicaID} < b.\text{id.replicaID}$. Assume that their insert messages (insert, a) or (insert, b) are both enabled in state σ , i.e., σ does not contain a or b but does contain all causally prior elements, so that a replica in state σ is allowed to insert either element. By an argument traditionally used for operation-based CRDTs [27, Proposition 2.2], to prove eventual consistency, it suffices to prove that (insert, a) and (insert, b) commute in their effect on sibs : inserting the elements in either time order gives the same list order.

First consider inserting a into σ . Let p_a be a 's placeholder during this insertion. Likewise, let p_b be b 's placeholder during its insertion into σ (not into σ with a already inserted).

Case $p_a = p_b$: Inserting b and then a results in the subsequence $p_a ab$: $p_a \rightarrow p_a b \rightarrow p_a ab$, since both elements are inserted immediately after $p_a = p_b$. Meanwhile, inserting a and then b also results in the subsequence $p_a ab$: a is inserted after p_a like in σ , but then b is inserted after a , since a wins over $p_a = p_b$ as b 's greatest candidate placeholder. Indeed, $a < b.\text{rightOrigin}$ because $p_a = p_b < b.\text{rightOrigin}$ and a is inserted immediately after p_a , $a.\text{id.replicaID} < b.\text{id.replicaID}$ by assumption, and $a > p_a = p_b$.

Case $p_a \neq p_b$: Inserting b and then a results in two non-overlapping subsequences $p_a a$ and $p_b b$. Inserting a and then b does likewise, unless a wins over p_b as b 's greatest candidate placeholder. Assume that this occurs; we will derive a contradiction.

Since a wins over p_b , we have $p_b < a < b.\text{rightOrigin}$. Now $p_b < a$ implies $p_b < p_a$, since a was inserted immediately after p_a and $p_a \neq p_b$. In particular, $p_a \neq \text{null}$, i.e., a was not inserted at the beginning of σ . Then $p_a.\text{id.replicaID} < a.\text{id.replicaID} < b.\text{id.replicaID}$. Together with $p_a < a < b.\text{rightOrigin}$, this implies that p_a is a candidate placeholder for b in σ . But $p_a > p_b$, contradicting the fact that p_b is b 's greatest candidate placeholder in σ . ◀

► **Theorem 15.** *Yjs satisfies the strong list specification [1], and it is forward non-interleaving.*

Proof. For the strong list specification, Proposition 13 and Lemma 14 describe the total order that Yjs uses to sort elements and show that it is consistent across replicas. Also, it is easy to check that Algorithm 3 always inserts a new element between its left and right origins, as required by the specification.

Forward non-interleaving follows from Proposition 3 and the fact that Yjs is equivalent to a depth-first pre-order traversal of the left-origin tree. ◀

► **Proposition 16.** *Yjs is not maximally non-interleaving.*

Proof. We give an example of backwards interleaving in which all elements have the same left origin, analogous to the code example in Appendix A.1.9.

Starting from an empty list, replica 3 inserts b . After receiving this insert, replica 1 inserts a before b , yielding ab . Concurrently to both operations, replica 2 inserts x into its (empty) list. Finally, all replicas receive all inserts.

Let $S = \{a, b, x\}$ be the left-origin children of start . Maximal non-interleaving mandates a depth-first post-order traversal of $T_R|_S$; the two such traversals are abx and xab . However, Yjs yields axb : when replica 1 receives x in state ab , x 's placeholder is a , since $a.\text{id.replicaID} < x.\text{id.replicaID} < b.\text{id.replicaID}$. ◀