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 has renewed the interest for efficient data structures that can allow a huge number of simultaneous users. Indeed, existing editors lack scalability in terms of the number of users, concurrency, and the size of the produced documents. Additionally, centralised editors suffer from privacy issues, single-point-of-failure, economic intelligence issues, and restrictions in terms of service.

Several replicated data structures implementing sequences have been proposed to alleviate most of the aforementioned issues. Yet, they have an unsatisfying space complexity. Either they use tombstones and then deleted elements are only hidden to the user or they allocate to each element of the sequence an identifier that may be as large as the number of elements in the sequence. This paper proposes an identifier allocation function named LSEQ. It provides identifiers enjoying a sub-linear space complexity without being application specific and proves the polylogarithmic size of these identifiers. Moreover, this paper provides all the outlines to develop distributed collaborative editors that scale in all the aforementioned dimensions. Using these outlines, we developed a prototype called CRATE (CollaboRATive Editor) that we evaluated on the Grid'5000 testbed. The results show that such editors, being decentralised and scalable, could constitute a serious competitor to current trending editors. 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 in distributed collaborative editors such as Google Docs, SubEthaEdit, or Etherpad, has not stopped growing. Such editors allow distributing the work across space, time and organisations. However, despite being undoubtedly useful, they have several limitations due either to centralisation (e.g. single-point-of-failure, privacy issues, economic intelligence issues, limitations in terms of service) or to decentralisation (e.g. number of users, concurrency and synchronisation, number of operations). For instance Google [1] allows only 50 users to write a document together at a same time. These scalability issues preclude the existence of a massive distributed collaborative editor.

Decentralised distributed collaborative editors use the optimistic replication scheme [2, 3] to provide high availability and responsiveness. Locally, each user involved in the collaboration owns a copy of the document (replica). Local operations are executed and become effective locally before being broadcast. Remotely, each user integrates the received operations 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].

Operational transformation (OT) approaches are the first techniques used 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 non practical because of unpredictable latency.

On the other side, a family of distributed data structure called Conflict-free Replicated Data Type [5, 6] (CRDT) has been introduced. This family encompasses data structures such as counters and variants of sequences, sets, and graphs. This family does not include data structures such as a register, a stack, etc. Fortunately, a sequence is a data structure well-suited for building distributed collaborative editors that scale in presence of concurrent operations. Contrarily to OT, most of the processing time is located in the local part of operations, making this approach more practical since each operation leads to N remote integrations (N being the number of collaborators). To ensure convergence, a unique and immutable identifier is associated with each element (e.g. character, words, line, paragraph) in the sequence (i.e. document). The set of identifiers is paired with a total order, and this latter actually makes the sequence.

Nevertheless, these sequence data structures have a space complexity issue. Indeed, they provide two update

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

operations (insert and delete) and use an identifier allocation function that either depends on the type of the performed operation [7, 8, 9, 10, 11, 12, 13, 14, 15], or the arguments of the operation [11, 16, 17]. In both cases, the space overhead induced by the identifiers can be prohibitively high and directly impacts performance. Unfortunately, this allocation is a crucial but a non-trivial problem.

In this paper, we study sequence data structures [1] at the core of which there is the identifier allocation function that provides the different collaborators with identifiers they assign to the elements of the associated sequence. This paper has four main contributions:

- A new identifier allocation function called LSEQ.
- A proof of the polylogarithmic space complexity of LSEQ.
- All the outlines to develop a distributed collaborative editor for massive editing of large documents and a prototype called CRATE (for CollaboRATive Editor).
- A experimental study that validates the space complexity of LSEQ and the scalability of CRATE. 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.

The remainder of this paper is organised as follows. Section 2 provides the necessary background to sequence data structures and the motivations. Section 3 follows with a description of LSEQ and the proof of its space complexity. Section 4 highlights the scalability of CRATE while validating the space complexity analysis of LSEQ. Section 5 reviews related works. Finally, Section 6 concludes the paper.

2. Preliminaries

Sequence data structures considered in this paper (for collaborative editing) belong to the optimistic class of replication. Each replica owner (a collaborator) directly performs operations on its local copy and informs the other owners by broadcasting these operations. The consistency criteria ensured by such optimistic replication is called 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. This consistency criterion is weaker than sequential consistency and linearisability and is not comparable with causal consistency, yet strong enough for collaborative edition [5].

Sequence replicated data structures (*sequence* for short) are the closest structures that can implement a shared document. A sequence provides two *commutative operations*: insert and delete. Indeed, these operations can be integrated in any order as soon as the deletion of an element is integrated after its insertion.

The sequence is composed of elements and a unique and immutable identifier is associated with each of these elements. The sequence can be seen as a set of pairs $\langle element, identifier\rangle$. A total order is assumed on identifiers and this order relation allows seeing the set of pairs as a sequence. The projection of a sequence on the elements builds the document.

When a collaborator performs an insert operation, it first allocates the identifier of the element to insert. For instance, let us consider a sequence $QWTY$ with the unique, immutable, and totally ordered integer identifiers 1, 2, 4, 8 respectively. A collaborator inserts the element E between W and T. The natural identifier that comes to mind is 3. The resulting sequence is $QWETY$. However, R cannot be inserted between E and T since 3 and 4 are contiguous. The space of identifiers must be enlarged to handle the new insertion. If we consider identifiers as decimal numbers, 3.1 can be associated with the character R. If a new character has to be inserted 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 non null 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 to keep the growth of the size of the identifiers under acceptable boundaries.

2.1. Variable-size Identifiers

Variable-size identifiers can be represented as a concatenation of basic elements (e.g. integers). The resulting sequence can be represented by a tree structure where the elements of the sequence are stored at the nodes and where the edges of the tree are labelled such that a path from the root to a node represents the identifier of the element stored at this node. For instance, the character R in the previous example is accessible following the path composed of the edges labelled 3 then 1. More formally, a sequence is a tree $\mathcal{T}$ where each node contains a value i.e. an element of the sequence (over an alphabet $\mathcal{A}$). The tree $\mathcal{T}$ is a set of pairs $\langle \mathcal{P} \subset \{\mathbb{N}\}^*, \mathcal{A}\rangle$, i.e., each element has a path. Additionally, a total order $(\mathcal{P},<_{\mathcal{P}})$ provides an ordering among the paths which allows to retrieve the order of the elements in the sequence.
Notation A path composed of p edges labelled $\ell_1, \ldots, \ell_p$ will be noted $[\ell_1.\ldots.\ell_p]$.

Figure 1 shows the underlying 10-ary tree $\mathcal{T}$ representing a sequences. Like in the previous scenario, the initial sequence is $QWTY$ with the respective paths [1], [2], [4] and [8]. The insertion of the character E between the pairs $\langle[2], W\rangle$ and $\langle[4], T\rangle$ results in the following pair: $\langle[3], E\rangle$. Then the insertion of the character R needs to start a new level since there is no room at the first level of the tree for a new path between E and T. The resulting path may be [3.1] if label one is chosen for the element R at the second

[1]A short version of this paper appeared in [18, 19].

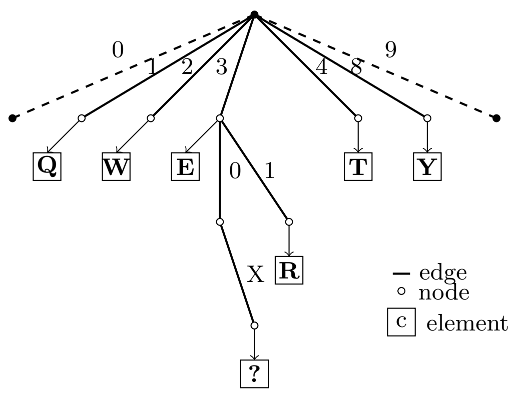

Figure 1: Underlying 10-ary tree $\mathcal{T}$ of a sequence. The paths $\mathcal{P}$ correspond to the concatenation of the edges from the root to the elements. The elements are characters. Using the total order ($\mathcal{P}$, $<_{\mathcal{P}}$), the sequence of elements is $QWERTY$.

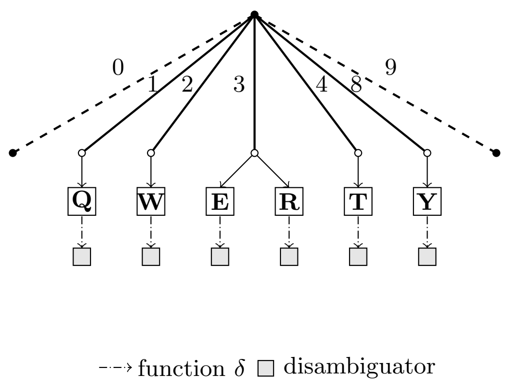

Figure 2: The 10-ary tree of a sequence including concurrent insertions. Using only ($\mathcal{P}$, $<_{\mathcal{P}}$) could lead to either $QWERTY$ or $QWRETY$. The disambiguators forces the final result to converge to $QWERTY$.

level. This would increase the depth of the tree in case there is an insertion between the elements E and R. The new path would be [3.0.X] where $0 < X < 10$ (recall that we assumed a 10-ary tree). The total order ($\mathcal{P}$, $<_{\mathcal{P}}$) allows retrieving the sequence $QWERTY$.

2.2. Disambiguation of concurrent cases

Two collaborators concurrently performing an operation on their respective replica may get different results after the integration of both operations. Indeed, ($\mathcal{P}$, $<_{\mathcal{P}}$) is a total order when a single collaborator edits. However, it becomes a partial order when the editing involves several collaborators. Consequently, it is necessary to totally order the elements inserted by different collaborators. To this end, the disambiguation function δ associates a disambiguator to each pair of $\mathcal{T}$. $\delta : \mathcal{P} \times \mathcal{A} \to \mathcal{D}$ such that ($\mathcal{D}$, $<_{\mathcal{D}}$) is a total order. The function δ is an accessor to additional values that are totally ordered even in presence of concurrency. Finally, the pairs in $\mathcal{T}$ can be totally ordered with a composition of the local total order ($\mathcal{P}$, $<_{\mathcal{P}}$) and the total order between collaborators ($\mathcal{D}$, $<_{\mathcal{D}}$). The composition leads to a total order ($\mathcal{T}$, $<_{\mathcal{T}}$).

Figure 2 depicts a tree containing 6 elements with only 5 distinct paths (two values are associated with the path [3]). Similarly to the previous example, the initial sequence was $QWTY$, however, in this example, the two elements E and R are inserted between the pairs $\langle[2], W\rangle$ and $\langle[4], T\rangle$ by two different collaborators concurrently. For both elements, the resulting path is [3]. After the two elements are inserted, the sequence becomes either $QWERTY$ or $QWRETY$. Nevertheless, let $\delta(\langle[3], R\rangle) = \delta_R$ and $\delta(\langle[3], T\rangle) = \delta_T$. If we assume $\delta_R <_{\mathcal{D}} \delta_T$, the total order ($\mathcal{T}, <_{\mathcal{T}}$) gives the sequence $QWERTY$. It is worth noting that disambiguators are usually computed using a monotonically increasing variable and a unique collaborator identifier just like Lamport timestamps [**?**]. Therefore, a collaborator cannot directly influence the final position of the character in the sequence using disambiguators. The sequence of the example could have ended in $QWRETY$ and it would have needed a correction.

2.3. Choosing the rightful path

The most critical part of sequences with variable-size identifiers consists in creating the paths. Algorithm 1 shows the general outlines of these sequences. It divides the operations (insert and delete) into to different parts, the local and the remote phases of the optimistic replication paradigm. As we can see, most of the core of the algorithm and the associated complexity represents the local part of the *insert* operation where the algorithm has to generate a path (cf. Line 6) and a disambiguator (cf. Line 7).

Algorithm 1 General outlines of a sequence with variable-size identifiers.

```
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 $\mathcal{I} : \mathcal{P} \times \mathcal{A} \times \mathcal{D}$. The set of the elements of a sequence is a subset of $\mathcal{I}$. For any element i of a sequence ($i \in \mathcal{I}$), let $i.P$, $i.A$, $i.D$ be the respective accessors to the path, the element, and the disambiguator of element i. The function *allocPath*

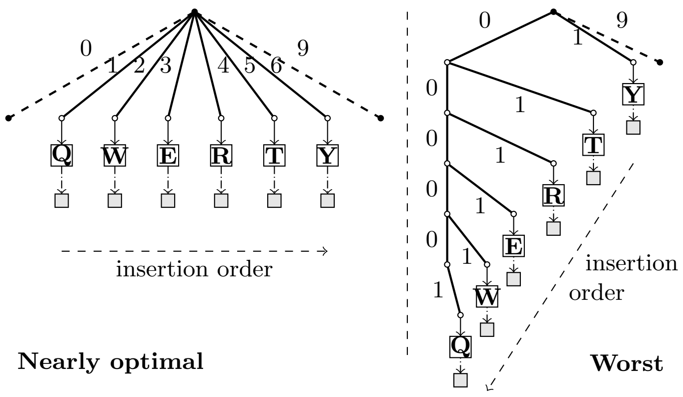

Figure 3: Two trees filled with the resulting identifiers of two different permutations resulting in an identical sequence $QWERTY$. They use the same function *allocPath* which allocates the leftmost branch in the tree. All paths of the nearly optimal case have a length of 1 while the tree of the worst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

chooses a path in the tree between two other paths $p.P$ and $q.P$ where $p <_{\mathcal{T}} q$. This means that element i has to be inserted between two elements p and q such that p precedes q in the sequence. However, since it is always create new levels in the tree and thus new sub-trees, the number of possible paths is infinite, and so is the number of *allocPath* functions. Nevertheless, the function *allocPath* should choose among the all the possible paths the one having the smallest length in order to have smaller identifiers keeping good performance. This observation reduces considerably the number of possible *allocPath* functions. Still, the allocation of paths without an *a priori* knowledge of the final sequence is a non-trivial problem (Appendix A depicts the problem abstracted from the collaborative editing context).

Figure 3 illustrates the difficulties of designing a function to allocate the paths. It represents the underlying trees of two sequences using the allocation function: they allocate the leftmost branch available at the lowest depth possible. In both cases the final sequence is $QWERTY$, however, the letters are not inserted in the same order. In the first case, Q is inserted first at position 0 (initially the sequence is empty), followed by W at position 1 (after Q) then E is inserted at position 3 (after W), etc. We call the sequence of insert operations $[(Q, 0), (W, 1), (E, 2), \ldots]$ the *editing sequence*. In the second case, the letter Y is inserted first at position 0 as the sequence is initially empty. Then T is inserted. However as the final intended word is $QWERTY$, T has to be inserted at a position before Y that represents the current state of the sequence. T is thus inserted at position 0 shifting Y to position 1, etc. The editing sequence that corresponds to this case is thus $[(Y, 0), (T, 0), (R, 0), \ldots]$.

- Case 1: Since the function *allocPath* allocates the leftmost branches, the following editing sequence $[(Q, 0), (W, 1), (E, 2), \ldots]$ leads to the paths $\langle[1], Q\rangle$, $\langle[2], W\rangle$, $\langle[3], E\rangle$, etc. In this case, the depth of the tree never grows. In this regard, this function *allocPath* is very efficient in terms of the size of the allocated identifiers.
- Case 2: The editing sequence $[(Y, 0), (T, 0), (R, 0), \ldots]$ leads to an increase of the depth of the tree at each element insertion. Indeed, as an element gets the smallest value at its level (cf. the allocation function), there is no way to insert a new element before at the same level hence the new level. The resulting sequence is $\langle[1], Y\rangle$, $\langle[0.1], T\rangle$, $\langle[0.0.1], R\rangle$, etc. Consequently, the size of the paths grows very fast.

This example shows how the insertion order impacts the length of the allocated paths. Unfortunately, the insertion order cannot be predicted, nor the size of the final sequence. Prior work on sequences often made the assumption of a 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]. Indeed, the editing depends on the type of the document and to the activity for example when correcting a document the editing in mainly random as the insertions/deletions may correspond to errors. Consequently, we are looking for an allocation function which provides identifiers with a sub-linear spatial complexity compared to the number of insertions whatever is the editing sequence.

3. LSEQ

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

The function allocPath chooses the path associated with each element in order to encode its relative position with regard to its adjacent elements in the sequence. For the sake of performance, the aim of *allocPath* is to keep the underlying tree with a small depth. Three components compose *allocPath*. Each of these components fails to provide an efficient allocation function. Nevertheless, their composition allows to get the best of each by cancelling 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 but 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 $p_1 \subset \mathbb{N}_{<32}$ and $p_2 \in \mathcal{P}$ such that $|p_2| = 2$, then $p_2 \subset \mathbb{N}_{<32}.\mathbb{N}_{<64}$ etc. Similarly, let i be 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 using $log_2(32) = 5$ bits, p_2 is encoded using $5 + 6 = 11$ 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 function *allocPath* uses two sub-allocation functions both designed for monotonic editing, i.e., inserting repeatedly at an adjacent position of the previously inserted element. These are application dependent; one is well-suited to end-editing (left-to-right) while the other is well-suited for front-editing (right-to-left).

3. The function *allocPath* 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. This provides the independence of the allocation function from any editing strategy/policy. 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 and shared within the document.

Combining the three components provides identifiers with a sub-linear average size 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*. First, it 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 transfers the call to one of the two a sub-allocation functions *endEditing* and *frontEditing*. These allocation functions work similarly: (#1) Get the depth and the interval of the new path to allocate. (#2) Limit the range within the level for the allocation of the new path. (#3) However, while *endEditing* uses the previous element in the sequence and concatenates the value of a new edge in the tree to build the new path, *frontEditing* does the opposite. Thus, by #2 the allocation function is efficient in monotonic editing and by #3 the allocation is

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
```

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: Similarly to the example of Figure 3) the example given in Figure 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, the exponential tree of LSEQ starts with an arity 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 Figure3), the antagonist scenario does not increase very much 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.

THE FUNCTION ALLOCDIS creates the disambiguators which have two goals. It ensures the uniqueness of the triples, even in presence of concurrency and it guarantees that triples can always be inserted between two other

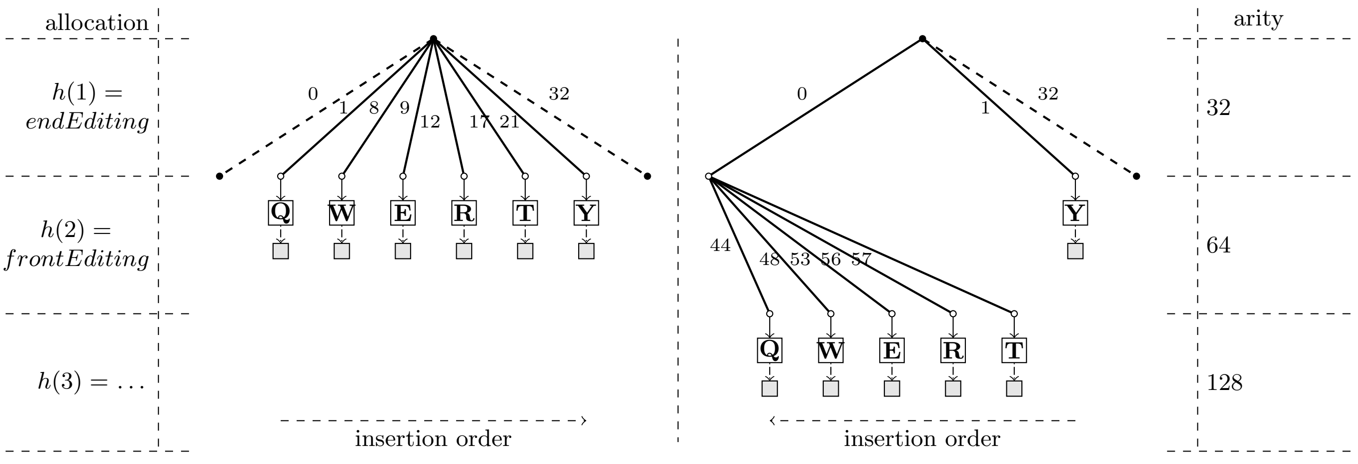

Figure 4: Example of LSEQ's exponential trees with two antagonist editing behaviours to create the sequence of characters $QWERTY$. Thanks to its hash function, LSEQ transfers the allocation of paths to the sub-allocation functions designed for front-editing and end-editing at first and second level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

triples even when the paths of the latter are identical. The example of Figure 3 illustrates the need of comparisons that preserve the order given by the paths and also examines 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 sequences.

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

```
let $site$                                    ▷ the unique site identifier
let $counter \leftarrow 0$                    ▷ a local counter

function ALLOCDIS($p \in \mathcal{I}$, $path \in \mathcal{P}$, $q \in \mathcal{I}$) $\rightarrow \mathcal{D}$
    let $dis \leftarrow []$;
    $counter \leftarrow counter + 1$;
    for $i$ from 1 to $|path|$ do
        $dis[i] \leftarrow \langle site,\ counter\rangle$;
        if $path[i] = q.P[i]$ then $dis[i] \leftarrow q.D[i]$;
        if $path[i] = p.P[i]$ then $dis[i] \leftarrow p.D[i]$;
    end for
    return $dis$;
end function
```

Algorithm 3 describes the allocation of a disambiguator which preserves the intention of the peer that performed the insertion. Identically to Lamport timestamps, 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.

Let us go back to the example in Figure 1. In this tree, collaborator 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 collaborator p_2, and $\delta_T = [\langle 3,\ 1\rangle]$ meaning that it is the first insertion of collaborator p_3. The resulting path associated with 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 $[\langle 2,\ 1\rangle,\ \langle 1,\ 1\rangle]$. This way, p_1 inserts a new character between E and R. As we get the path $[3.0.X]$, 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 an adjacent element (after or before exclusively). For instance, when an author writes $QWERTY$, she generally starts from the first letter Q to the last letter of the word Y. On the opposite, insertions in a logfile are usually performed at the beginning for practical reasons.

As a consequence, we focus on 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 a human collaborator 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 reduced to the space complexity of the identifiers of LSEQ.

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 insert operations 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:

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

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 compared the other cases. Furthermore, it is worth noting that usually, the authors do not write a document in a 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. In the case of random editing, the space complexity is asymptotically optimal $O(log\, n)$ while it is very interesting in a "normal setting" $O((log\, n)^2)$.

	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. The first part describes the core of the distributed CollaboRATive Editor CRATE by filling in the blanks left on causality tracking and eventual delivery of operations. The second part concerns the experiments that aim to:

1. Validate the space complexity of LSEQ and highlight the improvement over the state-of-the-art.
2. Show that CRATE scales with respect to the number of collaborators simultaneously involved in a document editing.
3. Show that concurrency does not impact the size of identifiers. Consequently, the space complexity of the identifiers can be studies in scenarios without concurrency.

4.1. CRATE

CRATE (CollaboRATive Editor) is a prototype of decentralised editor. We assume that collaborators can join the document editing and leave when they want. The group of collaborators can be seen as a peer-to-per network. Hence, CRATE uses a gossip dissemination protocol to scale with respect to the number of collaborators/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. Each peer broadcasts its messages and when a peer receives a specific message for the first time, it re-broadcasts this message. However, Section 2 states that strong eventual consistency is ensured upon the assumption of eventual delivery, and unfortunately, the gossiping broadcast is only reliable with high probability. To make it reliable, CRATE includes an additional anti-entropy protocol to retrieve the possibly missing operations (cf. Appendix B).

Section 2 also states that sequences can be seen as conflict-free data types under the condition that deletion of a particular element cannot precede its insertion. To ensure this condition, CRATE uses a vector structure called interval version vector [22] (IVV) whose values are version numbers called dates.

Similarly to version vectors, IVVs require a vector with one entry per peer involved in the authoring at the difference that 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 a local space complexity of the same order 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.

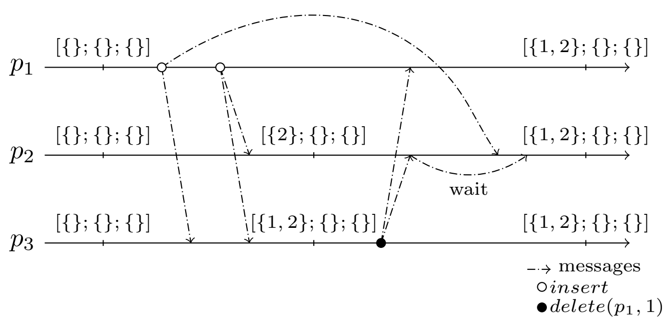

Figure 5: Timeline of operations performed by 3 collaborators. For the sake of simplicity, sets represent the intervals stored in the local vectors. In this example, the *insert* operations concern any element while the *delete* operation targets the first insertion of collaborator p_1.

The delivery algorithm using the IVV can be found in Appendix B.

Figure 5 depicts an example of causality tracking using an IVV with all possible configurations. Each of the three peers initialises its 3-entry vector with empty intervals. First, peer p_1 performs an insertion and broadcasts it to p_2 and p_3. The operation conveys the unique peer identifier p_1 and its corresponding stamp 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 p_1 conveys the stamp 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. 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 environments hosted the experiments. First, 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. Then, multiple machines distributed in a cluster of the Grid'5000 testbed are considered. 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 random editing, when a peer inserts an element, it randomises the position of the new element in the document following a uniform distribution. In monotonic editing, when a peer inserts 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. In contrast to this, 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 maximum 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 in 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[2] under the MIT license. Based on this implementation, we developed a prototype of distributed CollaboRATive Editor CRATE[3] following the outlines of this paper.

Objective: Validate the space complexity analysis of LSEQ. In particular, when the editing behaviour is monotonic, LSEQ has a polylogarithmic upper-bound on space complexity with respect 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 sequence structure Logoot [17, 23].

Description: A single machine hosts two peers communicating via a peer-to-peer connection. They produce 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 either CRATE, or a version of CRATE using Logoot instead of LSEQ. According to [7], Logoot constitutes the sequence 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 uses a constant base of 2^{16}. We measure the average size of messages transiting through the peers during the experiment.

Results: Figure 6 shows the result of the experiments. The y-axis corresponds to the average size of messages 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 sequences with a barely noticeable difference. On the other hand, for a monotonic editing behaviour Logoot exposes a linear growth of the size of its messages while the LSEQ-based sequence has a polylogarithmic validating the proof

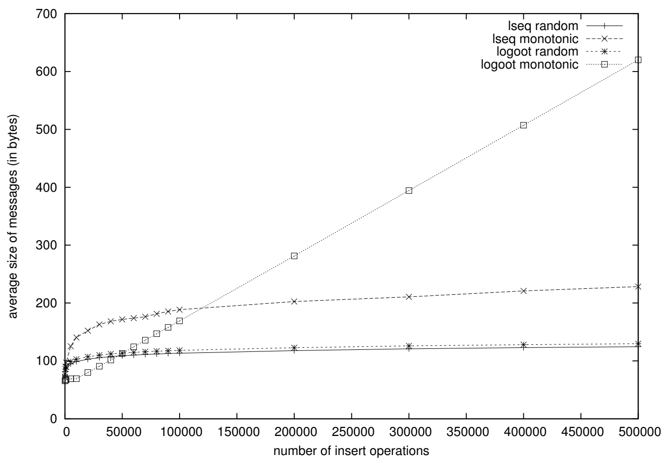

Figure 6: Average size of the messages exchanged between two peers during repeated insertions of elements in the sequence. The measurements concern two editing behaviours (monotonic and random) with two CRATE using different sequence structures (Logoot and LSEQ-based). The replicated document grows up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

given in Section 3. Thus, even if Logoot starts with a smaller size messages, it quickly becomes less efficient than the LSEQ-based sequence when the number of insertions increases.

Reasons: The random editing behaviour is slightly better for the LSEQ-based sequence compared to Logoot because LSEQ starts with a lower departure base. Nonetheless, a random editing behaviour leads to a same space complexity for both sequences. For 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: Show that CRATE scales in terms of the number of peers. In other words, the size of the network does not impact 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 produce 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 varies from 2 to 450. The generation of 166 operations per second is uniformly distributed among the peers and time. We measure the average size of messages.

Results: Figure 7 shows the results of these experiments. The y-axis corresponds to the average size of messages in bytes. The x-axis corresponds to the number of in-

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

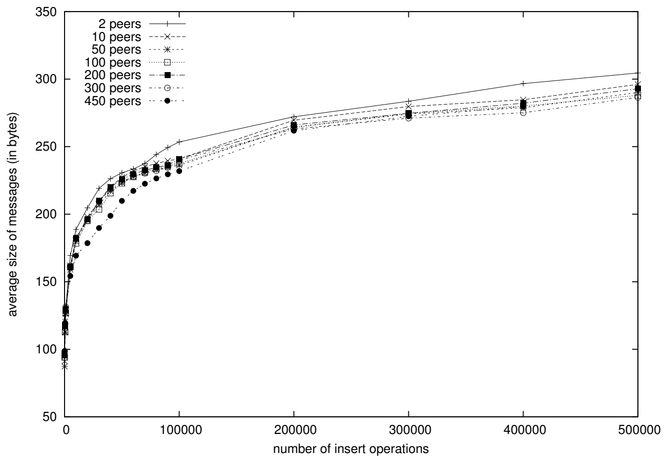

Figure 7: Average size of messages sent during a session of document editing. The number of peers varies from 2 to 450 collaboratively producing a document of 500 thousand characters in 50 minutes. The editing behaviour is monotonic.

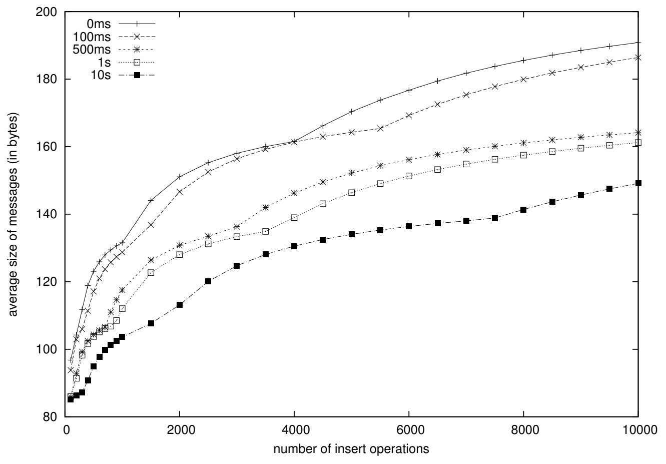

Figure 8: Average size of messages sent by peers during sessions of document editing. Ten peers produce documents of 10 thousand characters by repeatedly inserting at the end of the document at the rate of 3 operations per second. Five editing sessions with different network latencies are studied.

sert operations performed on the sequence. Figure 7 shows that, independently of the number of involved peers, 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 size of messages reflects the size of the identifiers and the additional causality tracking metadata. In Section 4.1 that describes CRATE, we saw only two scalars are required 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 the number of peers. Figure 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 a 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 a sequential execution. Therefore, the average size of messages slightly decreases when the number of peers increases.

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

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

Results: Figure 8 shows the results of the measurements done with the different editing sessions. The y-axis corresponds to the average size of messages in bytes and 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 given by the space complexity analysis in Section 3. Also, when the latency increases, the average size of messages decreases. Thus, the curve that corresponds to an almost sequential execution represents the largest 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 due to the latency instead of the number of peers. 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 size are due to an addition of a level in the paths. Indeed, depending on the depth of the tree, the average tends to a specific value. When the depth increases, this average value suddenly changes to an higher value. A finer grain of measurements in this experiment allows us observing these changes.

4.3. Synthesis

This section detailed a complete distributed collaborative editor. 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 classes. The Operational Transformation (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, this first class can be divided into two subclasses centralised and decentralised. Centralised approaches [1] serialise the order of operations on a central server easing the convergence. However, the topology in itself implies a single point of failure, privacy issues, economic intelligence issues, and scalability issues. Decentralised approaches [25] require a version vector to identify the generation context of the received operations. It transforms the arguments of the received operation against its concurrent and older operations to consistently execute it without undoing the latter operations. While the local operation is very efficient, the high price in case of concurrency must been paid at all distant sites. As a 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 LSEQ that does not require a logfile and whose complexity depends of the insert operations only, sends only two scalars in messages to guarantee the convergence, and supports concurrency.

The second class, besides the OT class, is the class based on conflict-free replicated data types (CRDTs) [5, 6]. This class shares 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 can be slit into two sub-classes, the tombstone class and the variable-size identifiers class. The tombstone 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 affect the overall performance. Thus, the complexity depends on the number of insert and delete operations. Including the deletes in the complexity is problematic because, for instance, in Wikipedia documents subject to vandalism, delete operations are very common. As a consequence, a page may contain few lines and yet use a large amount of storage due to the hidden tombstones. On the other side, 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 also encode the order of elements and hence the space complexity of these approaches is crucial and depends only on the number of insert operations.

In the literature, two kinds of allocation functions exist for the identifiers. Let us consider an insertion of b between two elements a and c; let id_a and id_c be their respective identifiers where $id_a < id_c$. Both kinds of allocation functions aim to provide the shortest identifiers possible. Thus, the maximum size of the identifier id_b corresponding to b is always: $min(|id_a|, |id_c|) + 1$. 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 the average half the level of identifiers in a branch. The second allocation function comes from the observations made on a Wikipedia corpus that favours end-editing. When the editing behaviour complies with this assumption, the average size of identifiers remains linear. None of the two functions provides reasonable size identifiers when associated with the dual editing behaviour.

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

LogootSplit [16] proposes an orthogonal solution to reduce the metadata generated by sequences with variable-size identifiers. 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 could use LSEQ to benefit from its advantages.

Using LSEQ, a sequence still requires some form of causality tracking. However, causality tracking is still an issue in distributed network subject to churn [26]. A 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 feasible 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 notify this clearly. Even though version vectors (or other structures with the same space complexity) are necessary to accurately track causality [28], there exist trade-offs between space complexity and accuracy. CRATE chooses to track 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 the context of collaborative editing, this paper proposed an identifier allocation function LSEQ for building sequence replicated data structures. The obtained sequences enjoy good space and time complexities. The proof on space complexity demonstrates logarithmic, polylogarithmic, and quadratic for, respectively, random, monotonic, and worst-case insertions. Using this allocation function and interval version vectors to track causality, we developed a distributed collaborative editor called CRATE. This editor is fully decentralised and scales in terms of the number of peers, concurrency, 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 and high latency. The experiments confirmed the space complexity of LSEQ and 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 can be seen as a serious competitor for current trending editors (e.g. Google Docs, SubEthaEdit, . . .), allowing massive 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.

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:

- $X : \mathbb{N} \rightarrow \mathcal{A}$
- $Y : \mathbb{N} \rightarrow \mathcal{A} \times \mathcal{M}$
- $Z : \mathbb{N} \rightarrow \mathcal{A} \times \mathcal{I}$
- $gen\beta : \mathcal{M} \rightarrow \mathcal{I}$
- 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: $(Z : Y \times \mathbb{N} \rightarrow Z)$:
  - $Z(\varnothing, \_) \rightarrow \varnothing$
  - $Z(\_, 0) \rightarrow \varnothing$
  - $Z(Y, i) \rightarrow$ 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 Delivery protocol

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

INITIALLY:
    for $k$ from 0 to $|r|-1$ do $s[k] \leftarrow \varnothing$;

LOCAL UPDATE:
    for $k$ from 0 to $|r|-1$ do $s[k] \leftarrow r[k]$;
    on insert (args):
        $s[i] \leftarrow r[i].add(r[i].last() + 1)$;
        $broadcast('insert', alloc(args))$;
    on delete (args):
        $broadcast('delete', args)$;

RECEIVED UPDATE:
    for $k$ from 0 to $|r|-1$ do $s[k] \leftarrow r[k]$;
    on insert (args):
        let $\langle src, cnt \rangle \leftarrow unique(args)$;
        if $(\neg(r[src].contains(cnt)))$ then
            $s[src] \leftarrow r[src].add(cnt)$;
            $deliver(m)$;
        end if
    on delete (args):
        let $\langle src, cnt \rangle \leftarrow unique(args)$;
        if $(r[src].contains(cnt))))$
            then $deliver(m)$;
            else $delay(m)$;
```

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.

Example 1: 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 Anti-entropy protocol

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

Algorithm 5 depicts the anti-entropy event added to 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.