A Scalable Sequence Encoding for Massive Collaborative Editing

Brice Nédelec[a,*], Pascal Molli[a,*], Achour Mostefaoui[a,*]

[a] *Université de Nantes, LINA, 2, rue de la Houssinière, BP 92208, 44322 Nantes Cedex 3, France*

Abstract

The recent development of distributed collaborative editors such as Google Docs, SubEthaEdit, Etherpad, has intensified the interest in an efficient data structure which allows a massive number of users to write documents. However, (i) centralised editors bring single-point-of-failure, privacy issues, economic intelligence issues, and restrictions in terms of service. (ii) Both centralised and decentralised approaches lack in scalability: number of users, concurrency rate, size of documents. The Conflict-free Replicated Data Type (CRDT) for distributed sequences alleviates most of the aforementioned issues. Yet, state-of-the-art CRDTs representing the document have an unsatisfying space complexity. Either they use tombstones where deleted elements are only hidden to the user (e.g. an empty document contains 1 million tombstones if 1 million deletions have been performed) or they allocate a variable-size identifier to each element (e.g. a document where 1 million insertions have been performed may contain identifiers of length 1 million). In the latter, the allocation of the identifiers is crucial to maintain good performance however state-of-the-art allocation functions are application dependent and provide linearly upper-bounded identifiers. Fortunately, the recent allocation function LSEQ provides identifiers empirically enjoying a sub-linear space complexity without being application specific. Knowingly, contributions of this paper are threefold: (i) it provides the proof of the polylogarithmic space complexity of LSEQ, (ii) it provides all the outlines to develop distributed collaborative editors that scales in all the aforementioned dimensions. Using these outlines, we developed a prototype of CollaboRATive Editor CRATE, (iii) it evaluates CRATE on large scale experiments ran on the Grid'5000 testbed. The results show that such editors, being decentralised and scalable, could constitute a serious competitor to current trending editor. In particular, it allows massive collaborative authoring of huge documents which opens the field to new kinds of distributed applications.

Keywords: Document authoring, distributed collaborative editing, optimistic replication, polylog sequence encoding, Conflict-free Replicated Data Types.

1. Introduction

In recent years, the interest over distributed collaborative editors such as Google Docs, SubEthaEdit, or Etherpad, has not ceased to increase. Such editors allow distributing the work across space, time and organisations. Despite being undoubtedly useful, they have flaws. When centralised, they bring single-point-of-failure issues, privacy issues, economic intelligence issues, limitations in terms of service, etc. For instance Google [1] allows only 50 users to write a document together at a same time. The decentralised approaches solve most of these problems, but the performance issues remain: the algorithms depend on the number of users, the concurrency between operations, the number of operations, etc. These scalability issues preclude the existence of a massive distributed collaborative editor.

Commonly, distributed collaborative editors use the optimistic replication scheme [2, 3] to provide high availability and responsiveness of documents. (i) Locally, each user involved in the collaboration owns a replica of the document, directly modifies it, and broadcast the changes. (ii) Remotely, each user integrates the received operation to her replica. The convergence property of optimistic replication states that replicas are allowed to temporarily diverge. Yet, the resulting replicas become identical when all collaborators have received all changes [4].

Historically, the Operational Transformation (OT) approaches are the most ancient approaches to build distributed collaborative editors. However, the time complexity of the integration of operations is upper-bounded by $O(H^2)$ where H is the number of concurrent operations in the logfile relatively to the integrated one. Consequently, all collaborators suffer from few concurrent operations making OT approaches impracticable because of unreliable latency.

On the other side, the Conflict-free Replicated Data Type [5, 6] (CRDT) for sequences constitutes a simple mean to build distributed collaborative editors that scales in presence of concurrent operations. Contrarily to OT, most of the processing time is located in the local part of operations, making CRDTs approaches theoretically better since each operation leads to N remote integration (N being the number of peers). To ensure convergence,

[*] *Corresponding authors:* `first.last@univ-nantes.fr`

CRDTs allocate a unique and immutable identifier to each element (e.g. character) in the sequence (i.e. document). The set of identifiers is paired with a total order, and this latter actually makes the sequence.

Nevertheless, CRDTs for sequences hide their complexity in the memory usage. Indeed, CRDTs for sequences use an allocation function for their identifiers and provide two updating operations: insert and delete. However, either depending on the type of the performed operations [7, 8, 9, 10, 11, 12, 13, 14, 15], or the arguments of the performed operation [11, 16, 17], the space overhead induced by the identifiers can be prohibitively high. This directly impacts performance. Unfortunately, the allocation is crucial but a non-trivial problem.

In this paper, we focus on LSEQ [18, 19]: an allocation function that empirically provides identifiers enjoying a sub-linear upper bound on their size. Nevertheless, the sub-linear space complexity of LSEQ was only a conjecture confirmed by simulations [18]. Contributions of this paper are threefold:

- It provides a proof of the polylogarithmic space complexity of LSEQ.
- It provides all the outlines to develop a distributed collaborative editor which allows massive editing of large documents in real-time. Using these outlines, we built a prototype of CollaboRATive Editor called CRATE.
- It validates the space complexity of LSEQ and the scalability of CRATE on experiments. Parts of theses experiments ran on the Grid'5000 testbed and involved up to 450 emulated peers creating a document of half a million characters at a global rate of 166 operations per second. It shows that CRATE scales in terms of: number of users, concurrency rate, and size of documents.

The remainder of this paper is organised as follows: Section 2 provides the necessary background to understand CRDTs for sequences and the motivations. Section 3 follows with a description of LSEQ and the proof of its space complexity. Section 4 it highlights the scalability of CRATE while validating the space complexity analysis of LSEQ. Section 5 reviews related work. Finally, Section 6 concludes this paper and discusses perspectives.

2. Preliminaries

Conflict-free Replicated Data Type approaches belong to the optimistic class of replication. Thus, each peer involved in the collaboration owns a local replica of the CRDT. A peer directly performs operations on its replica and broadcasts the results of these operation to all peers (including itself) where they are integrated. CRDTs provide strong eventual consistency [5] upon the assumption of the eventual delivery of operations, i.e., they guarantee that all replicas will eventually converge to an identical state when the system becomes quiescent.

In this paper, we focus on CRDTs for sequences, the closest structure corresponding to a document. A CRDT for sequences is an abstract data type that provides two commutative operations to update a sequence: insert and delete. Indeed, these operations can be integrated in any order while it fulfils the invariant: the deletion of an element is integrated after its insertion.

When a peer performs an insert operation, its CRDT for sequences allocates a unique and immutable identifier encoding the total order among the elements. Thus, an element corresponds to each identifier.

For instance, let us consider a sequence $QWTY$ with the unique, immutable, and totally ordered integer identifiers 1, 2, 4, 8 respectively. A peer inserts the element E between W and T. The natural identifier that comes to mind is 3. The resulting sequence is $QWETY$. Then the peer inserts R between E and T. However, since 3 and 4 as integers are contiguous, the space of identifiers must be enlarged to handle the new insertion. Hence, the decimal number 3.1 is allocated to the character R. If the peer inserts a new character between E and R, a new identifier will be allocated between 3 and 3.1. Again, the space will be extended resulting in a new identifier 3.0 suffixed by any integer. Let X be the suffix, the order is preserved since ($3 < 3.0.X < 3.1$). Such growing identifiers are called variable-size identifiers. The main objective is about keeping the growth under acceptable boundaries.

2.1. Variable-size Identifiers

Variable-size identifiers CRDTs for sequences use identifiers that can grow. Each of these identifiers can be represented as a concatenation of basic elements (e.g. integers). The resulting sequence can be represented by a tree where the elements of the sequence are stored at the nodes and where the edges of the tree are labelled such that a path in the tree from the root to a node represents the identifier of the element stored at this node. For instance, the character R from the previous example is accessible following the path 3 then the path 1. More formally, a variable-size identifiers CRDT for sequences incrementally builds a tree $\mathcal{T}$ where the leaves or the intermediary nodes contain a subset of the elements of the sequence (over an alphabet $\mathcal{A}$). Thus, the tree $\mathcal{T}$ is (i) a set of pairs $\langle \mathcal{P} \subset \{\mathbb{N}\}^*, \mathcal{A} \rangle$, i.e., each element has a path. (ii) A total order $(\mathcal{P}, <_{\mathcal{P}})$, i.e., an ordering among the paths which allows to retrieve the sequence of elements.

EXAMPLE 1: Fig. 1 shows the underlying 10-ary tree $\mathcal{T}$ of a variable-size identifiers CRDT for sequences. Like in the previous scenario, the initial sequence is $QWTY$ with the respective paths [1], [2], [4] and [8]. Then, a peer inserts the character E between the pairs $\langle [2], W \rangle$ and $\langle [4], T \rangle$ resulting in the following pair: $\langle [3], E \rangle$. When the peer inserts the character R, there is not enough space at the first level of the tree for a new path. Hence, the

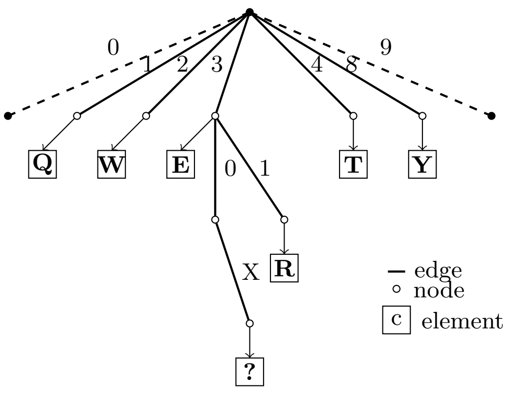

Figure 1: Underlying 10-ary tree $\mathcal{T}$ of a variable-size identifiers CRDT for sequences. The paths $\mathcal{P}$ correspond to the concatenation of the edges from the root to the elements. The elements are characters. Using the total order ($\mathcal{P}$, $<_\mathcal{P}$), the sequence of elements is $QWERTY$.

depth of the tree must grow. The resulting path is [3.1]. Another insertion between the elements E and R would increase the depth of the new path. The path would be [3.0.X] where $0 < X < 10$. Using the total order ($\mathcal{P}$, $<_\mathcal{P}$) allows retrieving the sequence $QWERTY$.

2.2. Disambiguation of concurrent cases

Two peers concurrently performing an operation on their respective replica may get different results after the integration of each other's operation. Indeed, ($\mathcal{P}$, $<_\mathcal{P}$) is a total order when a single peer edits, however, it becomes a partial order when the editing involves multiple peers. Consequently, disambiguate the order among such elements is necessary. For this purpose, the disambiguation function δ associates a disambiguator to each pair of $\mathcal{T}$. (iv) $\delta : \mathcal{P} \times \mathcal{A} \to \mathcal{D}$. (v) A total order ($\mathcal{D}$, $<_\mathcal{D}$). The function δ is an accessor to additional values that are definitely totally ordered even in presence of concurrency. Finally, the pairs in $\mathcal{T}$ can be totally ordered with a composition of the total orders ($\mathcal{P}$, $<_\mathcal{P}$) and ($\mathcal{D}$, $<_\mathcal{D}$). The composition leads to (vi) a total order ($\mathcal{T}$, $<_\mathcal{T}$).

Example 2: Fig. 2 depicts a tree containing 6 elements with only 5 distinct paths. Similarly to the previous example, the initial sequence is $QWTY$, however, in this example, two peers concurrently insert the element E and R between the pairs $\langle[2], W\rangle$ and $\langle[4], T\rangle$. In both cases, the resulting path is [3]. When they integrate the resulting pair from each other, the sequence becomes either $QWERTY$ or $QWRETY$. Nevertheless, let $\delta(\langle[3], R\rangle) = \delta_R$ and $\delta(\langle[3], T\rangle) = \delta_T$, in our example, $\delta_R <_\mathcal{D} \delta_T$. Hence, using the total order ($\mathcal{T}, <_\mathcal{T}$), the resulting sequence becomes $QWERTY$. It is worth noting that disambiguators are usually computed using a monotonically increasing value and a unique peer identifier. Therefore, a peer cannot directly influence the final position of the character in the sequence using disambiguators. This means that the sequence of the example could have ended in $QWRETY$ and it would have needed a correction.

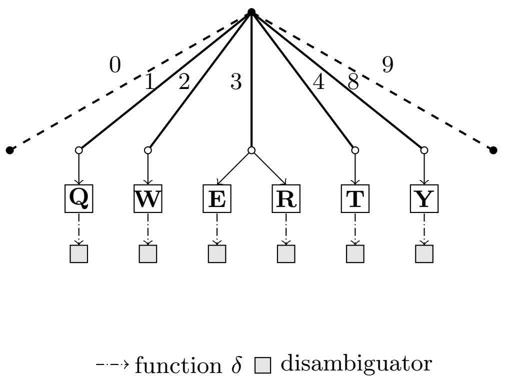

Figure 2: The 10-ary tree of a variable-size identifiers CRDT for sequences including concurrent insertions. Using only ($\mathcal{P}$, $<_\mathcal{P}$) could lead to either $QWERTY$ or $QWRETY$. The disambiguators forces the final result to converge to $QWERTY$.

2.3. Choosing the rightful path

The most critical part of variable-size identifiers CRDTs for sequences consists in creating the paths. Algorithm 1 shows the general outlines of variable-size identifiers CRDTs for sequences. It divides the operations insert and delete between the local and remote phases of the optimistic replication paradigm. As we can see, most of the complexity of the algorithm is located in the local part of the *insert* operation where the algorithm must generate a path (cf. Line 6) and a disambiguator (cf. Line 7).

Algorithm 1 General outlines of variable-size identifiers CRDTs for sequences

```
 1: INITIALLY:
 2:     $\mathcal{T} \leftarrow \varnothing$;            ▷ structure of the CRDT for sequences
 3:
 4: LOCAL UPDATE:
 5:     on insert ($p \in \mathcal{I}$, $\alpha \in \mathcal{A}$, $q \in \mathcal{I}$):
 6:         let $path \leftarrow allocPath(p.P,\ q.P)$;
 7:         let $dis \leftarrow allocDis(p,\ path,\ q)$;
 8:         $broadcast('insert',\ \langle path,\ \alpha,\ dis\rangle)$;
 9:     on delete ($i \in \mathcal{I}$):
10:         $broadcast('delete',\ i)$;
11:
12: RECEIVED UPDATE:
13:     on insert ($i \in \mathcal{I}$):          ▷ once per distinct triple in $\mathcal{I}$
14:         $\mathcal{T} \leftarrow \mathcal{T} \cup i$;
15:     on delete ($i \in \mathcal{I}$):  ▷ after the remote $insert(i)$ is done
16:         $\mathcal{T} \leftarrow \mathcal{T} \setminus i$;
17:
```

Let $\mathcal{I}$ be the set of unique triples $\mathcal{I} : \mathcal{P} \times \mathcal{A} \times \mathcal{D}$. For all i in $\mathcal{I}$, let $i.P$, $i.A$, $i.D$ be the respective accessors to the path, the element, and the disambiguator of i. The function *allocPath* chooses a path in the tree between two other paths $p.P$ and $q.P$ where $p <_\mathcal{T} q$. However, since the tree can always have sub-trees, the number of possible

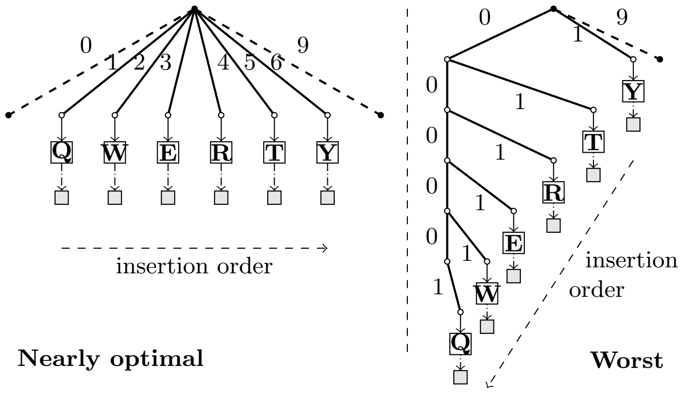

Figure 3: Two trees filled with the resulting identifiers of two different permutations resulting in an identical sequence $QWERTY$. They use the same function *allocPath* which allocates the leftmost branch in the tree. All paths of the nearly optimal case have a length of 1 while the tree of the worst case grows up to a depth of 6.

paths is infinite, and so is the number of *allocPath* functions. Nevertheless, the function *allocPath* should choose among the available paths with the smallest length in order to keep the system with good performance. This observation reduces considerably the number of possible *allocPath* functions. Still, the allocation of paths without *a priori* knowledge of the final sequence is a non-trivial problem (Appendix A depicts the problem abstracted from the CRDT and editing context).

Example 3: Fig. 3 illustrates the difficulties of designing a function to allocate the paths. The figure represents the underlying trees of two variable-size identifiers sequences using an identical allocation function: they allocate the leftmost branch available at the lowest depth possible. In both cases the final sequence is $QWERTY$, however, the editing sequences (i.e., the order of insertions of the characters) producing this outcome are different. The editing sequence can be represented as a sequence of pairs (*character*, *index*) where each character is sequentially inserted at the targeted index, shifting the right part of the string.

- In the good case, the permutation of the insert operations produces the following editing sequence: $[(Q,\ 0)$, $(W,\ 1)$, $(E,\ 2)$, $\ldots]$. Since the function *allocPath* allocates the leftmost branches, the resulting paths are: $\langle[1],\ Q\rangle$, $\langle[2],\ W\rangle$, $\langle[3],\ E\rangle$, etc. In this case, the depth of the tree never grows. In this regard, this function *allocPath* is close of the optimal allocation with such permutation.
- On the other side, the permutation of the insert operations produces the following editing sequence: $[(Y,\ 0)$, $(T,\ 0)$, $(R,\ 0)$, $\ldots]$. When the insertions occur, it chooses to allocate the leftmost branch of the tree. Thus, each insertion increases the depth of the tree. The latter becomes filled with the following pairs: $\langle[1],\ Y\rangle$, $\langle[0.1],\ T\rangle$, $\langle[0.0.1],\ R\rangle$, etc. Consequently, the average size of the paths quickly increases.

This example shows how the insertion order impacts on the length of the allocated paths. Unfortunately, the insertion order cannot be predicted, nor the size of the final sequence. Prior work on CRDTs for sequences often made the assumption of left-to-right editing due to observations made on corpus [11, 17]. However, there exist human edited documents that do not correspond to this kind of editing [18]. It sorely increases the size of the underlying structure representing the sequence. Knowingly, we are looking for an allocation function which provides identifiers with a sub-linear spatial complexity compared to the number of insertions whatever the permutations of insertions since we do not know which one will create the final sequence.

3. LSEQ

LSEQ (stands for polyLogarithmic SEQuence) is the name of the proposed allocation function. As such, any variable-size identifiers CRDTs for sequences can use it. LSEQ allocates identifiers with a space complexity polylogarithmically upper-bounded compared to the number of insert operations. Since CRDTs for sequences scale in number of peers and concurrency rate, building a CRDT-based distributed collaborative editor using LSEQ would allow massive collaborative editing of huge documents in real-time. This section describes LSEQ: the two functions *allocPath* and *allocDis*. Then, it provides the proof of the space complexity of its identifiers.

The allocPath function chooses which path will be included with the element in order to encode its relative position with regard to its adjacent elements in the sequence. For the sake of performance, the *allocPath* objective is to keep the underlying tree with a small depth. Three components compose *allocPath*. Each of these components fails to provide a usable allocation function. Nevertheless, their composition compensates their respective deficiency.

1. The first component is an exponential tree [20, 21]. Regarding the formalisation of Section 2, it means a restriction over $\mathcal{P}$. Indeed, the path is not a sequence of natural numbers, instead, the numbers starts within a subset of $\mathbb{N}$, and the size of this subset is doubled at each concatenation. For instance, let $p_1 \in \mathcal{P}$ such that $|p_1| = 1$, then $p_1 \subset \mathbb{N}_{<32}$. Let $p_2 \in \mathcal{P}$ such that $|p_2| = 2$, then $p_2 \subset \mathbb{N}_{<32}.\mathbb{N}_{<64}$ etc. Similarly, let i the cardinal of the subset $\mathcal{P}$ with one concatenation, let $p_n \in \mathcal{P}$ such that $n \in \mathbb{N}^+$ and $|p_n| = n$, then $p_n \subset \mathbb{N}_{<i}.\mathbb{N}_{<i*2} \ldots \mathbb{N}_{<i*2^{n-1}}$. Due to the growth of the subsets, such representation of the paths requires one additional bit to encode each concatenation. With the same examples, it implies that p_1 is encoded on $log_2(32) = 5$ bits, p_2 is encoded on $5 + 6 = 11$ bits, ... The total order $<_{\mathcal{P}}$ is similar to the lexicographic order.

We define it as: $\forall p_j, p_k \in \mathcal{P}$ with $|p_j| = j$, $|p_k| = k$. There exists an index $0 \leq l \leq min(|p_j|, |p_k|)$ such that $p_j = X.Y$ and $p_k = X.Z$ with $X \subset \{\mathbb{N}\}^l$, $Y \subset \{\mathbb{N}\}^{j-l}$, $Z \subset \{\mathbb{N}\}^{k-l}$. Finally, $p_j <_{\mathcal{P}} p_k$ iff $Y[1] < Z[1]$.

2. The *allocPath* function uses two sub-allocation functions. Both are designed for monotonic editing, i.e., inserting repeatedly at an adjacent position of the previously inserted element. Nevertheless, they are application dependent, i.e., one is specialised in end-editing (left-to-right) while the other is specialised in front-editing (right-to-left).

3. The *allocPath* function uses a third component to choose the sub-allocation function employed at each depth of the exponential tree. The choice is made randomly using a hash function following a uniform distribution. Such function does not favour any editing behaviour while remaining efficient. Furthermore, as shown in [19], using antagonist sub-allocation functions forces all peers to make identical choices. To reach that goal, they all use a similar hash function initialised with a same seed shared within the document.

Combining the three components provides a sub-linear space complexity on the size of identifiers compared to the number of insertions performed which makes it well-suited for text editing, and consequently, for distributed collaborative editing.

Algorithm 2 presents the three components of the function *allocPath*. At first, the latter gets the depth of the new path using the function *getDepthInterval* which also returns the interval between the arguments (i.e., the number of available paths at the processed depth). Then, by calling the hash function h, it defers the call to a sub-allocation function: either *endEditing* or $frontEditing$. These allocation functions are nearly identical. (#1) Both functions get the depth and the interval of the new path to allocate. (#2) Both functions limit the range of allocation of the new path. (#3) However, while the *endEditing* uses the previous identifier and adds a value to build the new path, the $frontEditing$ does the opposite. Thus, by #2 the allocation function is efficient in monotonic editing and by #3 the allocation is efficient in left-to-right or right-to-left editing. The unexplicit function *subPath* returns a truncated path from the root to the depth passed as argument.

Example 4: Similarly to the prior example (cf. Fig. 3), Fig. 4 depicts the resulting trees after two antagonist scenarios creating the sequence $QWERTY$: (i) the left-to-right editing sequence $[(Q, 0), (W, 1), \ldots]$ and (ii) the right-to-left editing sequence $[(Y, 0), (T, 0), \ldots]$. In both cases, LSEQ's exponential tree starts with an arity of 32 and doubles it at each level. Also, it uses the $frontEditing$ and *endEditing* sub-allocation functions at the first and the second level of the tree respectively. Since the first level of the tree uses the function *endEditing*, the scenario involving the left-to-right editing sequence results in a tree of depth 1. On the other hand, contrarily to the allocation function presented in the prior example (cf. Fig 3), the antagonist scenario does not sorely increase the depth of the tree. Indeed, the identifiers of LSEQ quickly reach a level of the tree where the sub-allocation function is designed to handle the right-to-left editing behaviour.

Algorithm 2 The *allocPath* function of LSEQ

```
1: let $boundary \leftarrow 10$;          ▷ Any constant
2: let $h : \mathbb{N} \rightarrow (\mathcal{P} \times \mathcal{P} \rightarrow \mathcal{P})$;   ▷ get sub-allocation function

3: function ALLOCPATH($p, q \in \mathcal{P}$) → $\mathcal{P}$
4:     let $depth$, _ ← $getDepthInterval(p, q)$;
5:     return $h(depth)(p, q)$;          ▷ Defers the call
6: end function

7: function ENDEDITING($p, q \in \mathcal{P}$) → $\mathcal{P}$
    ▷ #1 Get the depth of the new path
8:     let $depth, interval \leftarrow getDepthInterval(p, q)$;
    ▷ #2 Process a maximal space between two identifiers
9:     let $step \leftarrow min(boundary, interval)$;
    ▷ #3 Create the new path
10:    return $subPath(p, depth) + rand(0, step)$;
11: end function

12: function FRONTEDITING($p, q \in \mathcal{P}$) → $\mathcal{P}$
13:    let $depth, interval \leftarrow getDepthInterval(p, q)$;    ▷ #1
14:    let $step \leftarrow min(boundary, interval)$;              ▷ #2
15:    return $subPath(q, depth) - rand(0, step)$;                ▷ #3
16: end function

    ▷ Which depth has enough space for 1 path
17: function GETDEPTHINTERVAL($p, q \in \mathcal{P}$) → $\mathbb{N} \times \mathbb{N}$
18:    let $depth \leftarrow 0$; $interval \leftarrow 0$;
19:    while ($interval < 2$) do
20:        $depth \leftarrow depth + 1$;
21:        $interval \leftarrow subPath(q, depth) - subPath(p, depth)$;
22:    end while
23:    return $\langle depth, interval \rangle$;
24: end function
```

The allocDis function creates the disambiguators which have two missions. (i) They ensure the uniqueness of the triples, even in presence of concurrency. (ii) They guarantee that triples can always be inserted between two other triples even when the paths of the latter are identical. Example 3 illustrates the need of comparisons that preserve the order given by the paths and also examine the disambiguators. During the editing session, the insertion between the elements W and T resulted in an identical path: [29]. Therefore, without *allocDis*, the order among W and T is ambiguous, and inserting between them becomes impossible. Indeed, if any path $[29.X]$ is greater than [29], the newly inserted elements will always end up after the W and T in the sequence. Consequently, the disambiguators are necessary to build an allocation function for variable-size identifiers CRDTs for sequences.

A disambiguator is a list of pairs $\langle source, clock \rangle$. Similarly to the tree of paths, the set of all disambiguators

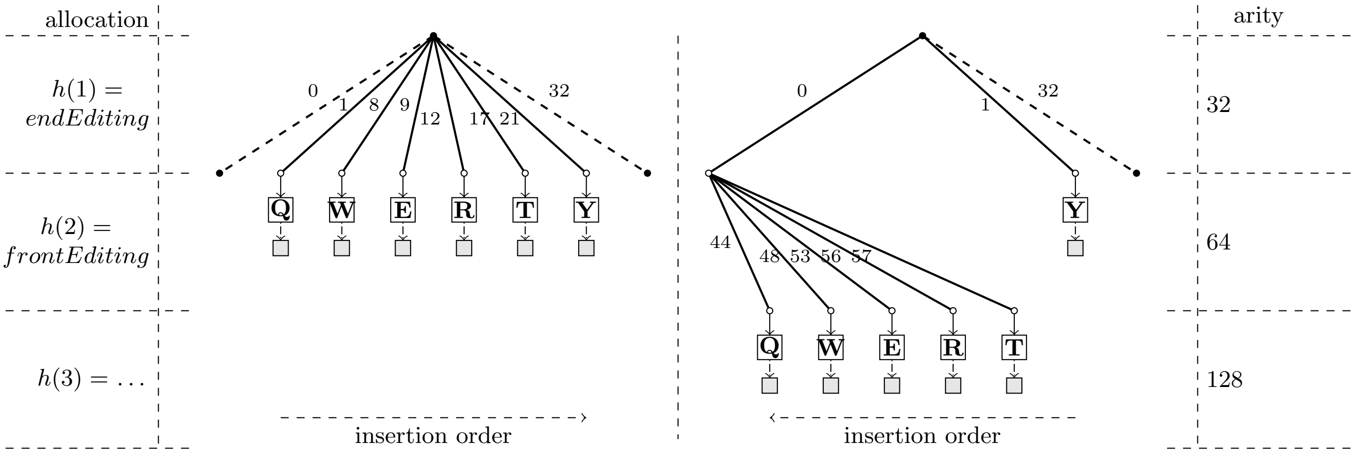

Figure 4: Example of LSEQ's exponential trees with two antagonist editing behaviours to create the sequence of characters $QWERTY$. Thanks to its hash function, LSEQ defers the allocations of paths to the sub-allocation functions designed for front-editing and end-editing at first and second level respectively. Furthermore, the arity is doubled at each level of the tree. Contrarily to the example of Fig. 3, the trees do not linearly grow.

can be represented as a tree equipped with a lexicographic total order $(\mathcal{D}, <_\mathcal{D})$. While the total order $<_\mathcal{P}$ ultimately compares two numbers, the total order $<_\mathcal{D}$ compares two pairs. With the same notation than *allocPath*, let $\langle s_1, c_1\rangle$ and $\langle s_2, c_2\rangle$ be the $l+1$ couples of the disambiguators d_j and d_k respectively. $d_j <_\mathcal{D} d_k$ iff $s_1 < s_2$, or if $s_1 = s_2$, $c_1 < c_2$.

Algorithm 3 The function *allocDis* of LSEQ

```
1: let site                                  ▷ the unique site identifier
2: let counter ← 0                           ▷ a local counter

3: function ALLOCDIS($p \in \mathcal{I}$, $path \in \mathcal{P}$, $q \in \mathcal{I}$) → $\mathcal{D}$
4:     let dis ← [ ];
5:     counter ← counter + 1;
6:     for i from 1 to |path| do
7:         dis[i] ← ⟨site, counter⟩;
8:         if path[i] = q.P[i] then dis[i] ← q.D[i];
9:         if path[i] = p.P[i] then dis[i] ← p.D[i];
10:    end for
11:    return dis;
12: end function
```

Algorithm 3 describes the allocation of a disambiguator which preserves the intention of the peer that performed the insertion. It uses a unique site identifier and a monotonically increasing counter. Basically, it copies the disambiguator of its neighbours at the depth where they are equal, and, if none is equal, it copies its own site and counter. The latter case happens at least once per insertion which guarantees the uniqueness of each identifier. The space complexity of disambiguator is upper-bounded by the length of the new path. Furthermore, depending on the scenario, it can be drastically compressed.

Example 5: Let us go back to Fig. 1. In this tree, the peer p_1 inserts the character R between the two pairs $\langle[3], E\rangle$ and $\langle[4], T\rangle$. The resulting path is [3.1]. Now, we add the disambiguators $\delta_E = [\langle 2, 1\rangle]$ meaning that it is the first insertion of the peer p_2, and $\delta_T = [\langle 3, 1\rangle]$ meaning that it is the first insertion of the peer p_3. The resulting path linked to R is [3.1]. Since the first integer of the path of R is similar to the path of the character E, the algorithm copies the first part of δ_E. Then, since none of the adjacent pairs have a length of 2, the algorithm copies the values of p_1, i.e. $\langle 1, 1\rangle$. The resulting disambiguator is $[\langle 2, 1\rangle, \langle 1, 1\rangle]$. In the same manner, p_1 inserts a new character between E and R. The resulting path being $[3.0.X]$, the resulting disambiguator is $[\langle 2, 1\rangle, \langle 1, 2\rangle, \langle 1, 2\rangle]$.

The global total order $(\mathcal{I}, <_\mathcal{I})$ is a composition of the aforementioned sets $\mathcal{P}$ and $\mathcal{D}$, and their respective total orders $(\mathcal{P}, <_\mathcal{P})$ and $(\mathcal{D}, <_\mathcal{D})$.

The global total order is defined as: $\forall p, q \in \mathcal{I}, p < q \Leftrightarrow \exists i$ with $\forall_{k=0}^{i-1}, p.P(k) = q.P(k) \wedge p.D(k) = q.D(k)$ such that $\exists j = i+1$ with $p.P(j) < q.P(j) \vee (p.P(j) = q.P(j) \wedge p.D(j) < q.D(j))$.

3.1. Space complexity

In text editing, most of the editing behaviours can be empirically summarised as a composition of two basic editing behaviours. (i) The random editing behaviour where the author inserts new elements at what appears random positions in the sequence. For instance, this behaviour mostly arises when syntactic corrections are performed, e.g., the author writes $QWETY$ and realises that the R is missing. She adds the missing character in a second time. (ii) The monotonic editing behaviour where the author repeatedly inserts new elements between the last inserted element and the adjacent element. For instance, when an author writes $QWERTY$, she generally starts from the Q to the Y. On the opposite, insertions in a logfile are usually performed at the beginning for practical reasons.

As consequence, we focus the space complexity analysis of LSEQ to random and monotonic editing behaviours to which we add a worst-case complexity analysis. The analysis does not include an average case since it requires to know the average distribution of the position of edits performed by the humans which is obviously very complex.

Also, since the space complexity of disambiguators provided by *allocDis* is upper bounded by the paths allocated by *allocPath*, the space complexity analysis of the path is generalised to the space complexity of the identifiers of LSEQ.

THE RANDOM EDITING BEHAVIOUR is equivalent to a uniform distribution of the positions of the insert operations within the range of the sequence. As a consequence, the underlying tree of the allocation function is balanced. Depending on the depth k, the tree can hold a number n of elements:

$$n = \sum_{i=0}^{k} 2^{(i^2-i)/2} \tag{1}$$

Therefore, after n insertions performed on the sequence, the number of concatenations composing a path is upper-bounded by:

$$O(\sqrt{log\, n}) \tag{2}$$

Furthermore, a path is a sequence of numbers $[a_1.a_2 \dots a_k]$. Each number a_j requires one more bit than its preceding number a_{j-1}. Let b be the number of bits to encode a_0. Thus, a path is encoded by:

$$\sum_{i=1}^{k} b + i = kb + k(k+1)/2 = O(k^2)\, bits \tag{3}$$

By a simple replacement of k in the latter expression by the former, the resulting space complexity of identifiers is the optimal bound:

$$O(log\, n)\, bits \tag{4}$$

THE MONOTONIC EDITING BEHAVIOUR is similar to repeatedly: (i) fill one branch of the tree and (ii) increase the depth of the tree. Consequently, with a classical K-ary tree, the number of concatenations of the paths grows linearly. However, using an exponential tree, the number of available nodes in a branch still grows exponentially. Depending on the depth k, the tree can hold n elements:

$$n = 2^{k+1} - 1 \tag{5}$$

Therefore, the number of concatenations of the path is:

$$O(log\, n) \tag{6}$$

After the replacement of the variable k, we obtain the following space complexity of the monotonic editing behaviour:

$$O((log\, n)^2)\, bits \tag{7}$$

THE WORST-CASE corresponds to one element per level in the tree. Therefore, the growth of the paths is linear for both K-ary tree and exponential tree. Thus, when n insert operations are performed on the sequence, the number of concatenations composing a path is:

$$O(n) \tag{8}$$

However, while the space complexity remains linear in the case of the N-ary tree, the exponential tree implies the following space complexity on its identifiers:

$$O(n^2)\, bits \tag{9}$$

To summarise, the space complexity analysis reveals that the allocation function LSEQ sacrifices on the worst-case space complexity to improve the space complexity of the monotonic editing behaviours. Nevertheless, in text editing, the worst-case happens with a very low probability while the other analysed cases are frequent. Furthermore, it is worth noting that usually, the authors do not write a document in single round, starting from the beginning to go straight up to the end (as the monotonic analysis could suggest). On the contrary, the authors write sections, structure the documents, perform corrections, rewrite parts of the document, reorganise, etc. This behaviour (between random and monotonic behaviour) tends to even up the branches of the underlying tree of LSEQ.

	Random	Monotonic	Worst
Id.size	$O(\sqrt{log\, n})$	$O(log\, n)$	$O(n)$
Id.bit-length	$\sum_{i=1}^{k} b + i = kb + k(k+1)/2 = O(k^2)$		
Space complexity	$O(log\, n)$	$O((log\, n)^2)$	$O(n^2)$

Table 1: Spatial complexity of LSEQ. Where n is the number of insert operations performed. k is the size of an identifier, i.e., the number of concatenations. And b the starting bit-length of numbers composing the identifiers.

4. Experiments

This section is divided into two parts. (i) The first part describes the core of the distributed CollaboRATive Editor CRATE by filling in the blanks left on causality tracking and reliable broadcast. (ii) The second part concerns the experiments that have the following goals:

1. To validate the space complexity of LSEQ and highlight the improvement over the state-of-the-art variable-size identifiers CRDTs.
2. To show that CRATE scales in number of peers involved in the document editing at a same time.
3. To show that concurrency does not negatively impact the size of identifiers. Thus, scenarios without concurrency show the upper-bound on the size of identifiers.

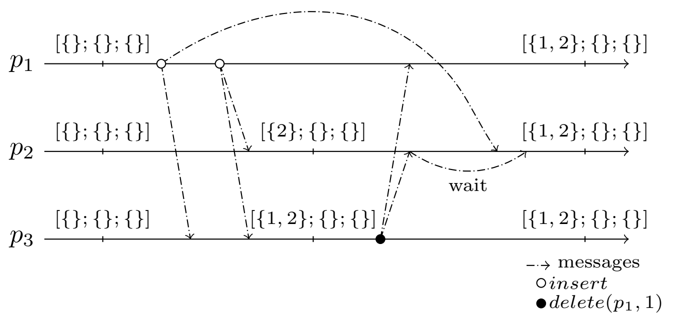

Figure 5: Timeline of operations performed by 3 peers. For the sake of simplicity, sets represent the intervals stored in the local vectors. The *insert* operations concern any element while the *delete* operation targets the first insertion of the peer p_1.

4.1. CRATE

CRATE (stands for CollaboRATive Editor) is a prototype of decentralised CRDT-based editor. CRATE uses a gossip dissemination protocol to scale in number of peers involved in the authoring. In this protocol, each peer has a set of neighbours and communicates with them. When a peer performs an operation, it creates a message containing the result of the operation. (1) Each peer broadcasts its messages. (2) When a peer receives a specific message for the first time, it broadcasts this message. However, Section 2 states that CRDTs provide strong eventual consistency upon the assumption of eventual delivery, and unfortunately, the gossiping broadcast is only reliable with high probability. To make it entirely reliable, CRATE includes an additional anti-entropy protocol to retrieve the possibly missing operations (c.f. Appendix B).

Section 2 also states that CRDTs for sequences provide commutative operations with the additional invariant that the deletion of a particular element cannot precede its insertion. To ensure this condition, CRATE uses a vector structure called interval version vector [22] (IVV).

Similarly to version vectors, IVVs require a vector with one entry per peer involved in the authoring. Contrarily to the former, each entry of an IVV contains a vector of intervals. Furthermore, when a peer broadcasts an operation, only two scalars are required to preserve the causal dependency instead of a full vector. This difference considerably lowers the network overload while keeping an eventual local space complexity at a same order of magnitude than version vectors. It constitutes an essential trade-off since local memory is often considered as virtually infinite while the network can be easily overloaded. CRATE uses an IVV to (i) minimise the retransmission of operations (i.e. each peer broadcast only once each operation), (ii) integrate the insert operation only once, (iii) preserve the aforementioned invariant. The delivery algorithm using the IVV can be found in Appendix B.

Example 6: Fig. 5 depicts an example of causality tracking using an IVV with all possible configurations. The 3 peers initialise their 3-entry vector with empty intervals. First, p_1 performs an insertion and broadcasts it to p_2 and p_3. The operation conveys the unique site identifier p_1 and its corresponding clock 1. When p_3 receives the message, it adds the entry to the corresponding interval, and immediately delivers the new element. The second operation of the peer p_1 conveys the clock 2. When p_2 receives this message, it merges the entry with its local vector and delivers the element, even if the first operation of p_1 is not received yet. The peer p_3 does the same resulting in the intervals $[\{1,2\};\{\};\{\}]$. Then, p_3, not satisfied by the first operation of p_1 deletes it. The message conveys these two scalars. When p_2 receives the message, the interval corresponding to the insertions at p_2 does not contain the first one. Consequently, the *del* operation must wait the delivery of the first insertion of p_1. On p_1, the *del* operation is immediately delivered.

4.2. Setup and results

Two kinds of environment hosted the experimentation. (1) A single machine emulating multiple peers. Despite the poor scalability of these experiments due to replication, it allows running the experiments in a controlled environment. (2) Multiple machines distributed in a cluster of the Grid'5000 testbed. A single machine can host multiple peers and communicate with other machines. The magnitude of these experiments grows up to 450 connected peers.

In these experiments, we focus our analysis on two editing behaviours: random and monotonic. In the random editing behaviour case, when a peer must insert an element, it randomises the position of the new element in the document following a uniform distribution. In the monotonic editing behaviour case, when a peer must insert an element, it inserts the new element at the end of the document. As stated in Section 3.1, this case is not favourable because it tends to unbalance the underlying tree representing the sequence. On the opposite, performing insertions at different locations tends to even up the branch of the tree, i.e., the average size of identifiers is lowered.

We focus our measurements on the average size of the messages broadcast by each peer. Since the delete messages have a low and constant size, we focus on messages corresponding to insert operations. Therefore, the measurements reflect the average size of the identifiers. The initial base parameter, i.e., the number of children at the root of the exponential tree, starts from a low value in order to accelerate the growth of the identifiers. In this section, we are more interested in the shape of the curves than the actual values.

In order to perform these evaluations, we developed a JavaScript data type for Web applications. The code is released on the Github platform[1] under the MIT license. Based on this implementation, we developed a prototype

[1]`https://github.com/Chat-Wane/LSEQTree.git`

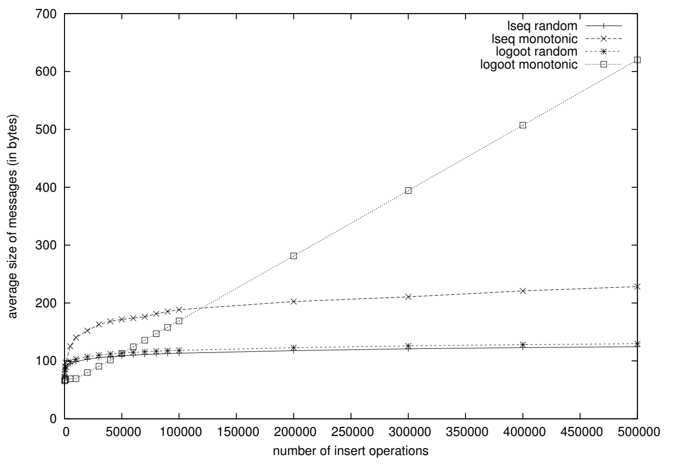

Figure 6: Average size of the messages exchanged between two peers during repeated insertions of elements in the sequence. The measurements concern two editing behaviours (monotonic and random) with two CRATE using different CRDTs for sequences (Logoot and LSEQ-based). The replicated document grows up to half a million characters.

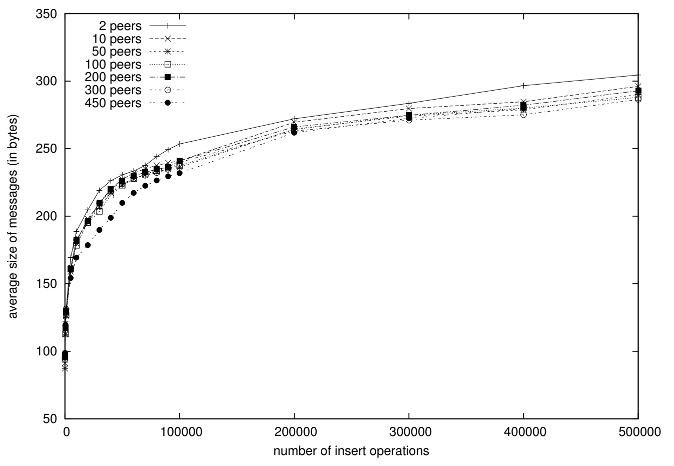

Figure 7: Average size of messages sent by peers during a session of document editing. The number of peers varies from 2 to 450 collaboratively creating a document of 500 thousand characters in 50 minutes. The editing behaviour is monotonic.

of distributed CollaboRATive Editor CRATE[2] following the outlines of this paper.

Objective: To validate the space complexity analysis of LSEQ. In particular, when the editing behaviour is monotonic, LSEQ has a polylogarithmic upper-bound on space complexity compared to the number of insert operations. When the editing behaviour is random, LSEQ has a logarithmic space complexity. A second objective is to compare LSEQ with an existing CRDT for sequences: Logoot [17, 23].

Description: A single machine hosts two peers communicating via a peer-to-peer connection. They aim to create a document of half a million characters in 50 minutes, i.e., they insert 166 characters per second. Two editing behaviours are studied: monotonic and random. They use whether CRATE, or a version of CRATE using Logoot instead of LSEQ. According to [7], Logoot constitutes the CRDT with variable-size identifiers that delivers the best performance overall. As such, it is an ideal baseline. While LSEQ starts with a departure base of 2^8 and doubles it when required, Logoot starts and stays with a base of 2^{16}. We measure the average size of messages transiting through the peers during the experiment.

Results: Fig. 6 shows the result of the experiments. The y-axis corresponds to the average size of messages transiting through peers in bytes. The x-axis corresponds to the size of the document, i.e., the number of insert operations performed on the sequence. The random editing behaviour leads to a logarithmic growth of the size of messages with both kinds of CRDT for sequences with a barely noticeable difference. On the other hand, the monotonic editing behaviour measurements show two different growths. Logoot exposes a linear growth of the size of its messages while LSEQ-based CRDT has a polylogarithmic shape proved in Section 3. Thus, even if Logoot starts with a smaller size messages, it quickly becomes less efficient than the LSEQ-based CRDT when the number of insertions increases.

Reasons: The random editing behaviour is slightly better for the LSEQ-based CRDT compared to Logoot because LSEQ starts with a lower departure base. Thus, Logoot is higher of a small constant degree. Nonetheless, the random editing behaviour leads to a same space complexity for both CRDTs. In the case of monotonic editing, the identifiers of Logoot linearly grow because Logoot uses a K-ary tree. Consequently, it does not adapt itself to the growing number of insertions in the sequence, i.e., each node in the tree can store as many nodes as its parent. On the opposite side, LSEQ doubles the number of available children when required. Consequently, when the document size increases, the number of possible identifiers grows even more.

Objective: To show that CRATE scales in terms of number of peers. In other terms, the size of the network does not modify the space complexity upper bound of messages.

Description: A cluster of the Grid'5000 testbed hosts a varying number of peers using the application CRATE. The peers aims to create a document of 500 thousand characters by repeatedly inserting at the end of the document (monotonic editing). The experiments last 50 minutes. The number of peers goes from 2 to 450. Overall, the generation of 166 operations per second are uniformly distributed among the peers and time. We measure the average size of messages sent by peers.

Results: Fig. 7 shows the results of these experiments.

[2]`https://github.com/Chat-Wane/CRATE.git`

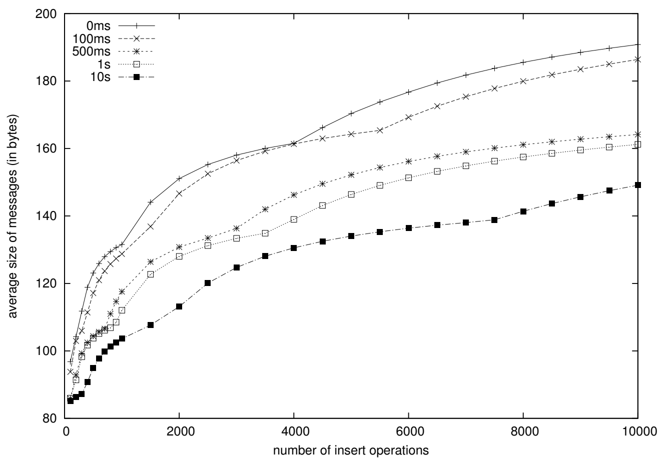

Figure 8: Average size of messages sent by peers during sessions of document editing. Ten peers create documents of 10 thousand characters by repeatedly inserting at the end of the document at a rate of 3 operations per second. Five editing sessions with different network latency are studied.

The y-axis corresponds to the average size of messages in bytes. The x-axis corresponds to the number of insert operations performed on the sequence. Fig. 7 shows that, independently of the number of peers involved, the average size of messages grows polylogarithmically compared to the size of the document. Also, even if the values of the 2-peers curve are invariably above the others, they do not demonstrate significant differences. Consequently, CRATE scales in two dimensions: number of peers and number of insertions.

Reasons: The messages size reflects the average size of the identifiers and the additional causality tracking metadata. Section 4.1 describes CRATE that requires only two scalars to track semantically related operations. Since these scalars also exist within the identifiers, we reuse them. Consequently, the average size of messages only depends of the average size of the identifiers. Since the space complexity of LSEQ is independent of the number of peers, so are the messages of the CRATE. Still, the latter locally requires a linearly growing vector in terms of number of peers. Fig. 7 shows small measurement variations between the different runs. This result is due to a growing number of peers that implies an increasing concurrency rate. When some peers perform operations concurrently at same position, they call the function *insert* with the same bounds as arguments. Therefore the resulting paths share a same allocating range. Thus, they are closer from each other compared to sequential execution. Therefore, the average size of messages slightly decreases when the number of peers increases.

Objective: To show that concurrency does not negatively impact the size of identifiers. Thus, scenarios without concurrency shows the upper-bound on the size of identifiers.

Description: A single machine emulates 10 peers using the application CRATE. The peers create a document of 10 thousand characters at a rate of 3 insertions per second uniformly distributed among the peers. The tool *netem* provides network emulation functionality. In particular, it allows us to change the delay of messages travelling through a specific network interface. We design five runs with the approximate following latency: $0.02ms$, $100ms$, $500ms$, $1s$, and $10s$. As usual, we measure the average size of messages transiting through the peers.

Results: Fig. 8 shows the results of measurements done with the different editing sessions. The y-axis corresponds to the average size of messages in bytes. The x-axis corresponds to the number of insert operations performed on the sequence. In all cases, the growth of the average size of messages follows the expectation from space complexity analysis in Section 3. Also, when the latency increases, the average size of messages decreases. Thus, the curve which corresponds to an almost sequential execution represents the heaviest messages in average.

Reasons: The explanation behind the decreasing in the average size of messages when the latency increases is similar to the one provided in previous experiment. However, instead of the number of peers, the latency impacts on the concurrency. Ultimately, when the latency is higher than the run time, each peer of the set of collaborator works alone and shares its operations afterwards. The resulting tree becomes populated with 10 local executions. Nevertheless, its depth does not grow. The bounces in the average message sizes are due to an addition of level in the underlying tree. Indeed, the average tends to a specific value depending on the depth in the tree. When the depth increases, the average size quickly grows to converge to the new value. A finer grain of measurements in this experiment allows us to observe these changes.

4.3. Synthesis

This section detailed a complete distributed collaborative editor allowing real-time editing. Using the outline, we developed CRATE and used it to perform the experiments.The first experiment confirmed the space complexity analysis of Section 3. Therefore, CRATE scales with respect to the number of insert operations performed on the document. The second experiment showed that using a causality tracking mechanism focused on semantically dependent operations does not impact the size of messages. As a consequence, CRATE scales with respect to the number of peers. Finally, the third experiment confirmed the observations made in the second one: the concurrency rate does not negatively impact the size of identifiers. Also, the sequential execution gives the upper bound on the average size of messages.

5. Related Work

Distributed collaborative editing tools can be divided in two approaches. (1) The Operational Transform (OT) approach [3] is the most ancient. A wide variety of OT approaches exist for text editing, image editing etc. In the context of text editing, OT can provide the usual *insert* and *delete* operations plus a range of string-wise operations such as *move*, $cut - paste$, etc. However, the correctness analysis requires to carefully examine each pair of operations and their parameters. As a consequence, few OT approaches are actually correct [24]. Furthermore, (i) centralised approaches [1] serialise the order of operations on a central server. Consequently, guaranteeing the convergence properties is simpler. However, the topology in itself implies single point of failure, privacy issues, economic intelligence issues, and scalability issues. (ii) Decentralised approaches [25] require a version vector to identify the generation context of the received operations and transform the arguments with the concurrent operations in the logfile. As consequence, OT approaches work fine in confined configurations only. However, they are not designed to handle massive authoring of large documents. In comparison, CRATE uses (a) LSEQ which does not require a logfile and whose complexity depends of the insert operations only, (b) an interval version vector which allows sending only two scalars in messages to guarantee the convergence of the CRDT, (c) a CRDT approach which supports concurrency. (2) The Conflict-free Replicated Data Type [5, 6] approaches share the computational cost between the local and remote part of the optimistic replication. While CRDTs significantly improve the time complexity compared to decentralised OT approaches, they hide the complexity in the memory usage.

CRDTs for sequences are split into two classes: (1) The tombstone class of CRDTs [7, 8, 9, 10, 11, 12, 13, 14, 15] marks the deleted elements. Therefore, while these elements do not appear to the user, they still exist in the underlying model and impact on performance. Thus, the complexities depend of the number of both operations insert and delete. Including the deletes in complexities is problematic because, for instance, in Wikipedia documents subject to vandalism, delete operations are very common. As consequence, the pages could contain few lines and yet use a lot of storage due to the hidden tombstones. (2) The variable-size identifiers class of CRDTs [11, 16, 17] assigns an identifier to each element in the document. Contrarily to tombstone approaches, deleted elements are truly removed. However, the identifiers are lists encoding the order of elements. As a consequence, the space complexity of these approaches is crucial and depends of the number of insert operations only.

In the literature, there are two kinds of allocation functions for the identifiers. Let us consider an insertion of b between the id_a and id_c where $id_a < id_c$. Both the allocation functions aim to provide the shortest identifiers possible. Thus, the maximum size of the identifier id_b corresponding to b is always: $min(|id_a|, |id_c|) + 1$. (1) The first allocation function consists in randomising a path between id_a and id_c. This function aims to make the conflicts very unlikely. However, with a monotonic editing behaviour, such function consumes on average half the level of identifiers in a branch. (2) The observations made on a Wikipedia corpus led to a function designed for end-editing. When the editing behaviour complies with this assumption, the average size of identifiers remains linear. Employing the opposite editing behaviour, the allocation function becomes unreliable.

The allocation function LSEQ [18, 19] improves state-of-the-art allocation functions by lowering the space complexity of identifiers from linear to sub-linear. Fig. 6 shows how it impacts the average size of identifiers. As consequence, LSEQ scales compared to the size of the document.

LogootSplit [16] proposes an orthogonal solution to reduce the metadata implied by variable-size CRDTs for sequences. LogootSplit does not reduce the space complexity of the identifiers but modifies the way identifiers are linked to elements. Indeed, LogootSplit handles multiple granularities. LogootSplit aims to allocate identifiers to coarse grain elements whenever it is possible in order to reduce the number of identifiers. Being orthogonal, LogootSplit and LSEQ could be used together to provide the improvements of both approaches.

Using LSEQ, a CRDT for sequences still requires a form of causality tracking. However, causality tracking is still an issue in distributed network subject to churn [26]. The full causality tracking requires piggybacking a version vector with N entries within each message, N being the number of peers *ever* involved in the network. Of course, it is impracticable in contexts where peers can join and leave the network freely. Some other approaches [27] reuse entries of left peers but require that leaving peers notifies it clearly. Despite the fact that version vectors are the smallest structures to accurately characterise causality [28], there exist other trade-offs between space complexity and accuracy. CRATE chooses to track the semantically related operations accurately. The interval version vector [22] has a local space complexity eventually comparable to the space complexity of the version vector while not overwhelming the network with causality metadata.

6. Conclusion

In this paper, we refined the allocation function LSEQ by providing the proof of its space complexity. It has a logarithmic, polylogarithmic, and quadratic upper bound on its space complexity for random insertions, monotonic insertions, and worst-case respectively. Using this allocation function and the interval version vector to track causality, we developed a distributed collaborative editor for real-time called CRATE. This editor is fully decentralised and scales in terms of: number of peers, concurrent operations, and document size. We validated the approach on a setup close of the real life conditions and using extreme parameters: low startup values, high number of insertions, high number of peers, high latency. The experimentation confirmed the space complexity of LSEQ. It also confirmed our expectation on the scalability of CRATE. As such, it alleviates the issues brought by centralised approaches: single point of failure, privacy issues, economic intelligence issues, limitations in terms of service, etc. Also, it alleviates the scalability issues of decentralised approaches. As such, it may become a serious competitor for current trending editors (such as Google Docs, SubEthaEdit, . . .), allowing massive real-time collaborative editing without any service providers. More specifically, it opens the field to a new range of distributed applications such as massive editing of online courses, or collaborative reviewing of co-located events, webinars etc.

Future work concerns the causality tracking issue. Indeed, the chosen trade-off proposed by the interval version vectors allows scaling in number of peers. Nevertheless, when the network is subject to churn, the local memory used to store the vector can grow quickly. We envision an approach trading accuracy for memory without sacrificing on correctness.

Acknowledgement

This work was partially funded by the French ANR project SocioPlug (ANR-13-INFR-0003), and by the DeSceNt project granted by the Labex CominLabs excellence laboratory (ANR-10-LABX-07-01).

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see `https://www.grid5000.fr`).

References

[1] D. A. Nichols, P. Curtis, M. Dixon, J. Lamping, High-latency, low-bandwidth windowing in the jupiter collaboration system, in: Proceedings of the 8th annual ACM symposium on User interface and software technology, ACM, 1995, pp. 111–120.

[2] P. R. Johnson, R. H. Thomas, Maintenance of duplicate databases, RFC 677 (Jan. 1975).
URL http://www.ietf.org/rfc/rfc677.txt

[3] Y. Saito, M. Shapiro, Optimistic replication, ACM Comput. Surv. 37 (1) (2005) 42–81. doi:10.1145/1057977.1057980.
URL http://doi.acm.org/10.1145/1057977.1057980

[4] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, D. Terry, Epidemic algorithms for replicated database maintenance, in: Proceedings of the sixth annual ACM Symposium on Principles of distributed computing, ACM, 1987, pp. 1–12.

[5] M. Shapiro, N. Preguiça, C. Baquero, M. Zawirski, A comprehensive study of Convergent and Commutative Replicated Data Types, Rapport de recherche RR-7506, INRIA (Jan. 2011).
URL http://hal.inria.fr/inria-00555588

[6] M. Shapiro, N. Preguiça, C. Baquero, M. Zawirski, Conflict-free replicated data types, Stabilization, Safety, and Security of Distributed Systems (2011) 386–400.

[7] M. Ahmed-Nacer, C.-L. Ignat, G. Oster, H.-G. Roh, P. Urso, Evaluating CRDTs for Real-time Document Editing, in: ACM (Ed.), 11th ACM Symposium on Document Engineering, Mountain View, California, États-Unis, 2011, pp. 103–112. doi:10.1145/2034691.2034717.
URL http://hal.inria.fr/inria-00629503

[8] N. Conway, Language support for loosely consistent distributed programming, Ph.D. thesis, University of California, Berkeley (2014).

[9] V. Grishchenko, Deep hypertext with embedded revision control implemented in regular expressions, in: Proceedings of the 6th International Symposium on Wikis and Open Collaboration, WikiSym '10, ACM, New York, NY, USA, 2010, pp. 3:1–3:10. doi:10.1145/1832772.1832777.
URL http://doi.acm.org/10.1145/1832772.1832777

[10] G. Oster, P. Urso, P. Molli, A. Imine, Data consistency for p2p collaborative editing, in: Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work, ACM, 2006, pp. 259–268.

[11] N. Preguiça, J. M. Marqus, M. Shapiro, M. Letia, A commutative replicated data type for cooperative editing, in: Distributed Computing Systems, 2009. ICDCS'09. 29th IEEE International Conference on, Ieee, 2009, pp. 395–403.

[12] H.-G. Roh, M. Jeon, J.-S. Kim, J. Lee, Replicated abstract data types: Building blocks for collaborative applications, Journal of Parallel and Distributed Computing 71 (3) (2011) 354–368.

[13] S. Weiss, P. Urso, P. Molli, Wooki: A p2p wiki-based collaborative writing tool, in: B. Benatallah, F. Casati, D. Georgakopoulos, C. Bartolini, W. Sadiq, C. Godart (Eds.), Web Information Systems Engineering WISE 2007, Vol. 4831 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2007, pp. 503–512. doi:10.1007/978-3-540-76993-4_42.
URL http://dx.doi.org/10.1007/978-3-540-76993-4_42

[14] Q. Wu, C. Pu, J. Ferreira, A partial persistent data structure to support consistency in real-time collaborative editing, in: Data Engineering (ICDE), 2010 IEEE 26th International Conference on, 2010, pp. 776–779. doi:10.1109/ICDE.2010.5447883.

[15] W. Yu, A string-wise crdt for group editing, in: Proceedings of the 17th ACM international conference on Supporting group work, GROUP '12, ACM, New York, NY, USA, 2012, pp. 141–144. doi:10.1145/2389176.2389198.
URL http://doi.acm.org/10.1145/2389176.2389198

[16] L. André, S. Martin, G. Oster, C.-L. Ignat, Supporting Adaptable Granularity of Changes for Massive-scale Collaborative Editing, in: CollaborateCom - 9th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing - 2013, Austin, États-Unis, 2013.
URL http://hal.inria.fr/hal-00903813

[17] S. Weiss, P. Urso, P. Molli, Logoot: a scalable optimistic replication algorithm for collaborative editing on p2p networks, in: Distributed Computing Systems, 2009. ICDCS'09. 29th IEEE International Conference on, IEEE, 2009, pp. 404–412.

[18] B. Nédelec, P. Molli, A. Mostfaoui, E. Desmontils, LSEQ: an Adaptive Structure for Sequences in Distributed Collaborative Editing, in: ACM (Ed.), 13th ACM Symposium on Document Engineering, 2013. doi:10.1145/2494266.2494278.
URL http://doi.acm.org/10.1145/2494266.2494278

[19] B. Nédelec, P. Molli, A. Mostéfaoui, E. Desmontils, Concurrency effects over variable-size identifiers in distributed collaborative editing, in: DChanges, Vol. 1008 of CEUR Workshop Proceedings, CEUR-WS.org, 2013.

[20] A. Andersson, Faster deterministic sorting and searching in linear space, in: Foundations of Computer Science, 1996. Proceedings., 37th Annual Symposium on, IEEE, 1996, pp. 135–141.

[21] A. Andersson, M. Thorup, Dynamic ordered sets with exponential search trees, J. ACM 54 (3). doi:10.1145/1236457.1236460.
URL http://doi.acm.org/10.1145/1236457.1236460

[22] M. Mukund, G. Shenoy R., S. Suresh, Optimized or-sets without ordering constraints, in: M. Chatterjee, J.-n. Cao, K. Kothapalli, S. Rajsbaum (Eds.), Distributed Computing and Networking, Vol. 8314 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2014, pp. 227–241. doi:10.1007/978-3-642-45249-9_15.
URL http://dx.doi.org/10.1007/978-3-642-45249-9_15

[23] S. Weiss, P. Urso, P. Molli, Logoot-undo: Distributed collaborative editing system on p2p networks, IEEE Trans. Parallel

Distrib. Syst. 21 (8) (2010) 1162–1174.

[24] A. Imine, P. Molli, G. Oster, M. Rusinowitch, Proving correctness of transformation functions in real-time groupware, in: Proceedings of the Eighth Conference on European Conference on Computer Supported Cooperative Work, ECSCW'03, Kluwer Academic Publishers, Norwell, MA, USA, 2003, pp. 277–293.
URL http://dl.acm.org/citation.cfm?id=1241889.1241904

[25] D. Sun, C. Sun, Context-based operational transformation in distributed collaborative editing systems, IEEE Transactions on Parallel and Distributed Systems 20 (10) (2009) 1454–1470. doi:10.1109/TPDS.2008.240.

[26] R. Baldoni, M. Raynal, Fundamentals of distributed computing: A practical tour of vector clock systems, IEEE Distributed Systems Online 3 (2) (2002) 12.

[27] P. S. Almeida, C. Baquero, V. Fonte, Interval tree clocks, in: Proceedings of the 12th International Conference on Principles of Distributed Systems, OPODIS '08, Springer-Verlag, Berlin, Heidelberg, 2008, pp. 259–274. doi:10.1007/978-3-540-92221-6_18.
URL http://dx.doi.org/10.1007/978-3-540-92221-6_18

[28] B. Charron-Bost, Concerning the size of logical clocks in distributed systems, Inf. Process. Lett. 39 (1) (1991) 11–16. doi:10.1016/0020-0190(91)90055-M.
URL http://dx.doi.org/10.1016/0020-0190(91)90055-M

Brice Nédelec received the MS degree in Software Architecture from the University of Nantes (France) in 2012. He is currently a Ph.D. student at the University of Nantes as a team member of GDD. His main research interests concern distributed applications, collaborative editing, causality tracking, and eventual consistency.

Pascal Molli graduated from Nancy University (France) and received his Ph.D. in Computer Science from Nancy University in 1996. Since 1997, he is Associate Professor at University of Nancy. His research topic is mainly Computer Supported Cooperative Work, P2P and distributed systems and collaborative knowledge building. His current research topics are: Algorithms for distributed collaborative systems, privacy and security in distributed collaborative systems, and collaborative distributed systems for the Semantic Web.

Achour Mostefaoui received his M.Sc. in computer science in 1991, and a Ph.D. in computer science in 1994 both from the University of Rennes. He has been Associate professor in the Computer Science of the University of Rennes from 1996 to 2011 when he joined the University of Nantes as full professor. He spent one semester in the Theory of Computing group of the CSAIL Lab. in the Massachusetts Institute of Technology in 2007. Since September 2011, he is head of a Master's diploma in Computer Science (University of Nantes) and is co-head of the GDD research team within the LINA Lab. His research interests are on distributed computing. A first direction concerns agreement and calculability issues in asynchronous, fault-prone distributed systems. More specifically, he is interested in discovering the boundary between solvable and unsolvable tasks and, understanding further assumptions/relaxations needed to solve otherwise impossible tasks. A second direction is to study replicated date structure (queue, tuple space, transactionnal memory, wiki, etc.). How to specify and implement distributed data structures and then integrate them into programming languages in different distributed computing models. For both directions, distributed algorithms design is the key point in my work, either to study algorithmic mechanisms, to efficiently solve tasks, or to show impossibility results through reductions.

Appendix A. Abstract problem

To highlight the problem, we withdraw the unnecessary aspects inherent to CRDTs and distributed collaborative editing. The "From Mutable to Immutable" simply depicts the problem as a incremental translation from a set where the indices are mutable, (i.e., an element is linked to a position that may change) to a set where indices are immutable.

[From Mutable to Immutable problem] Let X the goal sequence composed of elements from an alphabet $\mathcal{A}$, Y the sequence resulting of the permutation of X and containing additional elements from a set $\mathcal{M}$ equipped with a total order $(\mathcal{M}, <_{\mathcal{M}})$, Z the set containing the elements of X and immutable elements from a set $\mathcal{I}$ equipped with a dense total order $(\mathcal{I}, <_{\mathcal{I}})$.

Given:

- $X : \mathbb{N} \to \mathcal{A}$
- $Y : \mathbb{N} \to \mathcal{A} \times \mathcal{M}$
- $Z : \mathbb{N} \to \mathcal{A} \times \mathcal{I}$
- $gen\beta : \mathcal{M} \to \mathcal{I}$
- Z-uniqueness:
  $\forall \langle \alpha_1, \beta_1 \rangle, \langle \alpha_2, \beta_2 \rangle \in Z, \beta_1 = \beta_2 \Rightarrow \alpha_1 = \alpha_2$
- Z-order-preservation: $Z(i) = \langle \alpha_i, \beta_i \rangle \Rightarrow X(i) = \alpha_i$
- Z-specification: $(Z : Y \times \mathbb{N} \to Z)$:
  - $Z(\varnothing, \_) \to \varnothing$
  - $Z(\_, 0) \to \varnothing$
  - $Z(Y, i) \to$ Let $Y(i) = \langle \alpha_i, \mu_i \rangle$: $Z(Y, i-1) \cup \{\alpha_i, gen\beta(\mu_i)\}$
  - $Z(Y, |Y|)$

The problem is to find an optimal function $gen\beta$ such that there do not exist any functions $gen\beta'$ such that for any permutation Y, the resulting set Z' using $gen\beta'$ has a lower binary representation than the resulting set Z using $gen\beta$.

Put back in the distributed collaborative editing context, the goal sequence X corresponds to the intention of authors, i.e., the final document that peers will eventually write (e.g., $QWERTY$ in the paper). The sequence Y is an editing sequence performed by the authors to reach that goal (e.g., $[(Q,\,0),\,(W,\,1),\,\ldots]$). However, using such indices to order elements can lead to divergent replicas depending on the integration order. For instance, let us consider two peers concurrently inserting $(Q,\,0)$ and $(W,\,0)$. When they receive the operation from each other, the first peer shifts the character Q while the other shifts W. They ends up with WQ and QW respectively. To solve this problem, the set Z uses a function $gen\beta$ to transform the mutable indices from $\mathcal{M}$ to immutable indices from $\mathcal{I}$. Using the dense total order $(\mathcal{I},\,<_{\mathcal{I}})$, we are able to retrieve the goal sequence. For instance, after the concurrent insertions of $(Q,\,0)$ and $(W,\,0)$, the set Z transforms the shifting indices to i_1 and i_2 from $\mathcal{I}$. The execution order of operations does not matter since the elements are identically ordered on both replicas. The peers end up with either QW or WQ. Trees are able to represent the dense total order $(\mathcal{I},\,<_{\mathcal{I}})$. In this regard, and taking into account the concurrency, the composition of *allocPath* and *allocDis* corresponds to $gen\beta$.

Finding an optimal function $gen\beta$ for any permutation Y is impossible. Indeed, there always exists an allocation function more suitable for a particular case. Nonetheless, focusing on the random and the monotonic editing behaviours provides a first answer restrained to text editing.

Appendix B. Message delivery protocol

This section provides the additional algorithms of the messaging between peers. The distributed collaborative editor CRATE uses these protocols to ensure that operations are eventually integrated, only once, and the deletion of an element is always integrated after its insertion.

Algorithm 4 describes the delivery protocol running at each peer. It highlights the optimistic replication scheme of CRDTs by dividing the operations between the local update and the received update. Since different implementations are possible, the functions relative to the interval structure are left aside. Semantically, the $r[i].add$ function adds the value in parameters to the vector of intervals. The $r[i].last$ function gets the highest value of the vector of intervals. The function *alloc* is an aggregation of *allocPath* and *allocDis*. The function *unique* extracts the unique site identifier and counter from a triple.

Line 20 prevents the application from multiple receptions. Without any causality tracking mechanism, a CRDT for sequences is not able to provide the idempotency property of CRDTs. Without idempotency, the replicas may diverge. Indeed, the remote part of the delete operations removes information from the data structure. When the causality tracking implicitly keeps the summary of all operations, it implies that a non-existing element cannot be mistaken for an element inserted then deleted.

Algorithm 4 Delivery protocol

```
1:  let $r$ the old ivv on peer $p_i$
2:  let $s$ the new ivv on peer $p_i$
3:  let $m$ the incoming message from peer $p_j$
4:  let $\langle type,\, args\rangle$ the specification of $m$

5:  INITIALLY:
6:      for $k$ from $0$ to $|r|-1$ do $s[k] \leftarrow \varnothing$;
7:
8:  LOCAL UPDATE:
9:      for $k$ from $0$ to $|r|-1$ do $s[k] \leftarrow r[k]$;
10:     on insert (args):
11:         $s[i] \leftarrow r[i].add(r[i].last()+1)$;
12:         $broadcast('insert',\, alloc(args))$;
13:     on delete (args):
14:         $broadcast('delete',\, args)$;
15:
16: RECEIVED UPDATE:
17:     for $k$ from $0$ to $|r|-1$ do $s[k] \leftarrow r[k]$;
18:     on insert (args):
19:         let $\langle src,\, cnt\rangle \leftarrow unique(args)$;
20:         if $(\neg(r[src].contains(cnt)))$ then
21:             $s[src] \leftarrow r[src].add(cnt)$;
22:             $deliver(m)$;
23:         end if
24:     on delete (args):
25:         let $\langle src,\, cnt\rangle \leftarrow unique(args)$;
26:         if $(r[src].contains(cnt))))$
27:             then $deliver(m)$;
28:             else $delay(m)$;
29:
```

Example 7: The following example depicts the need to integrate the operation only once in presence of potential duplication of messages. A first peer generates two operations with the unique element e ($insert(e) \rightarrow delete(e)$). When the $insert(e)$ operation arrives to a second peer, it is delivered. Similarly, when the $delete(e)$ operation arrives and sees the element e in the replicated structure, it is delivered, leading to the removal of the element e. However, somehow, a copy of the *insert* message arrives to the second peer. Since the element e does not exist in the structure any more, this peer will assume that it receives this operation for the first time and will apply it. Since the first peer does not necessarily receive the copy of the *insert* message, or receives the copy before performing the $delete(e)$ operation, the replicas of the two peers may become divergent.

As stated in Section 2, CRDTs require a reliable broadcast to provide strong eventual consistency. Nevertheless, CRATE uses a gossip dissemination protocol which scales in number of peers, yet unreliable.

Algorithm 5 depicts the anti-entropy event added to

Algorithm 5 Anti-entropy protocol

```
1: RECEIVED UPDATE:
2:     on antiEntropy ($args$):
3:         let $ivv \leftarrow args$;
4:         let $deltas \leftarrow diff(r, ivv)$;
5:         for ($\delta$ in $deltas$) do $sendTo(p_j, retrieve(\delta))$;
6:
```

the main delivery algorithm. This algorithm guarantees that all the operations will be eventually delivered to all peers. A peer p_j unevenly sends a message containing its interval version vector to another peer p_i triggering the event *antiEntropy*. The function $diff$ calculates the differences between the interval version vectors (from p_i and p_j). The function *retrieve* searches for each spotted difference and sends it back to p_j. Being costly, this protocol starts rarely.