Asynchronous rebalancing of a replicated tree

Marek Zawirski, Marc Shapiro, Nuno Preguiça

RenPar'20 / SympA'14 / **CFSE 8**
Saint-Malo, France, du 10 au 13 mai 2011

Asynchronous rebalancing of a replicated tree

Marek Zawirski*, INRIA & UPMC, Paris, France
Marc Shapiro, INRIA & LIP6, Paris, France
Nuno Preguiça, CITI, Universidade Nova de Lisboa, Portugal

Abstract
W e study the problem of rebalancing a replicated tree, while the tree is concurrently being updated in a P2P manner. Tree updates are asynchronous and commutative, as we aim at eventual consistency. However, rebalancing requires strong synchronisation, because only replicas that have performed the same rebalances can communicate with one another. In order to scale to large networks, we propose to break this synchronisation into two parts: commitment within a small *core* of replicas, followed by asynchronous, pairwise *catch-up* protocol between replicas at different rebalance numbers. We state the requirements and correctness conditions for this distributed algorithm, and propose a correct solution.
Keywords: replicated tree, collaborative editing, garbage collection, distributed systems

1. Introduction

In a large-scale distributed system, objects are widely replicated for availability and performance. In the popular *Eventual Consistency* approach [6, 10], a replica accepts updates at any time (ensuring performance and availability whatever the network conditions), and propagate these updates asynchronously.
A recent insight is that eventual consistency can be achieved easily if concurrent operations commute [3, 5, 12, 13, 15], since this means that replicas converge, irrespective of the order in which operations are received at each replica. We have designed a number of data types with this property, which we call *Commutative Replicated Data Types* (CRDTs) [4, 9]. Generally speaking, commutativity is achieved at the cost of potentially unbounded memory usage or message size; to keep this growth in check, garbage collection (GC) is needed, which requires synchronisation. The high-level issue studied in this paper is how to combine such (infrequent) synchronous operations into the general asynchronous system.
Our most involved design is Treedoc [4], a sequence abstraction for cooperative editing applications, where distributed users collaborate by editing a shared document. Treedoc assigns a position identifier to each atomic text element (called an atom). The order of identifiers determines the order of atoms. To ensure that a new atom can be added between any two existing ones, identifiers are organised in a grow-only tree. Over time the tree becomes unbalanced and removed atoms ("tombstones") waste space in the tree; therefore, the tree is occasionally *rebalanced* and tombstones discarded. Since rebalancing changes atom identifiers, replicas must commit to rebalance together, otherwise they cannot communicate.
In order to move commitment off the critical path, Leția et al. [2] propose a two-tier architecture. A small and stable subset of replicas (the core) commits a sequence of rebalance operations; each one moves the core into a new *epoch*. The other replicas (the nebula) asynchronously suffer rebalance operations originating in the core. Rebalance in the core is inherently concurrent with tree updates in the nebula. Replicas in different epochs cannot communicate with one another, since they refer to different base trees. The challenge, then, is to design a *catch-up protocol* to allow a nebula replica to rebalance, while moving to a later epoch and adjusting its updates to refer to the newly rebalanced tree.
This paper proposes the first safe catch-up algorithm for a replicated tree undergoing concurrent updates. As trees are a ubiquitous data structure, and although we focus on a particular kind of tree, we believe this is of general interest. Our problem is related to some basic distributed systems issues, such as resetting vector clocks or garbage collection. The specific contributions of this paper are as follows:

1. In Section 2, after introducing CRDTs, we provide a formal specification of the Treedoc algorithm.

* This research was supported in part by ANR project ConcoRDanT (ANR-10-BLAN 0208). Marek Zawirski is a recipient of the Google Europe Fellowship in Distributed Computing, and this research is supported in part by this Google Fellowship.

2. In Section 3, we formulate precisely the requirements and correctness conditions for a catch-up protocol, and identify some major errors in the algorithm of Leția et al. [2].
3. In Section 4, we propose algorithm *F-translate* that meets the requirements and safety conditions.

Finally, we compare to related work in Section 5 and conclude by Section 6.

2. Background

In this section, we formalise Treedoc data type [4] as a CRDT [9] and introduce the rebalance problem.

2.1. CRDT system model

We consider a distributed system of processes interconnected by an asynchronous network, without discussing failures. There is a replicated shared object x; its replica at process i is denoted x_i. External *clients* may invoke *operations* in the object's interface, at some selected replica of their choice.

A read-only *query* operation executes entirely at one replica. A mutating *update* is divided into two phases. The first synchronous phase, executes at the *source* replica where the update is invoked. It is read-only; it may return values to the caller, and/or compute arguments for the next phase. This *downstream phase* asynchronously updates the state of every replica x_i.

Invoked operations are propagated by reliable causal broadcast, so the downstream phase of every operation eventually takes effect at every replica. By design, an every pair of concurrent operations *commute* in this model, thus replicas eventually converge [9].

2.2. Treedoc

Treedoc [2, 4] is a CRDT designed for concurrent editing. It considers shared document as a sequence of text elements called *atoms*; an atom can be *added* or *removed*. Atoms must be identified non-ambiguously despite updates. Treedoc uses a variation of a tree to define the order between atoms and identifies each atom using its unique *position identifier* in the tree.

A position is associated with a *node* in the tree; its identifier is simply the path from the root to the node. Nodes are ordered left to right, according to infix tree traversal. Considering some node whose identifier is n, any left (resp. right) child has an identifier of the form $n \bullet 0d$ (resp. $n \bullet 1d$) and is ordered before (resp. after) n.

Collectively, the set of left (resp. right) children of n are considered to constitute the *left major node* (resp. right). Elements of a major node are called *mini-nodes*, and their distinguishing part d is called a *disambiguator*.[2] Mini-nodes are ordered within their major node by their disambiguator d.

Figure 1a illustrates this concept. It shows an example state from two different perspectives. The editing application is only interested in the sequential order at the bottom. At the top, we illustrate the tree. As an example, atom `"a"` has position identifier $d_o \bullet 0d_s \bullet 1d_a$.

Let us define formally basic relations between positions, corresponding to relations between tree nodes:

Definition 1. *Given two different Treedoc identifiers* a *and* b, $a < b$ *iff (hereafter* $B \in \{0, 1\}$*):*

- a *is a prefix of* $b = a \bullet j_1 \bullet \ldots \bullet j_m$ *and* $j_1 = 1d$*, whatever the disambiguator* d*; or*
- b *is a prefix of* $a = b \bullet i_1 \bullet \ldots \bullet i_m$ *and* $i_1 = 0d$*, whatever* d*; or*
- a *and* b *have a common prefix:* $a = c_1 \bullet \ldots \bullet c_n \bullet i_1 \bullet \ldots \bullet i_n$ *and* $b = c_1 \bullet \ldots \bullet c_n \bullet j_1 \bullet \ldots \bullet j_m$*, where* $i_1 = B_i d_i$ *and* $j_1 = B_j d_j$*, and* $B_i < B_j$ *or* $B_i = B_j \wedge d_i < d_j$.

Definition 2. a *is a* parent *of* b*, noted* a/b*, iff* a *is a prefix of* b*, such that* $b = a \bullet Bd$*, whatever* B *and* d.

Definition 3. a *is a* mini-sibling *of* b*, noted MiniSibling*(a, b)*, iff* $a = c_1 \bullet \ldots \bullet c_n \bullet B_a d_a$ *has a common prefix with* $b = c_1 \bullet \ldots \bullet c_n \bullet B_b d_b$*, and* $B_a = B_b$.

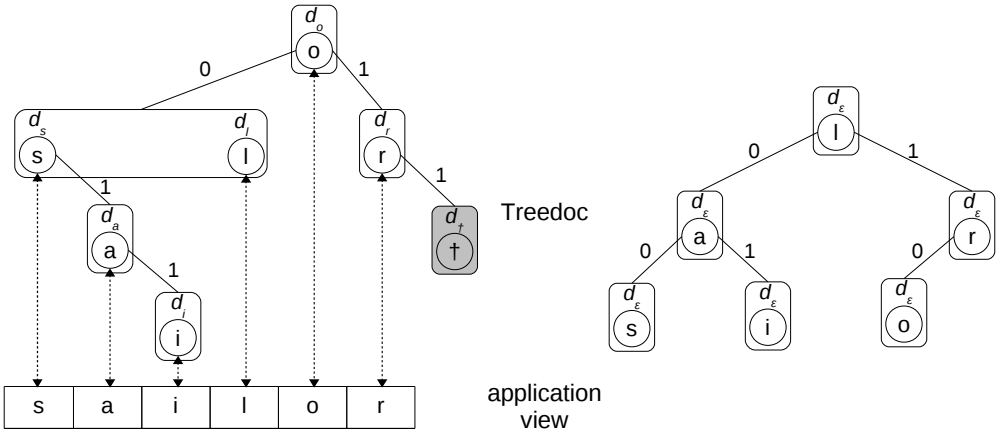
Specification 1 defines Treedoc using simple CRDTs specification language [9] and above definitions. There are two commutative update operations defined for the sequence:

- *addAt* creates a fresh position identifier between two existing positions by adding a new tree node.
- *removeAt* removes the atom from the position. It does not discard node from the payload. Instead, it marks it as empty, by replacing atom with *tombstone marker* † on every replica.

Treedoc identifier system maintains a few important properties:

i) Replicas of the same atom have the same identifier and different atoms have different identifiers.

[2] As it disambiguates among left (resp. right) children concurrently added under n.



(a) Unbalanced tree (b) Balanced tree

Figure 1: Treedoc representing the string `sailor`. A round-curved rectangle represents a major node; a circle represents a mini-node.

Specification 1 Treedoc operations

```
1:  payload set of of (PosID, atom) pairs                ▷ Hereafter x in specification refers to local replica.
2:      initial {(begin, †), (end, †)} where begin < end
3:  query allocateIDBetween (PosID a, PosID b) : PosID p
4:      pre a < b ∧ ∄(q, _) ∈ x : a < q < b
5:      let d = myID()                                   ▷ Identifier of source, assumed unique
6:      if a is an ancestor of b then
7:          let p = b • 0d
8:      else
9:          let p = a • 1d
10: update addAt (PosID a, atom c, PosID b)
11:     atSource (p)
12:         let p = allocateIDBetween(a, b)
13:     downstream (p, c)
14:         x := x ∪ {(p, c)}
15: update removeAt (PosID p)
16:     atSource ()
17:         pre (p, _) ∈ x
18:     downstream (p)
19:         pre (p, _) ∈ x
20:         x := x \ {(p, _)} ∪ {(p, †)}
```

ii) Adding or removing an atom does not change the identifier of any extant atom. *addAt* adds a new leaf to the tree, while *removeAt* only marks node as empty, but do not discard it.

iii) Identifier space is dense: it is always possible to add a leaf ordered between arbitrary adjacent nodes. Concurrent updates at the same location (i.e., within the same major node) are distinguished by their disambiguator, which we ensure are unique by using unique site identifiers.

2.2.1. The tree maintenance problem

In large-scale experiments with Treedoc against CVS or Wikipedia traces, we found that Treedoc would suffer from unbalance and an accumulation of empty nodes [4]. This wastes space, slows performance, and causes identifiers to grow bigger than the expected logarithmic size. To solve these issues, it was

proposed to rebalance the tree periodically, discarding useless empty nodes.
Rebalancing preserves the sequential order of atoms, but changes their position identifiers. The *rebalance* procedure operates by creating a new, balanced tree of appropriate size, and copying the atoms from the old tree to the new one, in order. A left branch is labeled $0d_\epsilon$, a right one $1d_\epsilon$, where d_ϵ is a constant disambiguator unused by *allocateIDBetween*. Figure 1b depicts the tree from Figure 1a after *rebalance*. We omit the formal specification of *rebalance*, as it is trivial.
Although the order of atoms remains unchanged after rebalance, identifiers are reassigned; *rebalance* does not commute with tree updates, and thus breaks the basic CRDT property. Therefore, only replicas that have performed the same sequence of rebalances on the same states can communicate with one another. A group of replicas may agree to rebalance together by running a commitment protocol. The drawback is that updates are blocked during the protocol, and commitment does not scale to large, dynamic, weakly connected systems [11].

3. The core-nebula solution

Leția et al. [2] introduce a two-tier architecture, enabling rebalance even in a large-scale system with churn. Replicas are divided into disjoint sets, the core and the nebula. The *core* (abbreviated C hereafter) is a group with known membership of well-connected replicas. Core replicas may be the source for tree update operations, and participate in the rebalance commitment. The core is assumed sufficiently small, stable and well-connected that the rebalance commitment usually succeeds and executes quickly.
Nebula replicas (abbreviated B for neBula) may join or leave the system, but are allowed to be source for tree updates only.
We define an *epoch* to demark each rebalance. Epochs are numbered sequentially; each epoch denotes a different identifier frame of reference. Replicas can communicate with one another within the same epoch only. Note that, by construction, replicas within the core are in the same epoch. Nebula replicas may update the tree, while it is concurrently rebalanced.
A nebula replica may be one or more epochs behind the core. The *catch-up protocol* moves a nebula site to the next epoch. It is a pairwise protocol between nebula replica x_i in epoch e and another (core or nebula) replica x_j in epoch $e+1$ or later.
Its role is to replay rebalance on replica x_i, bringing it into epoch $e+1$, and to transform the updates executed at x_i in epoch e to refer the rebalanced tree, so that they now apply to epoch $e+1$. We illustrate the evolution of x_i in Figure 2. On the left, the state before catch-up. The nodes forming the string `"lit"`, coloured blue (medium grey in greyscale), were rebalanced in the core; the yellow (light grey) node (`"f"`) has been inserted concurrently. On the right, the state after catch-up. The blue nodes now form a balanced tree, whereas the yellow nodes has remained attached to its parent, which has moved. Both before and after catch-up, the document reads `"lift"`.

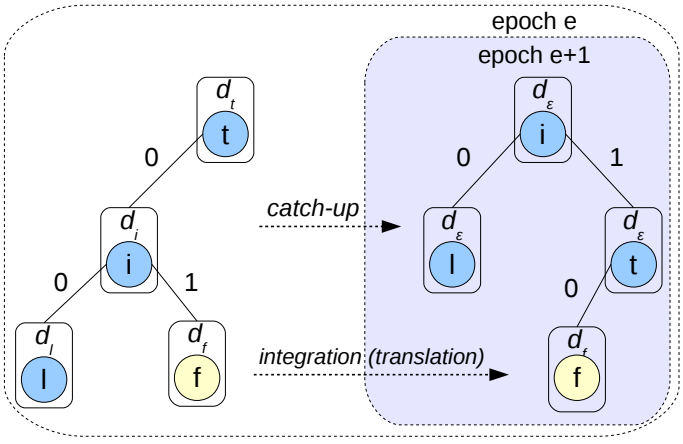


Figure 2: A nebula replica before and after catch-up.

3.1. Rebalance in the core

We note the epoch as a superscript; for instance, position p in epoch e is noted p^e. We denote $ops(x_i^e)$ the set of updates that took effect on replica x_i in epoch e, leading to state x_i^e. Within a single epoch, $ops(x_i^e)$ is a grow-only set, similar to a causal history [7].
Replicas within the core execute a commit protocol in order to rebalance, to agree on the *final* state and equivalent set of updates of the epoch e in the core before rebalance, noted $x^{e.F}$ and $ops(x^{e.F})$ respectively.[3] The *rebalance* algorithm from Section 2.2.1 processes this common state at every core replica;

[3] In practice, one may use the trick to use only quorum to rebalance, combined with catch-up at the core

therefore all core replicas start epoch $e+1$ with the same initial state, noted $x^{e+1.I}$. Conceptually, this initial state corresponds to the set of updates *addAt* creating a balanced tree, $ops(x^{e+1.I}) \triangleq \{addAt(p^{e+1}, c) : (p^{e+1}, c) \in x^{e+1.I}\}$. During a subsequent rebalance, it holds that $ops(x^{e+1.I}) \subseteq ops(x^{e+1.F})$, and so on for each epoch.

3.2. Catch-up of nebula replica

A nebula replica x_B can freely exchange updates with other nebula or core replicas operating in the same epoch. However, when it discovers a replica operating in a later epoch, it performs a pairwise catch-up protocol with the more up-to-date replica. Each nebula replica can catch up independently. The protocol described hereafter assumes a nebula replica x_B^e (*neBula*) catching up with a replica x_C^{e+1} (*Core*). If the distance between epochs is higher than one, the same protocol is executed iteratively.

The catch-up algorithm is performed locally on x_B^e, after contacting x_C^{e+1}. We present an abstract algorithm that works on *ops* set for brevity — not the actual implementation. It takes as an input $ops(x^{e.F})$, which is the information that must be available at and acquired from x_C^{e+1}. The implementation needs to ensure that the nebula replica received all these updates through its standard delivery mechanism, and that it is able to tell which updates are contained within this set. Reliable causal broadcast with vector clocks is sufficient to implement it, cheaper alternatives are possible. Specification 2 presents the algorithm that performs following steps:

1. Create an empty replica y^e as a sandbox to replay core behavior. Apply all updates from $ops(x^{e.F})$ both on y^e and x_B^e (if they were not applied before).
2. Replay *rebalance* at y^e, and initialize x^{e+1} from this sandbox.
3. Translate updates from $ops(x^e)$ that were not included in rebalance $ops(x^{e.F})$ (concurrent with *rebalance*) into the new epoch, then apply and propagate them. The algorithm may cause many replicas translating and propagating the same update, but it is safe since updates are idempotent.

Specification 2 Catch-up at nebula replica

```
1:  update catchUp (set of updates ops(x^{e.F}))
2:      y^e := ∅                                  ▷ Create a local replica for core rebalance emulation
3:      for u ∈ ops(x^{e.F}) do                   ▷ Iterate in order respecting causality
4:          y^e.apply(u)
5:          if u ∉ ops(x^e) then
6:              x^e.apply(u)
7:      y^e.rebalance
8:      x^{e+1} := y^{e+1}                        ▷ Initialize new tree with balanced part and switch to epoch e + 1
9:      for u ∈ ops(x^e) \ ops(x^{e.F}) do        ▷ Iterate in order respecting causality
10:         u.p^{e+1} := translate(x^{e.F}, u.p^e)
11:         x^{e+1}.apply(u)                      ▷ Apply locally and disseminate in e + 1
12:         x^{e+1}.bcast(u)
```

This algorithm safely transforms replica into the new epoch, assuming the existence of the *translate* function. This is the most complex part of catch-up; we will now show what requirements such a function must meet.

3.3. Requirements for translation algorithm

Catch-up is complex because the set of updates before translation $ops(x_B^e)$ can be completely unrelated at each replica.[4]. Indeed, any nebula node can catch-up independently and there is no restriction on how replicas exchange updates (other than the causality requirement).

Therefore, for instance, a nebula replica might not observed the orderthat need to be ordered followed during translation. Imagine a configuration with three nebula replicas x_a, x_b and x_c. Suppose that

[4] We are also investigating simpler translation algorithm that is trying to enforce such relations.

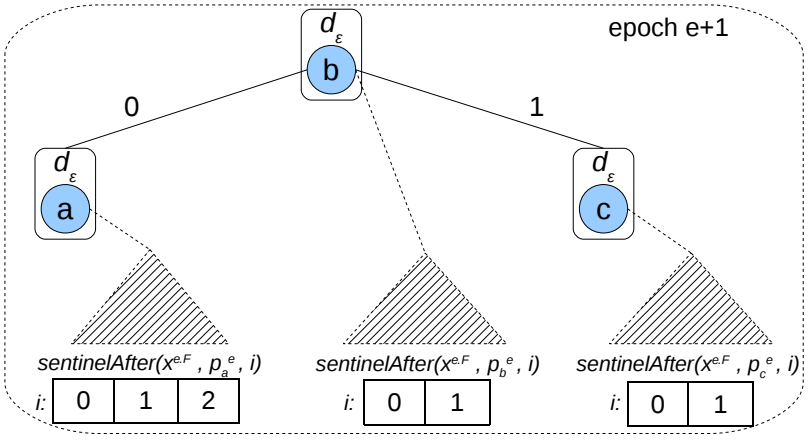


Figure 3: Sentinel positions abstraction on an exemplary rebalanced tree in epoch $e+1$; $i_{max} \in \{2,3\}$. Surface view of the tree corresponds to the total order, e.g. $p_b^{e+1} < \mathit{sentinelAfter}(x^{e.F}, p_b^e, 1) < p_c^{e+1}$.

replica x_a added (p_a^e, a), and replica x_b added (p_b^e, b). Replica x_c might receive and apply both operations and observe the order $p_a^e < p_b^e$, whereas x_a and x_b do not receive each other's updates. When x_a and x_b catch-up, they must enforce the order $p_a^{e+1} < p_b^{e+1}$ observed at x_c, even though they did not observe it themselves. Furthermore, translation should lead to the same output on every replica.
The crucial observation to build a correct translation is that there is a common knowledge between catching up replicas. The algorithm can utilize the set of final operations in previous epoch, $\mathit{ops}(x^{e.F})$, and the corresponding state $x^{e.F}$. Based on this observation, we formulate the safety requirements for the translation algorithm $\mathit{translate}(x^{e.F}, p^e) : p^{e+1}$:

1. Determinism: given the same arguments it results in the same output at every replica; in other words, a new position p^{e+1} depends only on the state before rebalance $x^{e.F}$ and the old position p^e:
$$\forall x_i^e, x_j^e, x^{e.F}, p^e : x_i^e.\mathit{translate}(x^{e.F}, p^e) = x_j^e.\mathit{translate}(x^{e.F}, p^e) \quad (1)$$
2. Non-disruptiveness: translation of a position from epoch e cannot conflict with an identifier created by *addAt* in epoch $e+1$:
$$\forall x_i^e, x_j^{e+1}, x^{e.F}, p^e, p_a^{e+1}, p_b^{e+1} : x_i^e.\mathit{translate}(x^{e.F}, p^e) \neq x_j^{e+1}.\mathit{allocateIDBetween}(p_a^{e+1}, p_b^{e+1}) \quad (2)$$
3. Order preservation: translation must maintain the order of positions between epochs:
$$\forall x_i^e, x_j^e, x^{e.F}, p^e, p'^e : p^e < p'^e \implies x_i^e.\mathit{translate}(x^{e.F}, p^e) < x_j^e.\mathit{translate}(x^{e.F}, p'^e) \quad (3)$$

The translation algorithm proposed before [2] missed first two observations, and used an algorithm similar to *allocateIDBetween* in order to translate, as in Figure 2. Such an algorithm relies on local state and not only $x^{e.F}$. Unfortunately, this intuitive idea is flawed (allowing, for instance, broken order) when nebula replicas communicate in P2P manner.

4. New F-translate algorithm

Based on the above requirements we propose a novel *F-translate* algorithm that relies on the *final* state before rebalance. The algorithm is presented in a simplified way for sake of clarity; we skip implementation details and symmetric cases common in tree handling; we decided to use verbose names for notations and focus on figures to omit straight-forward definitions.
The basic abstraction used by F-translate algortihm is the *sentinel position*. Since translation needs to be deterministic and can depend only on the set of final nodes (existing in the final state), it must have a predefined result for each potential input position. The algorithm analyzes state $x^{e.F}$ to investigate all potential subtrees and target each of these cases to a designated *sentinel position* below the rebalanced

tree $x^{e+1.I}$ in epoch $e+1$. Such predestination process requires usage of special "empty" nodes (technically equivalent to tombstone) below rebalanced tree to order positions properly. However, it is enough to materialize empty nodes only when the corresponding sentinel position is in use.

Definition 4. *For two adjacent positions in some final state $x^{e.F}$, p_a^e and p_b^{e+1}, where $p_a^e < p_b^e$, the sentinel position sentinelAfter$(x^{e.F}, p_a^e, i)$ in epoch $e+1$ is a position such that:*

$$\forall i \in \{0..i_{max}\}: \quad \ldots < x_a^{e+1} < \mathit{sentinelAfter}(x^{e.F}, p_a^e, i) < x_b^{e+1} < \ldots \tag{4}$$

$$\forall i, j \in \{0..i_{max}\}, i < j: \quad \mathit{sentinelAfter}(x^{e.F}, p_a^e, i) < \mathit{sentinelAfter}(x^{e.F}, p_a^e, j) \tag{5}$$

where i_{max} is a precomputed number of sentinels reserved for p^e. A sentinel position must use special disambiguator d_ϵ below $x^{e+1.I}$ in order to not conflict with positions created by addAt.

Figure 3 shows some example sentinel positions on a simple tree. We omit implementation details of this abstraction now, and focus on its application in translation algorithm. Importantly, sentinels do not conflict with *addAt*, so they allow to construct translation algorithm that meets property 2).

Specification 3 specifies *F-translate* algorithm which leverages sentinel positions. We discuss hereafter how this algorithm meet the ordering requirement 3).

The algorithm is recursive and delegates translation of each position into appropriate subroutine depending on the case. We discuss each of these cases with the help of Figure 4 displaying example execution.

Let us first consider the essential case, where translated p^e was not in the final state in $x^{e.F}$. The algorithm always looks for the *farthest non-final ancestor* (FNA, line 3) of p^e, i.e. an immediate child of some final node. It is enough to consider only this FNA node and translate the whole subtree along without any modification.[5] There are two principal cases for translation of a FNA position:

1. **FNA does not have any final left mini-sibling** (subroutine *translateByFinalParent*)
   This is the simplest case. Intuitively, if the FNA is a right child of a final node, the algorithm places the FNA just after its parent using *sentinelAfter* with $i = 0$, thus preserving the order.
   In the example from Figure 4, replica x_1^e translates p_q^e and p_u^e. FNA of p_q^e is p_u^e, so it is the root for the whole tree to translate. p_u^e along with its subtree are put as the first *sentinelAfter* of its (final) parent p_i^e.
2. **FNA does have a final left mini-sibling** (subroutine *translateByFinalLeftMiniSibling*)
   When the FNA position has a *final left mini-sibling* (LMS), translation requires more attention. Note that one cannot use the strategy described above here — let us consider example of replica x_2^e. It is because the FNA (e.g. p_r^e) not only needs to be ordered after the LMS (p_l^e) but also after the whole subtree of the LMS, as it took place in the epoch e. While in the epoch e it was enough to achieve that by ordering these two nodes at disambiguators (mini-siblings) level, in the new epoch it might not be, due to a new tree shape. If there was any *right descendant* RD of the LMS, $p_{LMS}^e < p_{RD}^e$ (like p_i^e or p_o^e), that is now in a different subtree in the newly balanced tree, the FNA must also be ordered after such RD.
   For this reason, the algorithm searches for the *rightmost final position* in the LMS subtree (line 37). The FNA is put after this rightmost position. However, there might be several different final positions having the same rightmost final node (in our example: p_l^e and p_i^e). In consequence, levels of mini-siblings ending up in the same rightmost element need to be ordered after such rightmost final node appropriately. For this purpose, we denote a set of final nodes sharing the same rightmost final position as a *right final edge*. Each right edge's rightmost node reserves a number of sentinel positions (computed by *reservedForRightEdgeMiniSiblings* using $x^{e.F}$) — one for every final node from the right edge. The closer mini-sibling to the root of the right edge the farthest sentinel position it gets (cf. line 39).
   In the considered example, p_l^e and p_i^e constitute right final edge sharing the same rightmost (p_i^e) and topmost (p_l^e) positions. The righmost p_i^e is reponsible for reserving 2 sentinel positions for potential mini-siblings. Mini-siblings lower in the right edge subtree (p_o^e) are ordered first (*sentinelAfter*$(x^{e.F}, p_i^e, 1)$), while those closer to the root (p_r^e) are put next (*sentinelAfter*$(x^{e.F}, p_i^e, 2)$), thus respecting existing the order.

[5] Due to the fact that state of the replica is always a tree (causality), there will be no final node within subtree of FNA, so it is safe.

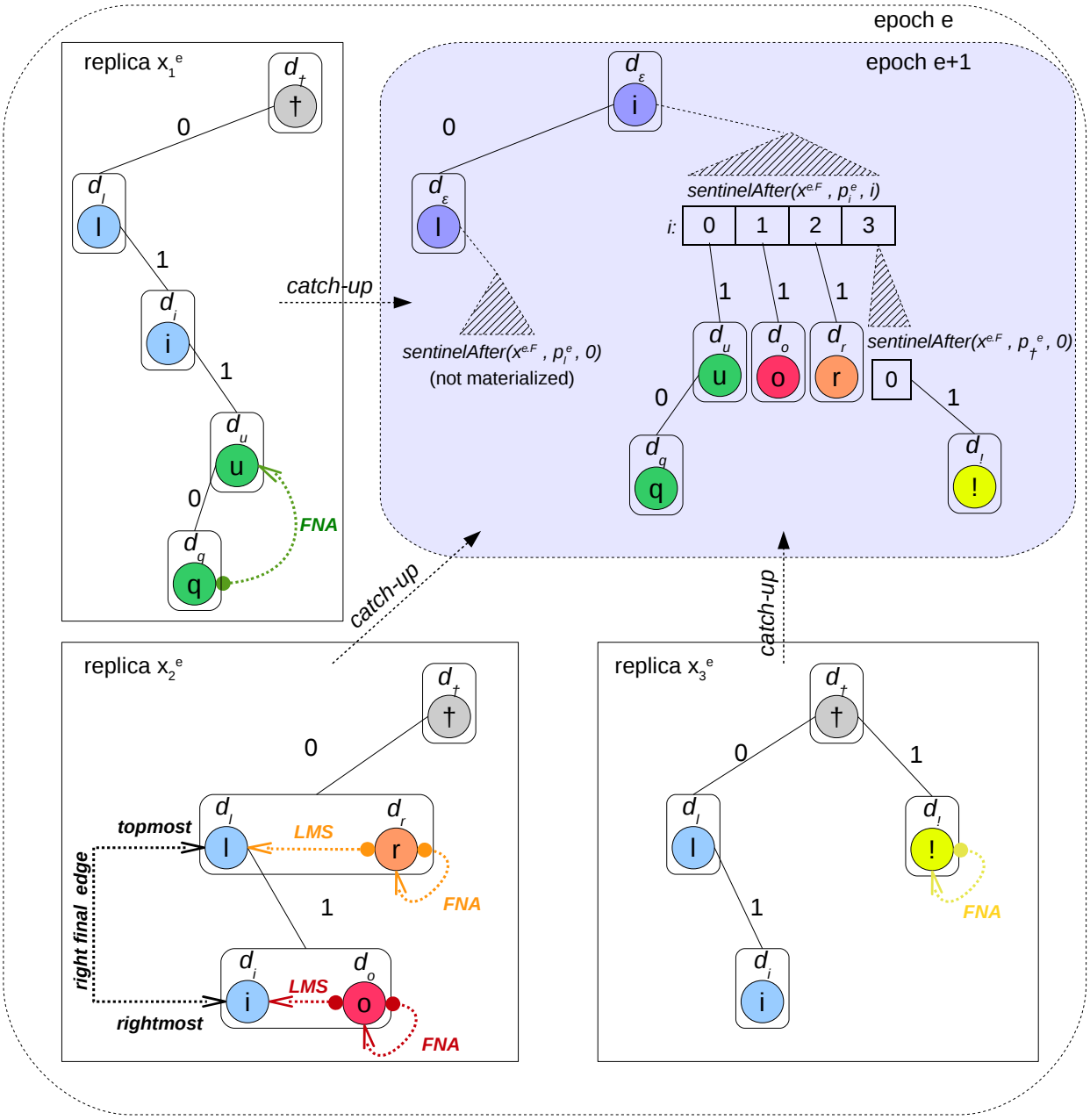


Figure 4: Three nebula replicas using F-translate to catch-up independently, preserving order within each catching-up replica as well as potentially observed inter-replica order. Replica x_1^e catches up with content `"liqu"`, x_2^e with content `"lior"` and x_3^e with `"li!"`. Initial state in the core after rebalance was `"li"`; after all nebula catch-up, they reach state `"liquor!"`. Note, that some sentinel positions are left unmaterialized, only reserved.

Specification 3 *F-translate* algorithm — simplified version, considering right positions only

query *translate* $(x^{e.F}, p^e) : p^{e+1}$
  if $(p^e, \_) \notin x^{e.F}$ **then**
    let $p^e_{FNA} = \mathit{farthestNonfinalAncestor}(x^{e.F}, p^e), p^e_{rel} : p^e_{FNA} \bullet p_{\mathit{relativeToFNA}} = p^e$
    if $\exists \mathit{nearestFinalLeftMiniSibling}(x^{e.F}, p^e_{FNA})$ **then**
      let $p^{e+1} = \mathit{translateByFinalLeftMiniSibling}(x^{e.F}, p^e_{FNA}) \bullet p_{\mathit{relativeToFNA}}$
    else if $\exists \mathit{nearestFinalRightMiniSibling}(x^{e.F}, p^e_{FNA})$ **then**
      let $p^{e+1} = \mathit{translateByFinalRightMiniSibling}(x^{e.F}, p^e_{FNA}) \bullet p_{\mathit{relativeToFNA}}$
    else
      let $p^{e+1} = \mathit{translateByFinalParent}(x^{e.F}, p^e_{FNA}) \bullet p_{\mathit{relativeToFNA}}$
  else
    let $c = (p^e, c) \in x^{e.F}$
    if $c = \dagger$ **then**
      let $p^{e+1} = \mathit{translateEmptyNode}(x^{e.F}, p^e)$
    else
      let p^{e+1} : given by $x^{e.F}.\mathit{rebalance}$
query *translateByFinalParent* $(x^{e.F}, p^e) : p^{e+1}$
  $p^e_{\mathit{parent}}, p_{\mathit{relative}} : p^e_{\mathit{parent}} / p^e \wedge p^e = p^e_{\mathit{parent}} \bullet p_{\mathit{relative}}$
  if $p^e > p^e_{\mathit{parent}}$ **then**
    let $p^{e+1} = \mathit{sentinelAfter}(x^{e.F}, p^e_{\mathit{parent}}, 0) \bullet p_{\mathit{relative}}$
  else
    ▷ Omitted for brevity: symmetric with above
query *rightmostFinalInSubtree* $(x^{e.F}, p^e) : p^e_s$
  let $P = \{p'^e : (p'^e, \_) \in x^{e.F} \wedge p'^e > p^e \wedge p^e / p'^e\}$
  if $P = \varnothing$ **then**
    let $p^e_s = p^e$
  else
    let $\mathit{rightmostFinalInSubtree}(x^{e.F}, \max(P))$
query *rightFinalEdge* $(x^{e.F}, p^e)$: set of *PosID* P (tree path)
  let $P = \{p'^e : (p'^e, \_) \in x^{e.F} \wedge \mathit{rightmostFinalInSubtree}(x^{e.F}, p'^e) = \mathit{rightmostFinalInSubtree}(x^{e.F}, p^e)\}$
query *reservedForRightEdgeMiniSiblings* $(x^{e.F}, p^e)$: integer i
  if $\max(\mathit{rightFinalEdge}(x^{e.F}, p^e)) = p^e$ **then**
    let $i = |\mathit{rightFinalEdge}(x^{e.F}, p^e)|$
  else
    let $i = 0$
query *translateByFinalLeftMiniSibling* $(x^{e.F}, p^e) : p^{e+1}$
  let $p^e_{LMS} = \mathit{nearestFinalLeftMiniSibling}(x^{e.F}, p^e_{FNA}), p_{\mathit{relativeToLMS}} : p^e_{LMS} \bullet p_{\mathit{relativeToLMS}} = p^e$
  let $p^e_{\mathit{rightmost}} = \max(\mathit{rightFinalEdge}(p^e_{LMS}))$
  let $p^e_{\mathit{topmost}} = \min(\mathit{rightFinalEdge}(p^e_{LMS})), \mathit{distanceFromTopmost} : p^e_{\mathit{topmost}} \bullet c_1 \bullet \ldots \bullet c_{\mathit{distanceFromTopmost}} = p^e_{LMS}$
  let $i = \mathit{reservedForRightEdgeMiniSiblings}(p^e_{\mathit{rightmost}}) - \mathit{distanceFromTopmost}$
  let $p^{e+1} = \mathit{sentinelAfter}(x^{e.F}, p^e_{\mathit{rightmost}}, i) \bullet p_{\mathit{relativeToLMS}}$
query *translateByFinalRightMiniSibling* $(x^{e.F}, p^e) : p^{e+1}$ ▷ Omitted for brevity: symmetric to above
query *translateEmptyNode* $(x^{e.F}, p^e) : p^{e+1}$
  let $p^e_{\mathit{nearestFinalWithAtom}} : \max(\{p'^e : (p'^e, \_ \neq \dagger) \in x^{e.F} \wedge p'^e < p^e\})$ ▷ Corner case handling omitted for brevity
  let $\mathit{emptyNodesBetween} = |\{p^e_\dagger : (p^e_\dagger, \dagger) \in x^{e.F} \wedge p^e_{\mathit{nearestFinalWithAtom}} < p^e_\dagger < p^e\}|$
  let $i = \mathit{reservedForRightEdgeMiniSiblings}(p^e_{\mathit{nearestFinalWithAtom}}) + \mathit{emptyNodesBetween} + 1$
  let $p^{e+1} = \mathit{sentinelAfter}(x^{e.F}, p^e_{\mathit{nearestFinalWithAtom}}, i)$

We will now describe how final nodes are translated. The final non-empty node case is trivial, defined by *rebalance*. A more complex case is a final tombstone, as in replica x^e_3 from the considered example. Since a final empty node never reappears in the initial state $x^{e+1.I}$ (one of the goals is to discard empty nodes) when the algorithm encounters its non-final children (here: p^e_1) or mini-siblings, there must be another place to reintroduce them. The final empty node needs to be reintroduced as sentinel, respecting relative position from epoch e.

The strategy, as defined in *translateEmptyNode*, is to find the *nearest final node with atom* (line 43) and reintroduce the empty node after it. However, there could be more than one final tombstone position to reintroduce by the same node. Hence, each final node with atom is responsible for a number of final empty nodes that could be potentially reintroduced. This number is computed basing upon $x^{e.F}$ to reserve sentinel positions[6], used for each potential empty node that need to be recreated (line 44). Note, that reintroduced empty nodes needs to be ordered appropriately with regard to potential non-final mini-siblings and children of the same node (*sentinelAfter* is feeded with an appropriate i).
Replica x_3^e from the example needs to recreate $p_\dagger^e$ to translate $p_!^e$ .Position p_i^e is responsible for reserving 1 sentinel for $p_\dagger^e$ reintroduction (beside of 1 for its children, 2 for mini-siblings: 4 in total), namely $\mathit{sentinelAfter}(x^{e.F}, p_i^e, 3)$. In case when tombstone position is reintroduced, second layer of sentinel node is used ($\mathit{sentinelAfter}(x^{e.F}, p_\dagger^e, 0)$) to put $p_!^e$.
Such a translation algorithm meets the defined safety requirements: it predestinates each position using only commong knowledge (1), it does not overlap with added positions by using sentinel positions (2), it carefully maintains any order from the previous epoch (3).

4.1. Tree growth

Hitherto considerations were related to the algorithm safety. Now we discuss the evolution of the tree size. We observe that in our approach to position translation, we need to introduce some empty nodes when sentinel position get materialized. The question arises how to limit the number this introduced nodes and and whether their number is smaller than the number of nodes discarded during rebalance?
Theorem 1. *If no new updates are generated, all replicas will eventually reach balanced tree without empty nodes.*

Proof sketch. Let assume that we begin with the state where there are some empty nodes in the tree. After rebalance in the core, these nodes are discarded. Since there is no new *removeAt* submitted, empty nodes will not appear this way. However, there might be some old updates reaching the core by nebula replicas catching-up pervasively. This translated updates may generate empty nodes. However, eventually all such empty nodes will be also discarded by some subsequent rebalance. □

A more generic case, when updates are continously submitted is very complex to analyze, and remains partially an open question. In order to reduce the number of introduced empty nodes, an implementation of *sentinelAfter* uses balanced tree with sentinel positions in leaves. This way, to use one position, $O(\log i_{max})$ empty nodes is created. We designed *sentinelAfter* for a special case when $i = 0$, which is hopefully prevailing as it corresponds to lack of concurrency. When $i = 0$, *sentinelAfter* introduces only $O(1)$ empty nodes (using short escape code). We also found some way to avoid situations where $i \neq 0$ for *sentinelAfter*. Notice that origins of introduction of empty nodes can be classified into two categories:

1. Introduced due to asynchrony combined with tree compaction attempt,
2. Introduced regardless of asynchrony.

The second category can be optimized. The reason of adding big number of empty nodes can be *allocateIDBetween* algorithm creating either tombstone's child or mini-siblings, which risks costly translation if it is not part of the final state. Therefore, we use modified *allocateIDBetween* definition that tries to avoid these cases when possible (although it is not always possible).

4.2. Implementation

By reusing parts of implementation by Leția [2], we have built a prototype of binary tree-based Treedoc. This implementation consist of around 1,800 LOC in Java, whereas around 700 LOC is strictly related to the catch-up in the core-nebula architecture.
As we focused on the proof-of-concept, we simplified communication patterns. The core-nebula is implemented assuming 1 core replica, or external agreement abstraction, and unrestricted number of nebula replicas. We implemented catch-up protocol with presented translation, together with heuristic for *allocateIDBetween* to avoid siblings and tombstones; *sentinelAfter* uses balanced trees.
The prototype does not use causal delivery. Instead, we use retransmission protocol that guarantees only essential properties: when a replica receives a new position and it does not have all the ancestors in the tree, it asks for interval of operations (part of a path) that it is missing. Any site can answer that

[6] Details are not shown in the algorithm, but line 44 gives a clue.

request by sending missing operations. At the beginning of catch-up, we do transfer whole $x^{e \cdot F}$ state, without any optimization so far.
Together with implementation we have developed an extensive automated testing suite, which we used to verify safety in various corner cases of the catch-up protocol implementation. Our framework is suitable for further experiments. It could be also easily used for non-blocking rebalance (for instance, collaborative editing application) where an application always accesses nebula replica, so it never blocks.

5. Related work

This work is related to general problems of optimistic replication and garbage collection, as well as collaborative editing, which all have already been subject of studies.
The problem of tombstones is recognized in distributed systems since a long time [1]. We have recently proposed a comprehensive portfolio of CRDTs, and learned that metadata accumulation is a common problem related to commutativity [9]. To overcome such issues, Wuu and Bernstein proposed a *stability detection* algorithm [15] — one that allows to detect when every site learned about operation leaving some unnecessary data (e.g. tombstone), so it can be discarded. This algorithm requires membership management and (eventual) connectivity between sites to reach remain live. Clearly, if one wants to remain safe she cannot discard tombstones if some replicas are not responsive — which is often the case in a large scale system. For this reason, our work solves a slightly different problem. Instead, we try to accept asynchrony and utilize the fact that a subset of replicas may experience good connectivity: they could at least use more compacted representation, which incurs less communication and access time overhead. The remaining replicas can switch to such compacted representation later. Therefore, the problem we consider is somewhat orthogonal to the classic problem of memory occupation by tombstones, which is still useful and, for instance, hidden in delivery mechanism we assume.
Nevertheless, one may attempt to discard tombstones as a way to compact a tree. Use of the aforementioned Wuu's algorithm was proposed to discard tombstones in an early version of Treedoc [8]. Later Preguiça evaluated larger, but globally unique disambiguators (replica identifier and counter); using them, removed positions can be discarded immediately [4]. The same idea has been used by another CRDT, Logoot [11]. Leția noticed that even if only replica id is used as a disambiguator, it is enough to store tombstone only at the site owning this id [2]. However, none of these approaches solve the problem of tree unbalance, nor do they resolve the problem that descendants of discarded node still contain "discarded" position as a prefix of their identifier.
A different solution is to avoid the problem: avoid tree unbalance in the sequence implementation. Wu and Pu, inspired by Treedoc, propose another implementation of replicated continuum using rational numbers [14]. Naturally, their solution also require kind of rebalance, but they do not provide any novel technique to achieve it. Weiss et al. give an interesting insight. For Logoot continuum implementation, they consider ingenius heuristics for *allocateIDBetween* (a generalization of preallocating a subtree in Treedoc) to avoid early tree unbalance [11, 12]. These heuristics however, come at the price of an anomaly: concurrently inserted sequences can be intermixed. Nonetheless, we believe our approach could be adapted to Logoot.
Another interesting approach to replicated buffer design different than continuum is RGA by Roh et al. [5]. Roh uses explicit pointers to previous positions, expressed by constant size Lamport clocks (with replica identifier). Tree issues are not applicable there. RGA still poses classic problem of tombstones, resolved by Wuu's algorithm for well-connected systems. RGA uses linked list as a memory representation. One may try to optimize it and aggregate buffer into bigger blocks by performing operation similar to rebalance.

6. Conclusions

We have presented a detailed specification of catch-up protocol for the core-nebula architecture. We have shown new concerns related to correctness of the algorithm used during catch-up, which have not been previously considered. Finally, we have proposed a solution for catch-up translation algorithm and built a prototype implementation on this basis.
We plan to focus now on analyzing the tree size evolution — whether we can show that the tree size

eventually decreases in all or most of the cases. We are working on verifying it experimentally using generated workload of collaborative editing sessions.
If our solution proofs itself useful in practice, we believe it is a significant step towards application of sequence CRDT, such as Treedoc or Logoot, for collaborative editing and other applications. Furthermore, it encourages us to design and evaluate other CRDTs using similar scheme to perform rare operations requiring synchronization.

The bibliography

[1] Paul R. Johnson and Robert H. Thomas. The maintenance of duplicate databases. Internet Request for Comments RFC 677, Information Sciences Institute, January 1976.
[2] Mihai Leția, Nuno Preguiça, and Marc Shapiro. CRDTs: Consistency without concurrency control. In *SOSP W. on Large Scale Distributed Systems and Middleware (LADIS)*, volume 44 of *Operating Systems Review*, pages 29–34, Big Sky, MT, USA, October 2009. ACM SIG on Operating Systems (SIGOPS), Assoc. for Comp. Machinery.
[3] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Data consistency for P2P collaborative editing. In *Int. Conf. on Computer-Supported Coop. Work (CSCW)*, pages 259–268, Banff, Alberta, Canada, November 2006. ACM Press.
[4] Nuno Preguiça, Joan Manuel Marquès, Marc Shapiro, and Mihai Leția. A commutative replicated data type for cooperative editing. In *Int. Conf. on Distributed Comp. Sys. (ICDCS)*, pages 395–403, Montréal, Canada, June 2009.
[5] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. Replicated abstract data types: Building blocks for collaborative applications. *Journal of Parallel and Dist. Comp.*, (To appear) 2011.
[6] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1):42–81, March 2005.
[7] R. Schwarz and F. Mattern. Detecting causal relationships in distributed computations: In search of the Holy Grail. *Distributed Computing*, 3(7):149–174, 1994.
[8] Marc Shapiro and Nuno Preguiça. Designing a commutative replicated data type. Rapport de recherche RR-6320, Institut Nat. de la Recherche en Informatique et Automatique (INRIA), Rocquencourt, France, October 2007.
[9] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Rapport de recherche 7506, Institut Nat. de la Recherche en Informatique et Automatique (INRIA), Rocquencourt, France, January 2011. http://hal.archives-ouvertes.fr/inria-00555588/.
[10] Werner Vogels. Eventually consistent. *ACM Queue*, 6(6):14–19, October 2008.
[11] Stéphane Weiss, Pascal Urso, and Pascal Molli. Logoot: a scalable optimistic replication algorithm for collaborative editing on P2P networks. In *Int. Conf. on Distributed Comp. Sys. (ICDCS)*, Montréal, Canada, June 2009. see app:rep:1652.
[12] Stephane Weiss, Pascal Urso, and Pascal Molli. Logoot-undo: Distributed collaborative editing system on P2P networks. *IEEE Trans. on Parallel and Dist. Sys. (TPDS)*, 21:1162–1174, 2010.
[13] Qinyi Wu and Calton Pu. Consistency in real-time collaborative editing systems based on partial persistent sequences. Technical report, Georgia Institute of Technology. College of Computing, Atlanta, GA, USA, 2009.
[14] Qinyi Wu, Calton Pu, and João Eduardo Ferreira. A partial persistent data structure to support consistency in real-time collaborative editing. In *ICDE*, pages 776–779, 2010.
[15] Gene T. J. Wuu and Arthur J. Bernstein. Efficient solutions to the replicated log and dictionary problems. In *Symp. on Principles of Dist. Comp. (PODC)*, pages 233–242, Vancouver, BC, Canada, August 1984.