Designing a commutative replicated data type for cooperative editing systems

Nuno Preguiça*, Marc Shapiro#

Technical report 2-2008 DI-FCT-UNL

INRIA Paris-Rocquencourt & LIP6 Paris, France

* CITI/DI
Faculdade de Ciências e Tecnologia
Universidade Nova de Lisboa
Quinta da Torre, 2829-516 Caparica, Portugal

Designing a commutative replicated data type for cooperative editing systems*

Nuno Preguiça
CITI/DI, FCT, Univ. Nova de Lisboa

Marc Shapiro
INRIA Paris-Rocquencourt & LIP6

Abstract

Cooperative editing systems enable distributed users to collaborate by concurrently editing a shared document, but care is required to ensure convergence and correctness. This problem turns out to be surprisingly complex. In this paper, we present a new approach for managing replicated data in cooperative editing systems, Commutative Replicated Data Types (CRDT). In a CRDT, all concurrent operations commute. Along with causal ordering, this suffices to guarantee convergence and preserve correctness, requiring no further concurrency control. We propose a novel CRDT for concurrent editing, the treedoc, which requires considerably less storage than the state of the art. Furthermore, a novel technique, background consensus, is used to clean up redundant metadata without interfering with ordinary edit operations. This mechanism is generic, and could be used to improve other algorithms.

1 Introduction

Cooperative editing systems are an important class of groupware applications. These systems allow users to cooperate by editing a shared document concurrently in synchronous (e.g. Groove [6], CoWord [33] and Google Docs [8]) or asynchronous mode (e.g. CVS [3]). These systems maintain a replica of the document at every participating site.

As user can edit the document concurrently, such systems must control concurrency in order to guarantee that replicas eventually converge to a common state. Additionally, they must ensure correctness, i.e., that users' "intentions" are preserved [28].

*This work is supported in part by the EU FP6 project Grid4All, the French ARA project Respire, the French ARC project Recall, and the Portuguese FCT/MCTES project POSC/59064/2004, with FEDER funding.

Various solutions have been proposed in the literature. Pessimistic approaches, such as turn-taking or serialising the actions [11], avoid replica divergence, by allowing only one user to edit (some unit of) the document at a given time. This is considered unsuitable for collaborative editing, since it restricts cooperation. This problem is even worse in asynchronous or disconnected settings, because an unavailable user might block all other users.

Optimistic approaches allow replicas to temporarily diverge but guarantee eventual convergence [26]. The optimistic approahc of operational transformation (OT) has become the solution of choice in the groupware community [6, 29, 28, 15, 18, 31, 14, 16]. However, this approach is complex and error-prone, as problems found in previously proposed algorithms show [19].

In this paper, we suggest a different approach: design replicated data types such that concurrent operations commute with one another. Let us call such a type a *commutative replicated data type* or CRDT. CRDT replicas trivially converge. However, designing a non-trivial CRDT is difficult.

Although the advantages of commutativity are well known, the problem of designing data types for commutativity has been neglected. Recently, Oster et al. proposed a replicated character buffer CRDT called WOOT [20]. WOOT operations commute, because updates are non-destructive, and because the identity of a character does not change with concurrent edits. However, WOOT consumes a lot of storage, because it has a high metadata overhead, and because deleted text cannot be discarded.

This paper presents the design of a novel, non-trivial CRDT for concurrent editing, called *treedoc*. The treedoc data structure is a binary tree that supports non-destructive edits and invariant identification. Since it is a CRDT, convergence is guaranteed. We also prove that it preserves user intention.

We extend the binary tree structure to support concurrency and to save space by flattening the tree. To ensure such structural operations commute with edits, edits have precedence. Structural changes require a consensus, but we push the associated latency into the background, off the critical path of edit operations.

The treedoc design is simple and elegant; meta-data overhead is low; deleted information can be forgotten; and identifiers are kept short. Common edit operations respond locally and suffer no network latency.

Treedoc supports block operations, but this is out of the scope of the current paper; we refer the interested reader to the companion technical report [1].

In summary, the contributions of this paper are the following:

- We present the Commutative Replicated Data Type (CRDT) concept, a data type for which concurrent operations commute. CRDT replicas are guaranteed to converge, under simple and standard assumptions.
- We identify two alternative approaches to commutativity: genuine vs. precedence. Genuine is better, but is not always possible. Genuine commutativity preserves user intentions.

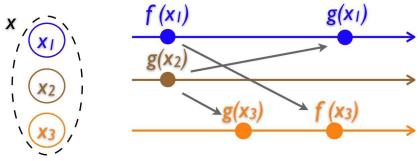

Figure 1: Three sites with replicas of object x. Site 1 initiates operation f, Site 2 initiates g. Site 3 replays them in the order $g; f$, whereas Site 1 replays g after f.

- We present the design of treedoc, a non-trivial, space-efficient, responsive CRDT for distributed editing. Edit operations are genuinely commutative.
- We notice that structural operations require consensus but have lesser precedence than edits. The consensus is moved into the background, and aborts if it conflicts with an essential operation.

The paper proceeds as follows. We describe our system model in Section 2. Section 3 describes the shared buffer abstract data type. We suggest a simple implementation of this data type in Section 4. The design is extended to support concurrent inserts in Section 5. In Section 6, we examine how to convert to an even more efficient representation and back. Section 7 compares with previous work. Section 8 compares the costs of treedoc with WOOT. Section 9 concludes the paper.

2 System model

2.1 Replicated execution and eventual consistency

We consider an distributed system, consisting of N *sites* (computers) connected by a network, storing replicas of shared *objects* (e.g., object x replicated at three sites in Figure 1). The system is asynchronous, i.e., we do not assume an upper bound on message delay.

A user accesses an object through his local replica, *initiating* operations at the current site. After executing at the initiator site, operations are reliably transmitted to the other sites. Eventually the operation is delivered and *replayed*. Thus, all sites execute the same operations (either by local submission or by remote replay), in some sequential order, but not necessarily in the same order.

We say operation o happens before o' (noted $o \rightarrow o'$) if some site initiates o' after the same site has executed o.[1] We require that if $o \rightarrow o'$, then all sites execute o before o' (although not necessarily immediately before). This is sometimes called the "causal ordering" property. Causal ordering is a standard

[1] The site executes o either because it was initiated locally, or because it was initiated at another site and delivered here in a message. Therefore, the $\rightarrow$ relation is identical to Lamport's happens-before [11].

requirement of cooperative editing systems, and techniques for ensuring it are well known [11, 21].

Operations are *concurrent* if neither happens before the other: $o \parallel o' \stackrel{\text{def}}{=} \neg(o \rightarrow o') \wedge \neg(o' \rightarrow o)$.

Informally, two operations *commute* iff executing them in either order from the same state leads to the same state. A *Commutative Replicated Data Type* (CRDT) is a data type where all concurrent operations commute with one another.

We prove that CRDTs guarantee *eventual consistency*: if CRDT operations execute in some order consistent with happens-before, the final state of replicas is identical at all sites. Execution order may differ between sites. For space reasons, we could not include the proof in the paper, but it is available in a companion technical report [1]. The proof relies on the fact that non-concurrent operations execute in the same order at all sites, and confirms the intuition that concurrent operations may execute in any order, since they commute.

2.2 Genuine commutativity vs. precedence

Consider some arbitrary state T, and operations α and β, such that execution sequences $\langle T \cdot \alpha \rangle$ and $\langle T \cdot \beta \rangle$ are both correct states. α and β commute, iff, for any such T, execution sequences $\langle T \cdot \alpha \cdot \beta \rangle$ and $\langle T \cdot \beta \cdot \alpha \rangle$ are both correct states and are equivalent.[2]

There are two basic approaches to ensuring this. We say operations commute genuinely if each one preserves the effect of the other; i.e., the post-condition of both operations is satisfied, whatever their relative execution order. This is the meaning in mathematics, for instance when we say that addition and subtraction of integers commute.

An alternative is to define a precedence, i.e., one operation masks or destroys the effect of the other. Thus, if β takes precedence over α: *(i)* in the order $\langle \alpha \cdot \beta \rangle$, β overwrites the results of α; but *(ii)* in the order $\langle \beta \cdot \alpha \rangle$, α reduces to a no-op. For instance, most replicated file systems follow the "Last Writer Wins" rule [26]: when two users write to the same file, the write with the highest timestamp takes precedence. The write with the lowest timestamp may be lost.

Technically, precedence satisfies the formal definition of commutativity, but it violates our intuition because the effect of the lesser operation is lost.

We make the standard assumption that objects are independent, hence operations over different objects always commute genuinely. Therefore, without loss of generality, we may consider a single object. For instance, replicated file systems assume that files are independent.

Clearly, the genuine approach is preferable to precedence; but precedence is often easier to achieve. In our proposal, edit operations commute genuinely but have precedence over internal clean-up operations.

[2] We assume that an operation that fails at the initiator has no effect, and is not propagated nor replayed. Conversely, an operation that succeeds at the initiator must also succeed at every replay site.

2.3 Intention preservation

In this section we give a precise definition of intention preservation in concurrent editing.

Edit operations operate over a document abstraction, which is an ordered, sequential list of *atoms* (e.g., characters). Each user has a replica of the document, and can modify it locally by initiating an edit operation. If successful, this operation is subsequently transmitted to all remote replicas and eventually replays there. We define two edit operations: $insert(insertpos, newatom)$ adds atom *newatom* into the list at position *insertpos*. Operation $delete(delpos)$ removes the atom at position *delpos* from the list.

While we defer a precise definition of position until later, we may still define operation semantics in terms of post-conditions. On the initiator site, any atom that is to the left of *insertpos*, remains to the left of *newatom* after executing $insert(insertpos, newatom)$. Similarly for any atom to its right. On the initiator site, there is an atom at *delpos* before $delete(delpos)$, and that atom is removed thereafter.

We wish edit operations to have the same effect at replay sites. If an atom was to the left of *insertpos* at *the initiator site*, then that same atom remains to the left of *newatom* at any *replay site* after it replays $insert(insertpos, newatom)$. Similarly at the right. The atom that was at *delpos* at *the initiator site* is removed from that position *at replay sites* after replaying $delete(delpos)$.

It should be clear that our definition of intent preservation is just a special case of genuine commutativity.

3 Generic shared buffer data type

We consider a shared, replicated document, consisting of a linear sequence of elements called *atoms*. An atom may be a character or some other non-editable element, e.g., a graphics file inserted inside the document [30].

3.1 Unique identifiers for positions

Each atom has an associated unique position identifier (UID), with the following properties: *(i)* Each position in the atom buffer has an identifier; no two different atoms have equal identifiers; an identifier remains constant for the whole lifetime of the document.[3] *(ii)* There is a total order of position identifiers, noted $<$. *(iii)* Given arbitrary identifiers P and F such that $P < F$, a fresh (i.e., previously-unused) unique identifier N such that $P < N < F$ can be generated.

Real numbers have the properties required for UIDs, but property *iii* would require infinite precision. In Section 4 we will present a practical alternative, based on binary trees.

[3] However, an unused identifier can be garbage-collected and re-used. We do not attempt to formalise this property.

3.2 Operations

Each user has a replica of the document, and can modify it locally by initiating one of the following *edit operations*:

- *insert*(*insertpos*, *newatom*) inserts atom *newatom* into the abstract document state. In the initiator's state, the position UID *insertpos* must be free.
- *delete*(*delpos*) removes the atom at position *delpos* from the document state. In the initiator's state, there must be an atom at *delpos*.

With UIDs with the properties described in Section 3.1, this generic shared buffer design ensures genuine commutativity as proved in the next subsection, i.e., the effect of *insert* and *delete* is the same at all sites.

Note that we are ignoring semantic issues, and are considering only the syntactic level [15, 29]. At the syntactic level, treedoc edit operations commute genuinely; however, a particular application might have stronger semantic requirements. For instance, one might demand that all edits pass a spell-checker; or one might consider it illegal for one user to insert characters inside a piece of text deleted by another user; or require a proper hierarchy of chapters, sections and paragraphs; etc. The detection and resolution of such semantic conflicts have been extensively studied in the literaure [2, 23, 27]. A practical text editor would manage such semantic conflicts above the treedoc layer. This issue is out of scope of this paper.

3.3 Abstract atom buffer CRDT

Consider an abstract data type whose state T is a set of $(atom, uid)$ couples, where *uid*s are unique. The content of state T is the sequence of all *atom*s in T ordered by their *uid*. Operation $insert(a, u)$ adds the pair (a, u) to the set. If a pair (a, u) exists in the set, operation $delete(u)$ removes the pair, whatever a. We now prove that concurrent operations of this data type commute.

Lemma 1. *Insert operations commute. For any data state T, any fresh unique identifiers u_1 and u_2, any atoms a_1 and a_2, and any originating sites S_1 and S_2: $\langle T \cdot insert(a_1, u_1) \cdot insert(a_2, u_2)\rangle \equiv \langle T \cdot insert(a_2, u_2) \cdot insert(a_1, u_1)\rangle$.*

Proof. After executing the two insert operations, the resulting state includes the two new atoms. Furthermore, atoms are ordered by unique identifiers. Therefore, the final state is the same. □

Lemma 2. *An insert operation commutes with a delete operation when they refer to different unique identifiers. For any state T, any fresh unique identifier u_1, any unique identifier $u_2 \neq u_1$, any atom a_1, and any originating sites S_1 and S_2: $\langle T \cdot insert(a_1, u_1) \cdot delete(u_2)\rangle \equiv \langle T \cdot delete(u_2) \cdot insert(a_1, u_1)\rangle$.*

Proof. Two cases must be considered. First, when T includes the atom with identifier u_2. By executing both operations in any order, the final state of T

will include an additional atom identified by u_1 and it will not include the atom identified by u_2. As atoms are ordered by their unique identifier, the final state is the same. Second, when T does not include the atom with identifier u_2. By executing both operations in any order, the final state of T will include an additional atom identified by u_1, but not the atom identified by u_2, as it was not in the original state. As atoms are ordered by their unique identifier, the final state is the same. □

Lemma 3. *If an insert operation and a delete operation refer to the same unique identifier, then the insert happens-before the delete.*

Proof. According to the specification of Section 3, a user may initiate operation $delete(u)$ at site S only if a pair (u, a) exists (for some a) in the current state at site S. This pair must have been inserted by an *insert* operation executed previously at site S. □

Lemma 4. *Delete operations commute. For any state T, any unique identifiers u_1 and u_2 and any originating sites: $\langle T \cdot delete(u_1) \cdot delete(u_2) \rangle \equiv \langle T \cdot delete(u_2) \cdot delete(u_1) \rangle$.*

Proof. For any original state T, the final state will not include the atoms identified by u_1 and u_2, but it will include all other atoms, as no other atom will ever has the same unique identifier. Thus, the final state will include the same set of atoms and, as atoms are ordered by their unique identifier, the final state is exactly the same. □

Theorem 1. *The data type described in this section is a CRDT.*

Proof. By the above lemmas, all concurrent operation pairs (insert-insert, delete-delete, insert-delete) commute. □

4 Single-user treedoc

Let us now turn to practical implementation of the above abstraction. We start with a simple design that does not support concurrent insertions. In later sections we will extend our algorithms to overcome this limitation.

4.1 Paths

We manage the document as a binary tree. The left child of a node is noted 0, its right child is noted 1. A node contains either a single atom, or nil.

The identifier of an atom is its *path* in the tree, a bitstring. The path to the root is the empty bitstring ϵ; the path concatenation operator is noted $\odot$. We note $[b_1 \dots b_n]$ for $b_1 \odot \dots \odot b_n$ when there is no ambiguity (we always omit ϵ when representing paths).

For example, Figure 2 represents the document state `"abcdef"`, with the following identifiers: $id(\texttt{a}) = [00]; id(\texttt{b}) = [0]; id(\texttt{c}) = []; id(\texttt{d}) = [10]; id(\texttt{e}) = [1]; id(\texttt{f}) = [11]$.

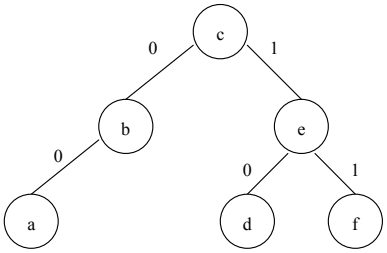

Figure 2: Identifiers in a shared text buffer, single-user version

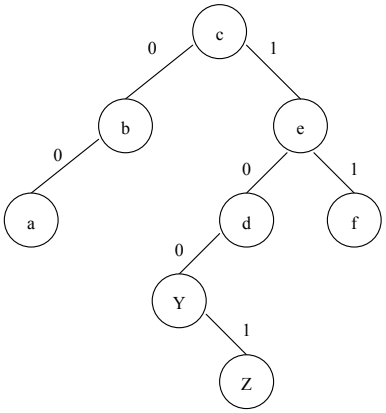

Figure 3: Identifiers after inserting two new character atoms

We define the following total order over identifiers, which results from walking the tree in infix order. Node id_1 is to the left of id_2 (or, equivalently, id_2 is to the right of id_1), noted $id_1 < id_2$, iff:

- $id_1 = [c_1...c_n]$ is a prefix of $id_2 = [c_1...c_n j_1...j_m]$ and $j_1 = 1$, or
- $id_2 = [c_1...c_n]$ is a prefix of $id_1 = [c_1...c_n i_1...i_m]$ and $i_1 = 0$, or
- $id_1 = [c_1...c_n i_1...i_n]$ has a common prefix with $id_2 = [c_1...c_n j_1...j_m]$ and $i_1 = 0$. The prefix may be empty.

We also define the ancestry of a node. Node u is the (direct) parent of node v, noted u/v, iff $id(v) = id(u) \odot 0 \vee id(v) = id(u) \odot 1$; equivalently, v is a (direct) child of u. Node u is an ancestor of v (or, equivalently, v is a descendant of u), noted $u/^+v$, if u is a parent, or grand-parent, or great-grand-parent, etc., of v.

4.2 Deleting

In this simple design, deleting an atom simply replaces its node's content with nil. Since the identification of the deleted node is unique, it is clear that the

initiator and replay executions will all delete the same node.

Later, we will discuss discarding (i.e., garbage collecting) deleted nodes. Sometimes, during replay, the node to be deleted may not exist, but this can only be because it was already deleted and discarded previously.

4.3 Insertion identifiers

To insert *newatom* between atoms P and F, we add a node to the tree. The new node has a fresh identifier, which must satisfy the relation $uid_P = id(P) < id(newatom) < uid_F = id(F)$.

Algorithm 1 New unique identifier for insert: single-user tree

```
1: function newUID (uid_p, uid_f)
2:     Require: uid_p < uid_f
3:     if ∃ a node with UID uid_m such that uid_p < uid_m < uid_f then return
   newUID(uid_p, uid_m)
4:     else if uid_p /+ uid_f then return uid_f ⊙ 0
5:     else if uid_f /+ uid_p then return uid_p ⊙ 1
6:     else  return uid_p ⊙ 1
```

Calling $newUID(uid_P, uid_F)$ generates fresh a new UID for inserting between P and F. Algorithm 1 recursively searches for two consecutive nodes p and f such that $uid_P \leq uid_p < uid_f \leq uid_F$. Then it allocates a fresh identifier, either $uid_f \odot 0$ or $uid_p \odot 1$ (heuristically).

In the example of Figure 2, to insert atom `Y` between `c` and `d`, a left child [100] is created under `d`. Thereafter, inserting `Z` between `Y` and `d` creates a right child [1001] under `Y`. This is illustrated in Figure 3.

5 Multi-user treedoc

A binary tree is insufficient if two users can concurrently insert an atom at the same position. To address this issue, we maintain the basic binary tree structure, but extend a node to contain any number of internal *mini-nodes*. A node containing mini-nodes will be called a major node, or just node when there is no ambiguity.

Inside a major node, mini-nodes are identified by a *disambiguator*. Disambiguators are assumed unique within their major node and ordered.

The content of a major node results from an infix-order walk: the content of the major node's left child, concatenated to the mini-node contents, concatenated to the contents of the major node's right child.

Figure 4 shows a major node containing several mini-nodes, each with left and right child. Figures 5 and 6 present the examples illustrated previously, with the new structure.

We extend paths (UIDs) to include a disambiguator when necessary, i.e., *(i)* at the last element of the path; *(ii)* whenever the path follows a child of

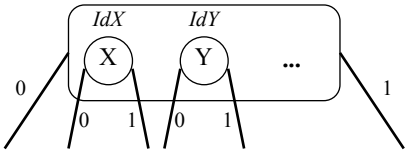

Figure 4: Major node

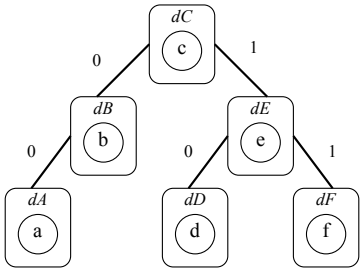

Figure 5: Identifiers in a shared text buffer: multi-user version of Figure 2

a mini-node explicitly. A path element without a disambiguator refers to the children of the corresponding major node. For example, in Figure 6, the path of atom Z is $[100(1 : idZ)]$.

We extend the total order on paths by following an infix-order walk of the tree including mini-nodes and their descendence. Thus $id_1 < id_2$, iff:

- $id_1 = c_1 \odot \ldots \odot c_n$ is a prefix of $id_2 = c_1 \odot \ldots \odot c_n \odot j_1 \odot \ldots \odot j_m$ and $j_1 = 1 \vee j_1 = (1 : d), \forall d$, or
- $id_2 = c_1 \odot \ldots \odot c_n$ is a prefix of $id_1 = c_1 \odot \ldots \odot c_n \odot i_1 \odot \ldots \odot i_m$ and $i_1 = 0 \vee i_1 = (0 : d), \forall d$, or
- $id_1 = c_1 \odot \ldots \odot c_n \odot i_1 \odot \ldots \odot i_n$ has a common prefix with $id_2 = c_1 \odot \ldots \odot c_n \odot j_1 \odot \ldots \odot j_m$ and $i_1 < j_1$.
  Given $i_1 = p_i \in \{0, 1\}$ and $j_1 = p_j \in \{0, 1\}$, we say that $i_1 < j_1$, iff: $p_i < p_j$.
  Given $i_1 = (p_i : d_i)$ and $j_1 = (p_j : d_j)$, we say that $i_1 < j_1$, iff: $p_i < p_j \vee (p_i = p_j \wedge d_i < d_j)$.
  Given $i_1 = p_i \in \{0, 1\}$ and $j_1 = (p_j : d_j)$, we say that $i_1 < j_1$, iff: $p_i < p_j \vee (p_i = p_j \wedge (\exists i_2 : i_2 = 0 \vee i_2 = (0 : d_{i2}), \forall d_{i2}))$.
  Given $i_1 = (p_i : d_i)$ and $j_1 = p_j \in \{0, 1\}$, we say that $i_1 < j_1$, iff: $p_i < p_j \vee (p_i = p_j \wedge (\exists j_2 : j_2 = 1 \vee j_2 = (1 : d_{j2}), \forall d_{j2})))$.

Mini-node u is a *mini-sibling* of v, noted $MiniSibling(u, v)$, if they are mini-nodes of the same major node.

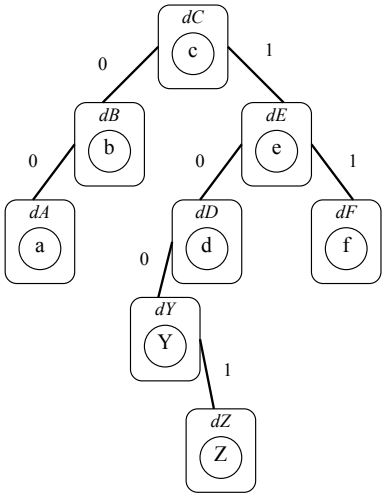

Figure 6: Identifiers after inserting two new character atoms

5.1 Generating fresh UIDs for insert: concurrent case

We extend *newUID* with mini-nodes and disambiguators. When inserting between mini-siblings, Algorithm 2 creates a child of one of the mini-siblings; otherwise it creates a child of major node. This approach keeps paths short.

Algorithm 2 New unique identifier for insert: concurrent version

1: **function** *newUID* (uid_p, uid_f)
2: // d: new disambiguator.
3: **Require:** $uid_p < uid_f$
4: **if** $\exists$ a mini-node with UID uid_m such that $uid_p < uid_m < uid_f$ **then return** $newUID(uid_p, uid_m)$
5: **else if** $uid_p /^+ uid_f$ **then** Let $uid_f = c_1 \odot \ldots \odot (p_n : u_n)$; **return** $c_1 \odot \ldots \odot p_n \odot (0 : d)$
6: **else if** $uid_f /^+ uid_p$ **then** Let $uid_p = c_1 \odot \ldots \odot (p_n : u_n)$; **return** $c_1 \odot \ldots \odot p_n \odot (1 : d)$
7: **else if** $MiniSibling(uid_p, uid_f)$ **then return** $uid_p \odot (1 : d)$
8: **else** Let $uid_p = c_1 \odot \ldots \odot (p_n : u_n)$; **return** $c_1 \odot \ldots \odot p_n \odot (1 : d)$

Let us consider again the example of Figures 2 and 3. Assume one user inserts `Y` between `c` and `d` and then `Z` between `Y` and `d`, while another user concurrently inserts `W` between `c` and `d`. The final state is presented in Figure 7 (assuming $dW < dY$). If, subsequently, some user wants to insert `X` between `W` and `Y`, this creates a child under mini-node `W`, as illustrated in Figure 8, where $id(X) = [10(0 : dW)(1 : dX)]$, .

This algorithm is just a straightforward approach for generating fresh UIDs.

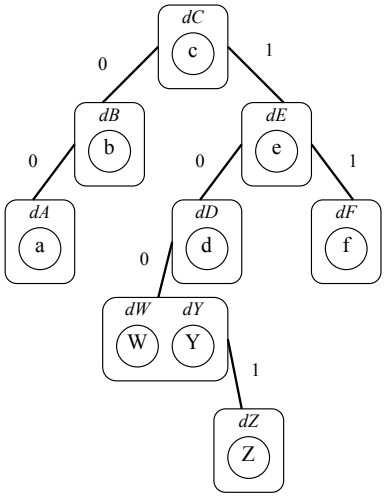

Figure 7: Identifiers after concurrent inserts

Note that, for any uid_p and uid_f, there is an infinite number of fresh UIDs, uid_m such that $uid_p < uid_m < uid_f$.

5.2 Site identifiers for disambiguators

Several alternatives are possible for disambiguators. Here we suggest using site identifiers, which are compact and easy to manage. For instance, MAC addresses are guaranteed unique, however they occupy 6 bytes. Shorter identifiers might be used instead for an object shared among a small number of sites.

5.3 Alternative supporting immediate discard

A possible alternative is a pair $(siteID, counter)$ for disambiguators, where *counter* is a per-site counter used to ensure global uniqueness. Such disambiguators are ordered as follows: $(s_1, c_1) < (s_2, c_2)$, iff: $c_1 < c_2 \vee (c_1 = c_2 \wedge s_1 < s_2)$.

With this approach, a mini-node that is deleted may be discarded (i.e., garbage collected) immediately, as proved in section 3.

When implementing treedoc as a tree, a mini-node can be discarded if it has no descendence. A major node with no mini-nodes can also be discarded. Conversely, the replay version of insert may find that ancestors of the new node have been discarded concurrently, and must re-create empty nodes to replace them.

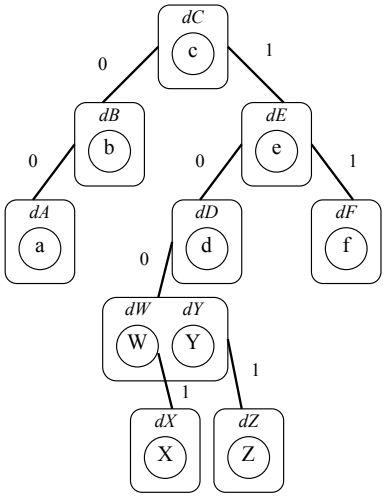

Figure 8: Identifiers after inserting atom between mini-siblings

6 Clean-up operations

The approach so far has some limitations. Paths are variable length and can become inefficient if the tree is unbalanced (e.g., if users always append to the end of the buffer). The tree metadata consumes memory and mini-nodes add complexity and overhead.

These issues can be mitigated somewhat. For instance, disambiguators could be removed once it is clear they are not necessary, i.e., that there is a single mini-node (nota that sibling mini-node only occur as a consequence of concurrent inserts). Deleted nodes can be garbage-collected. Imbalance can be avoided somewhat by using better heuristics when generating new unique identifier.

Rather than address these issues individually, this section proposes a more radical solution: structural clean-up operations that switch between the efficient, but unflexible, sequential buffer representation, and the more expensive edit-oriented tree representation. The specification of these operations is as follows.

- *explode*(*atomstring*). Returns a treedoc whose contents is identical to *atomstring*.
- *flatten*(*path*) Returns an atom string whose contents is identical to the sub-treedoc rooted at *path*.

The initiator and replay versions of these operations must have identical effect. In particular, *explode* must return exactly the same structure at all sites.

Observing that the capacity of a complete binary tree with *depth* levels is $2^{depth} - 1$, we suggest for *explode* the simple implementation of Algorithm 3. It is important to note that after executing an *explode*, the path of an atom is

a simple bitstring (as we can use ϵ for the disambiguators, with $\epsilon < d$, for all $d \neq \epsilon$ disambiguators).

Algorithm 3 *explode* and *flatten*

```
1: procedure explode (atomstring)  // atomstring: sequence of atoms
2:     depth = ⌈log2(length(atomstring) + 1)⌉
3:     T = Allocate a complete binary tree of depth depth
4:     Populate T in infix order with the atoms of atomstring
5:     Remove any remaining nodes
6:     Return T

7: procedure flatten (N)  // N: root of a subtree to be flattened
8:     Walk subtree in UID order
9:     Return a linear buffer containing the atoms of the non-empty nodes
```

With these clean-up operations, it becomes possible at any point in time to choose the most appropriate representation. For instance, when a tree becomes unbalanced, it suffices to *flatten* then *explode* it to fix the problem.

However, these internal clean-up operations do not genuinely commute with edit operations. We address this issue next.

6.1 Commuting clean-up with edits

A first observation is that the *explode* operation is not really necessary. Algorithm 3 can be interpreted as a mapping from a string to a canonical treedoc representation. Applying a path to a string implicitly converts the string to the canonical treedoc. Eliminating the explicit *explode* operation removes the need to make it commute with edits.

A second observation is that *flatten* is not an essential operation. When *flatten* is concurrent with an edit operation in the same subtree, then the edit should have higher precedence. More precisely, a conflicting edit causes a *flatten* to abort, leaving no side-effects (and causing no harm).

Therefore *flatten* executes a distributed commitment procedure. When executing *flatten* at some site, if this site observes the execution of a *insert*, *delete* or *flatten* within the sub-tree to be flattened, that site votes "No" to commitment, otherwise it votes "Yes." The operation succeeds only if all sites vote "Yes," otherwise it has no effect. Any distributed commitment protocol from the literature will do, for instance two-phase commit, three-phase commit, or Gray and Lamport's fault-tolerant protocol [9].

We may now envisage a mixed tree, where parts that are currently being edited are in treedoc representation, and parts that are currently quiescent are represented as strings.

6.2 Fault tolerance and disconnected operation

Ensuring fault tolerance and disconnected operation for disconnected edits is straightforward. Every site logs all its operations (whether locally initiated or remote) on persistent storage. When a site that was disconnected for some time reconnects with the rest of the system, it simply exchanges with other sites the missing information. If a site fails and recovers the situation is the same. If a site crashes, losing its memory, then when it restarts it behaves like a new site, and copies over the state of some other site; operations that it initiated before the crash and never sent to another site are lost.

The situation is more complex for *flatten*s, since they require a consensus. To ensure that consensus is solvable in the presence of crashes, we assume the existence of fault detectors [4].

To allow disconnected operation, fault detectors must be capable of distinguishing disconnection from a crash. During the commit phase of *flatten*, a disconnected site is assumed to be voting No, and *flatten* aborts. This is distinct from a crashed site, which is excluded from the commitment by the fault detector.

Note that if a disconnected site is falsely diagnosed as crashed, any operations that it initiated within a sub-tree that was *flatten*ed cannot be replayed, because they use now-forgotten node identities. Such operations are lost. Similarly, if this site initiated operations that depend on a node that was deleted and garbage-collected, then these operations are lost.

6.3 Generality of the approach

Clean-up operations and its execution relying on background consensus are used only for improving the efficiency of our solution, by reducing the size of metadata maintained by the treedoc. As such, an interesting question that arises is of its applicability to other concurrency control solutions previously proposed in literature.

When executing a *flatten* operation relying on a background consensus protocol, the system is guaranteeing that all replicas have received and executed the same set of operations (for a given subtree). Thus, for those operations, no concurrent operations may arise. As such, our approach could also be used in WOOT [20] for discarding the meta-data of characters inserted by such operations and for discarding the deleted characters, as this information is only necessary for handling concurrent updates.

7 Related work

A comparison of several approaches to the problem of collaboratively editing a shared text was written by Ignat et al. [10].

Operational transformation (OT) [6, 29, 28, 15, 14] considers collaborative editing based on non-commutative single-character operations. To this end, OT transforms the arguments of remote operations to take into account the

effects of concurrent executions. OT requires two correctness conditions [24]: the transformation should enable concurrent operations to execute in either order, and furthermore, transformation functions themselves must commute. The former is relatively easy. The latter is more complex, and Oster et al. [19] prove that all previously proposed transformations violate it. After this, several OT solutions have been proposed [18, 31, 16]. However, the new solutions are also complex and it remains hard to verify their correctness.

OT attempts to make non-commuting operations commute after the fact. We believe that a better approach is to design operations to commute in the first place. This is more elegant, and avoids the complexities of OT.

A number of papers study the advantages of commutativity for concurrency and consistency control [2, 32, for instance]. Systems such as Psync [17], Generalized Paxos [13], Generic Broadcast [22] and IceCube [23] make use of commutativity information to relax consistency or scheduling requirements. However, these works do not address the issue of achieving commutativity.

Weihl [32] distinguishes between forward and backward commutativity. They differ only when operations fail their pre-condition. In this work, we consider only operations that succeed at the submission site, and ensure by design that they won't fail at replay sites.

Roh et al. [25] were the first to suggest the CRDT approach. They give the example of an array with a slot assignment operation. To make concurrent assignments commute, they propose a deterministic procedure (based on vector clocks) whereby one takes precedence over the other.

This is similar to the well-known Last-Writer Wins algorithm, used in shared file systems. Each file replica is timestamped with the time it was last written. Timestamps are consistent with happens-before [11]. When comparing two versions of the file, the one with the highest timestamp takes precedence. This is correct with respect to successive writes related by happens-before, and constitutes a simple precedence rule for concurrent writes.

In the precedence design of Roh et al., concurrent writes to the same location are lost. This is inherent to the destructive assignment operation that they consider. In ours, concurrent inserts commute genuinely, which is important in order to support co-operative work.

Oster et al. recently proposed a replicated character buffer CRDT called WOOT that supports only insert and delete operations [20]. Each character has a unique identifier, and maintains the identifiers of the previous and following characters at the initial execution time; this is a lot of overhead, as discussed earlier in this paper. The WOOT data structure grows indefinitely, because there is no garbage collection or restructuring. WOOT does not support block operations.

In Lamport's replicated state machine approach [11], every replica executes the same operations in the same order. This total order is computed either by a consensus algorithm such as Paxos [12] or, equivalently, by using an atomic broadcast mechanism [5]. Such algorithms can tolerate faults, however they are complex and scale poorly. As consensus occurs within the critical execution path, it adds latency to every operation.

The precedence approach can be viewed as a poor-man's total order. It does not require an online consensus algorithm, but it loses work.

In the treedoc design, common edit operations execute optimistically, with no latency; it uses consensus in the background only. Previously, Golding relied on background consensus for garbage collection [7]. We are not aware of previous instances of background consensus for clean-up operations, nor of aborting consensus when it conflicts with essential operations.

8 Performance comparison

We compare the overheads of treedoc with a flat buffer and with WOOT [20]. The common case is that an atom represents a single character; in the following we assume $\mathsf{sizeof}(atom) = 8$ bits. Our results are summarised in Table 1.

8.1 Time complexity

The time complexity for executing an insert or remove operation in a linear buffer representation is constant $O(1)$. WOOT requires $O(n)$ for a linear search of the unique identifers. In treedoc, it is $O(\log n)$ for a balanced tree. In both cases, auxiliary indexing structures (e.g., hashing) could be used to improve performance at the cost of greater space.

8.2 Memory

The linear buffer structure is an array, which can remain approximately of size m, where m is the current number of non-deleted atoms. (generally $m < n$). In bits, the size is $m \times \mathsf{sizeof}(atom) = 8m$.

The WOOT data structure is a list of nodes, where each node stores an atom, the atom's identifier, and its predecessor and successor at the initiator. WOOT never discards deleted atoms. Atom identifiers (noted *WOOTID* hereafter) must be globally unique within the document for its whole lifetime; assume they are composed of a site identifier and a counter. Thus, we evaluate the storage size to $N \times (\mathsf{sizeof}(atom) + 3 \times \mathsf{sizeof}(WOOTID) + 1)$ (counting one bit to mark an atom as deleted), where N is the aggregated number of atoms ever inserted in the document. Assuming a pointer is 4 bytes, and a WOOT identifier is 10 bytes (6 bytes of MAC address and an integer counter), this comes to $N \times (8 + 3 \times 80 + 1) = 249N$ bits.

Using standard tree algorithms, the storage size of treedoc is $n \times (\mathsf{sizeof}(atom) + 2 \times \mathsf{sizeof}(pointer) + \mathsf{sizeof}(disambiguator) + 2)$, counting a deleted bit and a bit to distinguish mini-nodes. Here, n is the number of atoms currently in the structure; it is likely that $n \ll N$. In our first alternative, a disambiguator is a short integer. The second one (omitted from Table 1 requires a unique site identifier and a counter; however n is smaller in this case. In the former case, assuming 2-bytes disambiguators, the storage size comes to $n \times (8 + 2 \times 32 + 16 + 2) = 90n$ bits. In

the latter, assuming the same size as a *WOOTID*, the storage size comes to $n \times (8 + 2 \times 32 + 80 + 2) = 154n$ bits. (A tree can also be stored more compactly, e.g. as a parenthesised list, avoiding the cost of pointers, at the expense of performance.)

The *flatten* operation, presented in Section 6, results in zero storage overhead. The flattened storage size is exactly the same as the flat buffer.

In summary, the memory consumption of non-flattened treedoc is less than half or a third of WOOT, and flattened treedoc consumes even less memory. It is important to notice that this comparison does not consider the impact of discarding deleted atoms. Considering this possibility of our approach, the comparison would be even more favorable to our approach.

8.3 Message size

To transmit the insert operation, WOOT sends the new atom, and the identifiers of the new atom, of its predecessor and of its successor; the transmission size is $\mathsf{sizeof}(atom) + 3 \times \mathsf{sizeof}(WOOTID) = 248$ bits.

In contrast, treedoc transmits the atom and its path (UID). Assuming a balanced tree, and the common case of a path containing a disambiguator only at the end, the size of a path is at most $\log_2 n + \mathsf{sizeof}(disambiguator)$, where n is the current size of the document. Thus the transmission size is less or equal $\mathsf{sizeof}(atom) + \log n + \mathsf{sizeof}(disambiguator)$. For a one-megabyte text and 2-byte disambiguators, this comes to $8 + 20 + 16 = 44$ bits. For 10-byte disambiguators, this comes to $8 + 20 + 80 = 108$ bits.

To transmit the remove operation, WOOT sends the identifiers of the atom to be deleted; the transmission size is $\mathsf{sizeof}(WOOTID) = 80$ bits.

Treedoc also transmits the atom identifiers (UID). Assuming a balanced tree, and the common case of a path containing a disambiguator only at the end, the size of a path is at most $\log_2 n + \mathsf{sizeof}(disambiguator)$, where n is the current size of the document. For a one-megabyte text and 2-byte disambiguators, this comes to $20 + 16 = 36$ bits. For 10-byte disambiguators, this comes to $20 + 80 = 100$ bits.

In summary, for a large document, the message size of treedoc is between 5.6 and 2.3 times less than WOOT for inserts, and between 2.2 times less and 1.25 times more for deletes.

9 Conclusion

It was known previously that commutativity simplifies consistency maintenance, but the issue of designing systems for commutativity was neglected. This paper suggested a new paradigm for replication: the Commutative Replicated Data Type or CRDT, designed such that concurrent operations commute. Replicas of any CRDT converge if operations are executed respecting causality. This makes the implementation of replicated systems much simpler than alternative techniques.

	flat	WOOT	treedoc	
			tree	flat
Time	$O(1)$	$O(m)$	$O(\log m)$	
Space	m	$31.125N$	$11.25n$	m
(Ratio)	1	311	22	1
Message (add)	5	31	≤ 5.5	N/A

Table 1: Comparison. Sizes in bytes. m = current number of characters; n = current number of characters, including deleted characters; N = cumulated number of characters; $m \leq n \ll N$. Ratio computed for $m = 10^6$, $n = 2 \times m$, $N = 10 \times m$.

However, designing a non-trivial genuine CRDT (i.e., that preserves user intents) is not trivial. We give a genuine CRDT solution to the problem of a shared edit buffer, by implementing some known techniques in a novel and efficient way (invariant identifiers represented as paths in a binary tree) and by some new techniques (abortable consensus in the background). Our solution also guarantees intention preservation.

We also propose a novel technique, background consensus, to clean up redundant metadata without interfering with ordinary edit operations. This mechanism is generic and it could be used to improve other editor algorithms.

References

[1] Authors and title omitted for anonymity. Technical report.

[2] B. R. Badrinath and K. Ramamritham. Semantics-based concurrency control: beyond commutativity. *ACM Trans. Database Syst.*, 17(1):163–199, Mar. 1992.

[3] P. Cederqvist et al. *Version Management with CVS*. Network Theory Ltd., 2006.

[4] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, 1996.

[5] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, Dec. 2004.

[6] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *Int. Conf. on Management of Data (SIGMOD)*, pages 399–407, Portland, OR, USA, 1989. ACM SIGMOD, ACM. Invention of Operational Transformation.

[7] R. A. Golding. *Weak-consistency group communication and membership*. PhD thesis, University of California Santa Cruz, Santa Cruz, CA, USA, Dec. 1992. Tech. Report no. UCSC-CRL-92-52.

[8] Google. Google docs. http://documents.google.com/.

[9] J. Gray and L. Lamport. Consensus on transaction commit. *ACM Trans. Database Syst.*, 31(1):133–160, Mar. 2006.

[10] C.-L. Ignat, G. Oster, P. Molli, M. Cart, J. Ferrié, A.-M. Kermarrec, P. Sutra, M. Shapiro, L. Benmouffok, J.-M. Busca, and R. Guerraoui. A comparison of optimistic approaches to collaborative editing of Wiki pages. In *Int. Conf. on Coll. Computing: Networking, Apps. and Worksharing (CollaborateCom)*, number 3, White Plains, NY, USA, Nov. 2007.

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

[12] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[13] L. Lamport. Generalized consensus and Paxos. Technical Report MSR-TR-2005-33, Microsoft Research, Mar. 2005.

[14] D. Li and R. Li. Ensuring content and intention consistency in real-time group editors. In *Int. Conf. on Distributed Comp. Sys. (ICDCS)*, pages 748–755, Hachioji, Tokyo, Japan, Mar. 2004. IEEE Computer Society.

[15] D. Li and R. Li. Preserving operation effects relation in group editors. In *Int. Conf. on Computer-Supported Cooperative Work (CSCW)*, pages 457–466, New York, NY, USA, 2004. ACM.

[16] R. Li and D. Li. A new operational transformation framework for real-time group editors. *IEEE Trans. Parallel Distrib. Syst.*, 18(3):307–319, 2007.

[17] S. Mishra, L. Peterson, and R. Schlichting. Implementing fault-tolerant replicated objects using Psync. In *Symp. on Reliable Dist. Sys.*, pages 42–52, Seattle, WA, USA, Oct. 1989. IEEE.

[18] G. Oster, P. Molli, P. Urso, and A. Imine. Tombstone transformation functions for ensuring consistency in collaborative editing systems. In *Int. Conf. on Coll. Computing: Networking, Apps. and Worksharing (CollaborateCom)*, page 10, Atlanta, Georgia, USA, Nov. 2006. IEEE Computer Society.

[19] G. Oster, P. Urso, P. Molli, and A. Imine. Proving correctness of transformation functions in collaborative editing systems. Rapport de recherche RR-5795, LORIA – INRIA Lorraine, Dec. 2005.

[20] G. Oster, P. Urso, P. Molli, and A. Imine. Data consistency for P2P collaborative editing. In *Int. Conf. on Computer-Supported Cooperative Work (CSCW)*, pages 259–268, Banff, Alberta, Canada, Nov. 2006. ACM Press.

[21] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Trans. Softw. Eng.*, 9(3):240–247, 1983.

[22] F. Pedone and A. Schiper. Handling message semantics with generic broadcast protocols. *Distributed Computing Journal*, 15(2):97–107, 2002.

[23] N. Preguiça, M. Shapiro, and C. Matheson. Semantics-based reconciliation for collaborative and mobile environments. In *Int. Conf. on Coop. Info. Sys. (CoopIS)*, volume 2888 of *Lecture Notes in Comp. Sc.*, pages 38–55, Catania, Sicily, Italy, Nov. 2003. Springer-Verlag.

[24] M. Ressel, D. Nitsche-Ruhland, and R. Gunzenhäuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *Int. Conf. on Computer-Supported Cooperative Work (CSCW)*, pages 288–297, Boston, MA, USA, May 1996. ACM. Correctness conditions TP1 and TP2 for OT.

[25] H.-G. Roh, J.-S. Kim, and J. Lee. How to design optimistic operations for peer-to-peer replication. In *Int. Conf. on Computer Sc. and Informatics (JCIS/CSI)*, Kaohsiung, Taiwan, Oct. 2006.

[26] Y. Saito and M. Shapiro. Optimistic replication. *Computing Surveys*, 37(1):42–81, Mar. 2005.

[27] M. Shapiro, K. Bhargavan, and N. Krishna. A constraint-based formalism for consistency in replicated systems. In *Proc. 8th Int. Conf. on Principles of Dist.*

Sys. (OPODIS), number 3544 in Lecture Notes in Comp. Sc., pages 331–345, Grenoble, France, Dec. 2004.

[28] C. Sun and C. Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Int. Conf. on Computer-Supported Cooperative Work (CSCW)*, page 59, Seattle WA, USA, Nov. 1998.

[29] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *Trans. on Comp.-Human Interaction*, 5(1):63–108, Mar. 1998.

[30] C. Sun, S. Xia, D. Sun, D. Chen, H. Shen, and W. Cai. Transparent adaptation of single-user applications for multi-user real-time collaboration. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 13(4):531–582, dec 2006. CoWord and CoPowerpoint.

[31] D. Sun and C. Sun. Operation context and context-based operational transformation. In *CSCW '06: Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, pages 279–288, New York, NY, USA, 2006. ACM.

[32] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Trans. Comput.*, 37(12):1488–1505, Dec. 1988.

[33] S. Xia, D. Sun, C. Sun, D. Chen, and H. Shen. Leveraging single-user applications for multi-user collaboration: the coword approach. In *Int. Conf. on Computer-Supported Cooperative Work (CSCW)*, pages 162–171, New York, NY, USA, 2004. ACM.