h-LSEQ: a Polylog Sequence Encoding for Collaborative Editing

Brice Nédelec, Pascal Molli, and Achour Mostefaoui

Abstract—The recent development of distributed collaborative tools such as Google Docs, SubEthaEdit, Git, has intensified the interest in an efficient data structure that allows a virtually unlimited number of users to read and modify a document in real-time. The Conflict-free Replicated Data Type (CRDT) for sequences is an abstract data type that provides two basic updating operations on the sequence: insertion and deletion of an element. When the sequence is a document, the elements can be either characters, lines, or paragraphs. CRDTs for sequences associate a unique and immutable identifier to each element in the sequence. The allocation of the identifiers is crucial to preserve the intended total order on the elements while maintaining good performance. Unfortunately, it is a non-trivial problem. Recently, the allocation function *h*-LSEQ empirically showed a sub-linear space complexity on the size of its identifiers making CRDT-based distributed collaborative editors a good alternative to the aforementioned trending editors. In this paper, we introduce an abstraction of the allocation of identifiers problem. We enforce the safety of *h*-LSEQ by giving the proof of its polylogarithmic space complexity. Finally, we evaluate an *h*-LSEQ-based CRDT and show that it perfectly fits the requirements of the distributed collaborative editing context.

Index Terms—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 tools such as Google Docs, SubEthaEdit, or Git, has not ceased to increase. Such editors allow distributing the work across space, time and organizations.

Most of the distributed collaborative editors use an optimistic replication approach [1], [2] to provide high availability of the documents. Thus, each user involved in the collaboration owns a local replica of the document and directly modifies it. These changes are sent through the network to all collaborators where they are integrated to their own 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 [3].

The Conflict-free Replicated Data Type [4], [5] (CRDT) for sequences is an approach that belongs to optimistic replication. It constitutes a simple and efficient mean to build distributed collaborative editors. To ensure the convergence property, CRDTs allocate a unique and immutable identifier to each element in the sequence. When the sequence is a document, and depending on the granularity, the elements can be either characters, lines, or paragraphs. The set of identifiers is paired with a total order, and this latter actually makes the sequence.

CRDTs for sequences use an allocation function for their identifiers. These approaches provide two updating operations: insert and delete. However, either depending on the type of the performed operations [6], [7], [8], [9], [10], [11], [12], [13], or the arguments of the performed operation [9], [14], [15], 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 *h*-LSEQ [16], [17]: an allocation function that empirically provides identifiers enjoying a sub-linear upper bound on their size. Using *h*-LSEQ, CRDT-based distributed collaborative editors scale in terms of document size. Assuming the use of *h*-LSEQ and a CRDT that scales with respect to the number of collaborators, it constitutes an alternative to trending editors such as Google Docs [18], without any service provider and any service limitation. In particular, it would alleviate economic intelligence issues brought by third-party hosts.

Withal, the sub-linear space complexity of *h*-LSEQ was only a conjecture [17]. The experimentation confirmed the expectation afterwards. In this paper, we provide: (i) an abstraction of the problem of the allocation of identifiers. (ii) A consolidation of *h*-LSEQ with a proof on its space complexity. (iii) A full outline of an *h*-LSEQ-based CRDT for sequences that allows a straightforward building of a working distributed collaborative editor. (iv) A validation of the space complexity of *h*-LSEQ and the scalability of *h*-LSEQ-based CRDT by measuring the size of the operations sent through the network. Parts of the experiments were done on the Grid'5000 testbed and involve up to 450 peers creating a document of half a million characters.

The remainder of this paper is organized as follows: Section 2 provides the necessary background to un-

- *The authors are affiliated to Universit de Nantes, LINA, FRANCE. E-mail: first.last@univ-nantes.fr*

derstand the CRDTs for sequences. Section 3 details the "From Mutable to Immutable" abstract problem generalizing the problem of the allocation of identifiers. Section 4 follows with a formal description of h-LSEQ and the proof of its space complexity. Section 5 describes the full h-LSEQ-based CRDT for sequences. Then, it highlights its scalability while validating the space complexity analysis of h-LSEQ. Section 6 reviews related works. Finally, Section 7 concludes this paper and discusses the perspectives.

2 Preliminaries

CONFLICT-FREE REPLICATED DATA TYPES belong to the optimistic replication approach. Therefore, each peer involved in the collaboration owns a local replica of the CRDT. The operations performed by a peer directly impact his replica. In a second time, the peer broadcasts the result of the former to the remote peers where they are integrated. CRDTs provide strong eventual consistency [4] 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.

A CRDT aims to avoid the difficult and error-prone task of solving conflicts. There exist CRDTs for counters, sets, graphs, ...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. These operations are commutative. Indeed, for each pair of operations ($\langle$insert; insert$\rangle$, $\langle$insert; delete$\rangle$, $\langle$delete, delete$\rangle$), it does not matter which operation is integrated first. Thus, CRDTs do not genuinely require a causality tracking mechanism except for the particular case of $\langle$insert(e); delete(e)$\rangle$ that does not commute. Indeed, the delivery of the delete targeting a particular element cannot precede its insertion.

Unique and immutable identifiers are associated with the elements. These identifiers encode the total order among the elements. Thus, when a peer performs an insert operation, CRDTs allocate a unique and immutable identifier preserving the desired total order.

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 in 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. For example, and similarly to decimal numbers, 3.1 could be the new identifier associated with the character R. If the peer inserts a new character between E and R, a new identifier must be allocated between 3 and 3.1. Once again, the space must be enlarged 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.

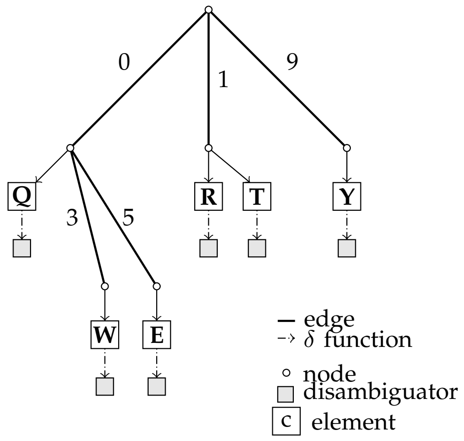

Fig. 1. Underlying tree of variable-size identifiers CRDTs for sequences. The paths $\mathcal{P}$ correspond to the concatenation of the edges from the root to the elements. The elements $\mathcal{A}$ are characters. The disambiguators $\mathcal{D}$ are associated to each element and are reachable through the function δ. When totally ordered, they compose the sequence $QWERTY$.

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). For instance, the real number $3.0X$ of the prior example corresponds to the sequence $[3.0.X]$ of concatenations. Such a 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 example, the character R is accessible following the path 3 then the path 1. More formally, the variable-size identifiers CRDTs for sequences replicate a tree $\mathcal{T}$ where the leaves or the intermediary nodes are the elements of the sequence $\mathcal{A}$. The tree $\mathcal{T}$ is (i) a set of pairs $\langle \mathcal{P} \subset \{\mathbb{N}\}^*, \mathcal{A}\rangle$. (ii) A total order $(\mathcal{P}, <_\mathcal{P})$ (iii) $\forall p.c \in \mathcal{P}$, $p \in \mathcal{P}$ and $c \in \mathbb{N}$.

EXAMPLE 1: Fig. 1 shows the underlying tree of a variable-size identifiers CRDT for sequences. The tree has an arity of 10 to stay with real numbers example. In this scenario, the elements of a sequence $\mathcal{A}$ are characters and the objective is the string $QWERTY$. In a first time, the sequence is initialized with the beginning and the end of the sequence: the characters Q and Y. Since all future insertions target a position between them, they obtain the minimum and maximum available paths of the first level of the tree, i.e., [0] and [9]. The resulting pairs are: $\langle$[0], $Q\rangle$ and $\langle$[9], $Y\rangle$. Then, two peers concurrently insert the characters R and T. The CRDTs concurrently allocate an identical path [1] to the characters. Then, a peer inserts the missing characters W and E between Q and R. However, since there is no room for further elements at this position in the first level of the tree, the paths become a concatenation of two numbers. The elements W and E are associated with the paths [0.3] and [0.5] respectively. Let us consider that the paths are the fractional part of real numbers. Following an ascending order, the resulting set of pairs $\langle path,\ element\rangle$ composing the tree is: { $\langle[0],\ Q\rangle$, $\langle[0.3],\ W\rangle$, $\langle[0.5],\ E\rangle$, $\langle[1],\ R\rangle$, $\langle[1],\ T\rangle$, $\langle[9],\ Y\rangle$ }.

2.2 Disambiguation of concurrent cases

Despite $<_\mathcal{P}$ being a total order locally, it becomes a partial order when multiple peers are involved. A path in $\mathcal{P}$ can be coupled with multiple elements in $\mathcal{A}$. Indeed, a peer directly modifies its replica without knowing the concurrent modifications performed by other peers. Chances exist that they choose identical paths. However, they will learn about it only when they receive the corresponding operations from each other. Still, replicas must converge to an identical state. Therefore, disambiguating the order among such elements is necessary. The disambiguation bijective function δ associates a disambiguator to each pair of $\mathcal{T}$. (iv) $\delta : \mathcal{T} \to \mathcal{D}$ and there is (v) a total order ($\mathcal{D}$, $<_\mathcal{D}$).

EXAMPLE 2: Fig. 1 depicts a tree containing 6 elements. Each of these elements has a path. However, only 5 distinct paths exist due to the concurrent insertions of R and T. The function δ associates a disambiguator to each of the elements. In this example, the δ function provides disambiguators in $\mathcal{D}$ such that $\delta([1],\ R) <_\mathcal{D} \delta([1],\ T)$. It allows distinguishing and ordering R and T.

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 ($\mathcal{T}$, $<_\mathcal{T}$).

Let $\mathcal{I}$ be the set of unique triples $\mathcal{I} : \mathcal{P} \times \mathcal{A} \times \mathcal{D}$. Additionally, $\mathcal{I}$ is paired with the total order $<_\mathcal{T}$. 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. Let $p, q \in \mathcal{I}$ such that $p <_\mathcal{T} q$. Let $allocPath : \mathcal{P} \times \mathcal{P} \to \mathcal{P}$ be a function returning a path $path$ in $\mathcal{P}$ and let $allocDes : \mathcal{I} \times \mathcal{I} \to \mathcal{D}$ be a function returning a disambiguator des in $\mathcal{D}$ such that $p <_\mathcal{T} \langle path,\ e \in \mathcal{A},\ des\rangle <_\mathcal{T} q$.

Two operations allow to update the sequence, and hence, the tree. With the prior formalization, we define the eventual effects of operations as:

- $insert(p \in \mathcal{I},\ \alpha \in \mathcal{A},\ q \in \mathcal{I})$:
  $\mathcal{T} \cup \langle allocPath(p.P,\ q.P),\ \alpha,\ allocDes(p,\ q)\rangle$
- $delete(i \in \mathcal{I})$: $\mathcal{T} \backslash \{i\}$

2.3 Choosing the rightful path

The $allocPath$ function chooses a path in the tree between two other paths. However, since the tree can always have sub-trees, the number of possible paths is infinite, and so is the number of $allocPath$ functions. However, to keep the system with good performance, the $allocPath$ function should choose among the available paths with the smallest number of concatenations. 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.

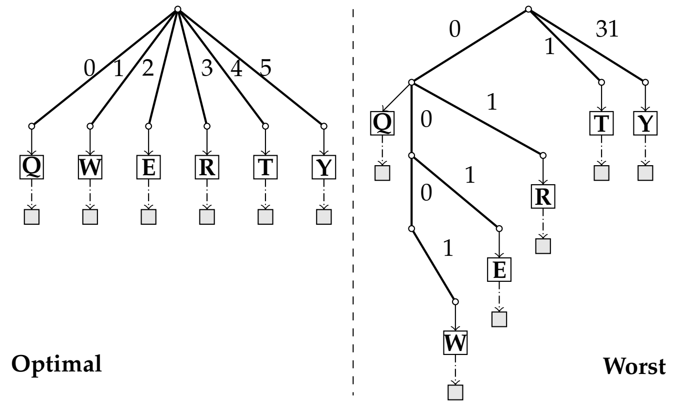

Fig. 2. Illustration of the allocation of paths with the optimal choices against the worst choices. Legends are similar to Fig. 1. In both cases, the scenario is similar and results in the same sequence $QWERTY$. However, while the optimal tree has a depth of 1, the worst tree grows up to a depth of 4.

EXAMPLE 3: Fig. 2 illustrates the difficulty of allocating paths with an acceptable length. Similarly to Fig. 1, it represents the underlying trees of two variable-size identifiers sequences but with different allocation functions. In both cases, the scenario is the following: the sequence is initialized with Q and Y. Then, the elements are inserted at the right of the character Q starting from the T to the W. An optimal allocation function needs to know the resulting sequence. Therefore, it can allocate exactly 6 branches and, when the insertions occur, it assigns the rightful path to each element. Here, it initializes the sequence with the two pairs: $\langle[0],\ Q\rangle$, and $\langle[5],\ Y\rangle$. Then, the insertion of the element T results in the pair $\langle[4],\ T\rangle$, the element R in $\langle[3],\ R\rangle$ etc. On the other side, the worst choice of allocation of the paths does not know the insertion pattern of the sequence nor the final sequence. Here, it assumes that insertions will be at the end. As a consequence, when the insertions occur, it chooses to allocate the leftmost branch of the tree to leave more available branches for future insertions. However, the elements are inserted in the opposite way. Each insertion increases the depth of the tree, i.e., the path size is linearly growing. In this scenario, the worst allocation function begins by initializing the sequence. The lowest path is straightforward: $\langle[0],\ Q\rangle$. On the opposite, the highest path is not trivial. Indeed, since the allocation function does not know the size of the final sequence, it must foresee the number of available paths. In this example, the path allocation function chooses $\langle[31], Y\rangle$. Then, the element T arrives resulting in the pair $\langle[1], T\rangle$. The element R arrives, but since there is no room for further insertions between Q and T, the tree must grow. The resulting pair is $\langle[0.1], R\rangle$. Each newly inserted character leads to a growth of the tree. As a consequence, the tree becomes unbalanced and impacts the performance of the system.

3 FMI Problem

This section introduces the "From Mutable to Immutable" problem as an abstraction of the allocation of identifiers. A source sends a sequence of operations with mutable indexes through an order-preserving channel. The receiver transforms on-the-fly the mutable indexes to immutable pairs without an *a priori* knowledge of the upcoming operations. A total order defined among the set of immutable pairs allows retrieving a sequence of elements identical to the initial one.

[From Mutable to Immutable problem] Given:

- A **source** $\mathcal{S}$ holding a final sequence $\mathcal{S}_f$ of elements such that $\mathcal{S}_f(i) \to \alpha$.
- Source $\mathcal{S}$ also holds a permutation of $\mathcal{S}_f$ called $\mathcal{S}_H$ such that $\mathcal{S}_H(i) \to \langle \alpha \in \mathcal{S}_f, \beta \in \mathcal{M}\rangle$, where $\mathcal{M}$ is a set of elements paired with a total order.
- There exists a function $P^{-1}(\mathcal{S}_H \in \mathcal{S}_H^*) \to \mathcal{S}_f$, where $\mathcal{S}_H^*$ is the set of all permutations of $\mathcal{S}_f$.
- A **receiver** $\mathcal{R}$ holding a set $\mathcal{R}_f$ of immutable couples $\langle \alpha \in \mathcal{S}_f, \gamma \in \mathcal{J}\rangle$, where $\mathcal{J}$ is a set of elements paired with a dense total order.
- γ uniqueness: $\forall\ \langle \alpha 1,\ \gamma 1\rangle,\ \langle \alpha 2, \gamma 2\rangle \in \mathcal{R}_f,\ \gamma 1 = \gamma 2 \Rightarrow \alpha 1 = \alpha 2$
- γ order-preservation: $\forall\ \langle \alpha 1,\ \gamma 1\rangle,\ \langle \alpha 2,\ \gamma 2\rangle \in \mathcal{R}_f$, $(\gamma 1 <_{\mathcal{J}} \gamma 2) \Leftrightarrow (\alpha 1 <_{\mathcal{S}_f} \alpha 2)$

Furthermore:

- $\forall t \,|\, 1 \leq t \leq |\mathcal{S}_H|$, $\mathcal{S}$.send$(\mathcal{S}_H[t], t)$
- $\forall \mathcal{S}$.send$(o_t \in \mathcal{S}_H, t \in \mathbb{N})$, $\mathcal{R}$.receive$(o_t,\ t)$
- $\forall \mathcal{R}$.receive$(o_t \in \mathcal{S}_H, t \in \mathbb{N})$,
  $\langle o_t.\alpha$, generate$\gamma(o_t.\beta,\ t)\rangle \in \mathcal{R}_f$,
  with **generate**γ: $\mathcal{M} \times \mathbb{N} \to \mathcal{J}$

The problem is to find an optimal generateγ such that: $\nexists$ generateγ' such that:

$$\sum_{\mathcal{S}_H \in \mathcal{S}_H^*} |R'_f|_2 < \sum_{\mathcal{S}_H \in \mathcal{S}_H^*} |R_f|_2 \tag{1}$$

Example 4: Fig. 3 illustrates the abstract problem. The source $\mathcal{S}$ holds a final sequence $\mathcal{S}_f$ composed of abc. Source $\mathcal{S}$ also holds a permutation of $\mathcal{S}_f$. The β values are the indices of insertion of α chosen in the set of natural integers. Their mutability is highlighted by the operations o_2 and o_3 which target a same index. As

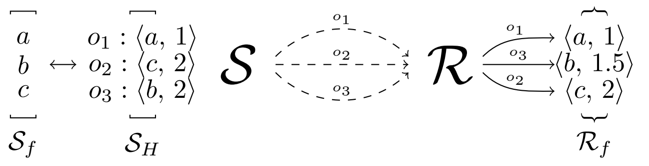

Fig. 3. $\mathcal{S}$ sends a sequence $\mathcal{S}_H$ of 3 insertions creating the sequence $\mathcal{S}_f$ composed of abc. Each time an operation is received, $\mathcal{R}$ creates an immutable pair with the received elements and a position by calling generate$\gamma(index)$. Ordered by positions, $\mathcal{R}_f$ builds the sequence abc.

a consequence, the element c shifts when o_3 is applied. The source $\mathcal{S}$ sends the sequence of operations $\mathcal{S}_H$ to the receiver $\mathcal{R}$. With t from 1 to $|\mathcal{S}_H|$, $\mathcal{R}$ creates pairs $\langle o.\alpha, \text{generate}\gamma(o.\beta,\ t)\rangle$. For instance, the third operation is transformed to $\langle b,\ 1.5\rangle$. This transformation does not modify the other pairs. In particular, it does not modify the result of o_2. The final set of immutable pairs $\mathcal{R}_f$ paired with a total order $<_{\mathcal{J}}$ results in the same initial sequence $\mathcal{S}_f$: abc.

Put back in the context of distributed collaborative editing, the set $\mathcal{S}_f$ corresponds to the intention of collaborators. In other words, this is the final document that peers aim to achieve. The set $\mathcal{S}_H$ is the editing sequence performed by the collaborators to reach that goal. However, since using mutable identifiers to order elements can lead to inconsistencies, $\mathcal{R}$ transforms them into immutable identifiers that can be shared by all collaborators while maintaining the desired order.

In text editing, most of the editing behaviours can be summarized as two basic editing behaviours or a composition of them. (i) The random editing behaviour where the author seems to insert new elements at random positions in the sequence. For instance, this behaviour mostly arises when syntactic corrections are performed, e.g., the user writes $wites$ and realizes 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 another specific element. For instance, when a user writes $writes$, she generally starts from the w and finishes with the s.

Finding an optimal function $generate\gamma$ for any permutation $\mathcal{S}_H$ is impossible (there always exists a better allocation function). Nonetheless, focusing on the random and the monotonic editing behaviours provides a first answer restrained to text editing or alike contexts.

4 *h*-LSEQ

h-LSEQ is an allocation function usable within variable-size identifiers CRDT for sequences. *h*-LSEQ is common to all the collaborators involved in the sequence editing and provides unique and immutable identifiers encoding the desired total order. This section describes the choice of paths and the choice of disambiguators through the

$allocPath$ and $allocDes$ functions respectively. Also, it provides the proof of the space complexity of the identifiers allocated by h-LSEQ.

THE ALLOCPATH FUNCTION chooses which path will be included with the element in order to encode its relative position with regard to the previous and to the next element 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 fills in their respective deficiency.

The first component is an exponential tree. Regarding the formalization 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 size 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 $1 \leq l \leq min(|p_j|, |p_k|)$ 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 $Y[1] < Z[1]$.

The definition of the total order $<_\mathcal{P}$ states that p_j is lower than p_k iff once their common prefix truncated, the first number of p_j is lower than the first number of p_k.

The $allocPath$ function uses two sub-allocation functions. Both are designed for monotonic editing, however, one is specialized in end-editing (left-to-right) while the other is specialized in front-editing (right-to-left).

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 to avoid the time-consuming task of processing a more clever choice. This random choice does not favour any editing behaviour while remaining very efficient. Also, it is shown in [16] that, with such antagonist sub-allocation functions, all peers must make identical choices. To reach that goal, they all use a similar hash function initialized 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.

EXAMPLE 5: Fig. 4 depicts the result of a scenario using the underlying tree induced by the h-LSEQ allocation function. The tree has two levels. The root node has 32 possible children and each of these children has 64 possible children etc. The elements of the sequence $QWERTY$. To create this sequence and the attached tree, the sequence is initialized with Q and Y with the respective paths [0] and [31]. Also, the sub-allocation function designed for front-editing is attached to the first level of the tree. In a second time, two peers concurrently insert the elements W and T in the middle of the string QY. Both peers use the same sub-allocation functions. In this case, since the new paths land in the first level of the tree, h-LSEQ uses its front-editing allocation function. As a consequence, the paths are close to the next identifier ([31]) to leave more available paths for future insertions supposedly in front. In this example, the $allocPath$ function provides the same path to both peers. Then, a peer completes the sequence with the three missing characters E, and R, between W and T. Both paths are sequences composed of the only number 29. Thus, inserting at the first level is impossible. The depth of the tree must increase. Since there were no prior insertions at the second level of the tree, $allocPath$ randomizes a sub-allocation function and assigns it to this level. Now, the $allocPath$ function will use an end-editing allocation function for identifiers landing in the second level of the tree. Consequently, $allocPath$ allocates paths near to the previous bound (implicitly [29.0]). In this example, it provides the paths [29.4], and [29.13], to E, and R respectively. The resulting sequence is whether $QWERTY$ or $QTERWY$.

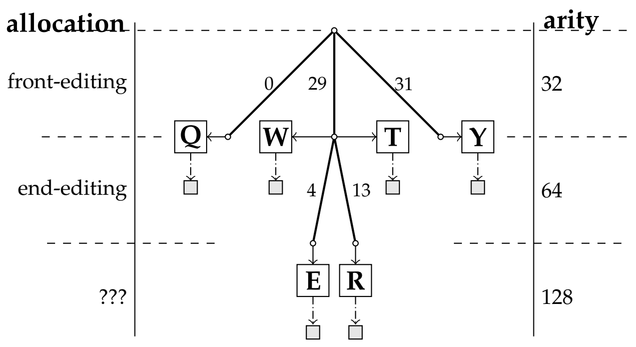

Fig. 4. Example of the h-LSEQ's exponential tree. The result is the sequence of characters $Q\ W\ E\ R\ T\ Y$. h-LSEQ uses the sub-allocation functions designed for front-editing and end-editing at first and second level respectively. The allocation functions for the higher levels of the tree are not assigned yet.

THE ALLOCDES FUNCTION allocates the disambiguators which have two missions. (i) They ensure the uniqueness of the triples among the peers in $\mathcal{I}$ implying the existence of a global total order $<_\mathcal{I}$. (ii) They guarantee that triples can always be inserted between two other triples even when the paths of the latter are identical.

EXAMPLE 6: Example 5 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 $allocDes$, 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 couples $\langle source, clock\rangle$. Similarly to the tree of paths, the set of all disambiguators can be summarized as a tree. The total order $<_{\mathcal{D}}$ of this tree is lexicographic. 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 $s_1 = s_2$, $c_1 < c_2$.

The unique site identifier of a peer and a local counter usually build the disambiguators. Their space complexity is upper-bounded by the size of the path. Furthermore, depending on the scenario, it can be drastically compressed. To illustrate, if any number of peers participate in an editing session without any concurrency, the $allocPath$ function is sufficient to provide unique identifiers. Therefore, there is no need for disambiguators at all.

THE GLOBAL TOTAL ORDER $(\mathcal{I}, <_{\mathcal{I}})$ is a simple merge of the aforementioned sets $\mathcal{P}$ and $\mathcal{D}$, and their respective total order $(\mathcal{P}, <_{\mathcal{P}})$ 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 $p.P(j) < q.P(j) \vee (p.P(j) = q.P(j) \wedge p.D(j) < q.D(j))$.

EXAMPLE 7: Fig. 4 shows the underlying tree of h-LSEQ. We can extract the two following triples: $\langle [29], W, [d^1_W]\rangle$ and $\langle [29], T, [d^1_T]\rangle$, with $d^1_{>} < d^1_T$. The comparison immediately stops with the dissimilar disambiguators. As a consequence, the element W precedes the element T. Considering another triple $\langle [29, 13], E, [d^1_E, d^2_E]\rangle$, where $d^1_W = d^1_E$, and $d^1_W < d^1_T$. It allows ordering the elements E and T. However, we still require differentiating the element W from the element E. In this case, since the element E has no additional path and disambiguator, the element E is automatically considered greater.

4.1 Space complexity

As stated in Section 3, we focus the space complexity analysis of h-LSEQ to random and monotonic editing behaviours to which we add a worst-case complexity analysis. Since the space complexity of disambiguators provided by $allocDes$ is upper bounded by the paths allocated by $allocPath$, the space complexity analysis of the path is generalized to the space complexity of the identifiers of h-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:

$$n = \sum_{i=0}^{k} 2^{\frac{i^2-i}{2}} \tag{2}$$

Therefore, after n insertions performed on the sequence, the number of concatenations is:

$$O(\sqrt{log\, n}) \tag{3}$$

Furthermore, a path is a sequence of numbers $[a_1.a_2 \ldots a_k]$. 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:

$$\sum_{i=1}^{k} b + i = kb + \frac{k(k+1)}{2} = O(k^2)\, bits \tag{4}$$

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

$$O(log\, n)\, bits \tag{5}$$

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:

$$n = 2^{k+1} - 1 \tag{6}$$

Therefore, the number of concatenations of the path is:

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

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

$$O((log\, n)^2)\, bits \tag{8}$$

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 in a path is:

$$O(n) \tag{9}$$

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:

$$O(n^2)\, bits \tag{10}$$

TABLE 1
Spatial complexity of *h*-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.

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

4.2 Discussion

As stated in Section 3, distributed collaborative editing often comprises monotonic editing behaviour or random editing behaviour or a composition of them. On the other side, the worst-case of *h*-LSEQ happens with a negligible probability. Theoretically, if we consider a uniform distribution of the positions of the n insert operations performed on the sequence, the worst-case editing behaviour happens with a probability of $\frac{1}{n!}$. However, the human editing behaviour cannot be simply modelled by a uniform distribution. Therefore, *h*-LSEQ randomly chooses over the two sub-allocation functions making the worst-case pattern difficult to predict.

To summarize, *h*-LSEQ sacrifices on the worst-case space complexity to improve the space complexity on the monotonic editing behaviour. The worst-case happens with a very low probability while other editing behaviours are common. Thus, the cost of the sacrifice is negligible while the reward is significant.

5 Experiments

This section is divided into two parts. (i) The first part describes the *h*-LSEQ-based CRDT by filling in the blanks left on causality tracking (ii) The second part concerns the results of experiments that have the following goals:

- To validate the space complexity of *h*-LSEQ and highlight the improvement over the state-of-the-art variable-size identifiers CRDTs.
- To show that the *h*-LSEQ-based CRDT proposed in this paper scales in terms of simultaneous peers involved in the editing session.
- To show that concurrency impacts on the size of identifiers. Also, the scenario without concurrency constitutes the upper-bound on the size of identifiers.

5.1 An *h*-LSEQ-based CRDT for sequences

As stated in Section 2, a CRDT is an abstract data type which provides strong eventual consistency upon the assumption of eventual delivery. The operations of CRDTs for sequences are commutative. However, there exists one exception: the insertion of an element and its deletion. To provide a consistent answer to this particular case, CRDTs for sequences require a form of causality tracking.

A CRDT for sets without ordering constraints has been introduced in [19]. It describes a vector structure called interval version vector which delivers operations in a non-blocking way. More specifically, the only blocking event is a delete received before its relative insert. Otherwise, the operations are immediately delivered.

Similarly to version vector, the interval version vector requires a vector with one entry per peer involved in the collaboration. However, contrarily to a version vector, each entry is an interval. Also, only two scalars are broadcast instead of the full vector. This difference considerably lowers the network overload while keeping an eventual local space complexity to 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.

Algorithm 1 Delivery protocol.

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

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

Algorithm 1 describes the delivery protocol running at each peer. Since different implementations are possible, the functions relative to the interval structure are voluntarily left aside. The function $alloc$ is an aggregation of $allocPath$ and $allocDes$ that creates the triples using a local site identifier and a local counter. 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

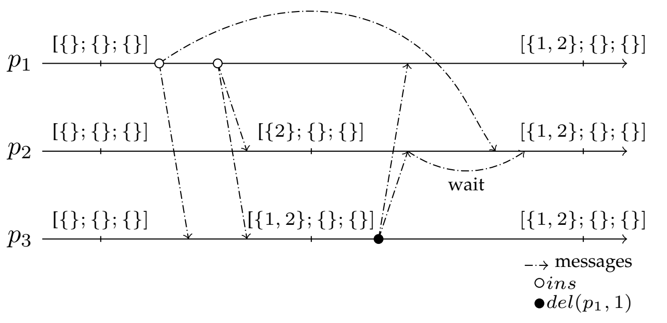

Fig. 5. Timeline of operations performed by 3 peers. For the sake of simplicity, sets represent the intervals stored in the local vectors. The ins operations concern any element while the del operation targets the first insertion of the peer p_1.

CRDT for sequences is not able to provide the idempotency property of CRDTs. Indeed, the del operations remove information from the data structure. The causality tracking implicitly keeps the summary of all operations.

EXAMPLE 8: A first peer generates two operations with the unique element e ($ins(e) \rightarrow del(e)$). When the $ins(e)$ operation arrives to a second peer, it is delivered. Similarly, when the $del(e)$ 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 ins 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 ins message, or receives the copy before performing the $del(e)$ operation, the replicas of the two peers may become divergent.

Line 28 shows that the only blocking condition of this causality tracking mechanism is located in the reception of the del messages. Indeed, the intervals representing the insertions performed by the targeted peer must contain the targeted entry of the del operation. Otherwise, a set buffer holds the del operation until the system obtains the notification that the entry has arrived. Thus, the ins operation is delivered and the del operation follows, removing the new element from the tree.

EXAMPLE 9: Fig. 5 depicts an example of causality tracking with all possible configurations. The 3 peers initialize 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.

5.2 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 scale of these experiments grows up to 450 peers simultaneously editing.

A gossip-based dissemination protocol performs message propagation. 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. The protocol is simple and is reliable with high probability. To make it entirely reliable, an additional anti-entropy protocol makes sure that all messages eventually arrive to all neighbours.

In all these experiments, 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. We voluntarily degenerated the initial base parameter, i.e., the number of children at the root of the exponential tree, in order to amplify the growth speed 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.

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

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

1. https://github.com/Chat-Wane/LSEQArray.git

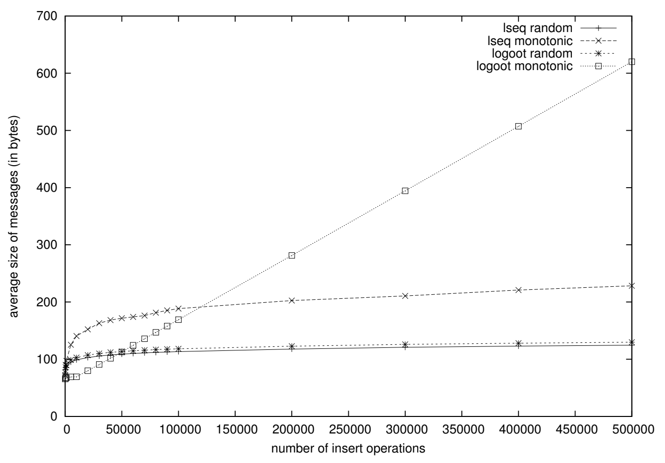

Fig. 6. Average size of the messages exchanged between two peers during repeated insertions of elements in the sequence. The measurements concern two editing behaviours (monotonic and random) with two CRDTs for sequences (Logoot and *h*-LSEQ-based). The replicated document grows up to half a million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

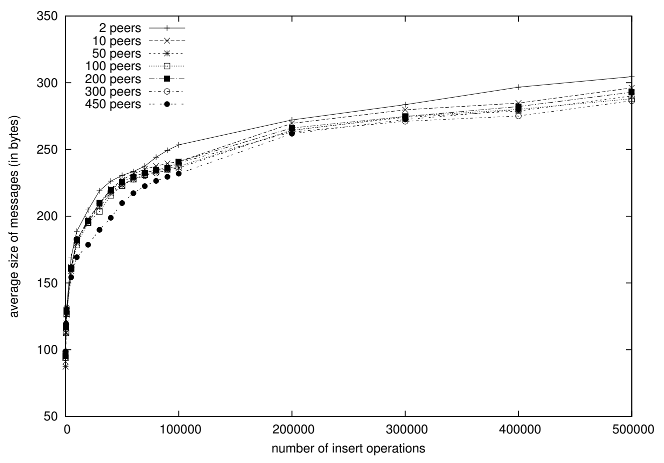

Fig. 7. Average size of messages sent by peers during a session of document editing. The number of peers varies from 2 to 450 collaboratively creating a document of 500 thousand characters in 50 minutes. The editing behaviour is monotonic.

editing behaviours are studied: monotonic and random. They use whether an *h*-LSEQ-based CRDT, or a representative of variable-size identifiers CRDTs for sequences named Logoot [20]. While *h*-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 *h*-LSEQ-based CRDT has a polylogarithmic shape proved in Section 4. Thus, even if Logoot starts with a smaller size messages, it quickly becomes less efficient than the *h*-LSEQ-based CRDT when the number of insertions increases.

REASONS: The random editing behaviour is slightly better for the *h*-LSEQ-based CRDT compared to Logoot because *h*-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, *h*-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 *h*-LSEQ-based CRDT for sequences scales in terms of number of peers. In other terms, the size of the network does not imply changes in the space complexity upper bound.

DESCRIPTION: A cluster of the Grid'5000 testbed hosts a varying number of peers. This latter 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. 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, the *h*-LSEQ-based CRDT for sequences 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 5.1 describes an *h*-LSEQ-based CRDT that requires only two scalars to track semantically related operations. Since these scalars are also required to build the identifiers, they are factorized within the latter. Consequently, the average size of messages only depends of the average size of the identifiers. Since the

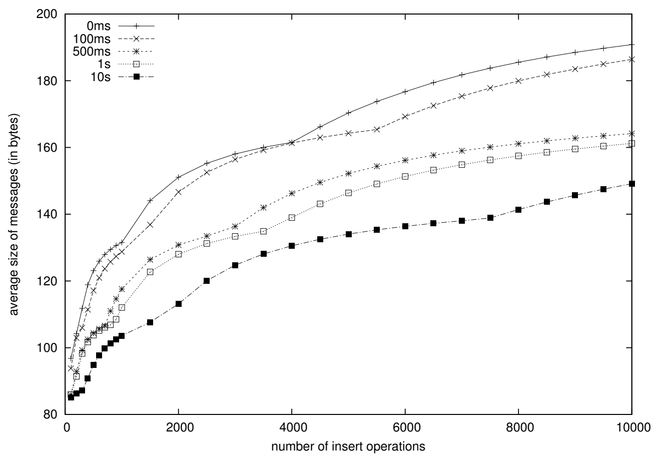

Fig. 8. Average size of messages sent by peers during sessions of document editing. Ten peers create documents of 10 thousand characters by repeatedly inserting at the end of the document at a rate of 3 operations per second. Five editing sessions with different network latency are studied.

space complexity of *h*-LSEQ is independent of the number of peers, so are the messages of the *h*-LSEQ-based CRDT. Still, the *h*-LSEQ-based CRDT 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, the resulting path in the identifiers 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: Show that concurrency and average size of identifiers are correlated and that executions without concurrency constitute an upper bound on the size of identifiers.

DESCRIPTION: A single machine emulates 10 peers. These latter 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 on the average size of messages follows the expectation from space complexity analysis in Section 4. Also, when the latency increases, the average size of messages decreases. The curve with the lowest latency corresponds to an almost sequential execution and exposes the heavier 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 tend to the new value. A finer grain of measurements in this experiment allows us to observe these changes.

5.3 Synthesis

This section detailed a complete *h*-LSEQ-based CRDT for sequences. The first experiment confirmed the space complexity analysis of Section 4. Therefore, using *h*-LSEQ, our CRDT 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 dependant operations does not impact the size of messages. As a consequence, *h*-LSEQ-based CRDT scales with respect to the number of peers. Finally, the third experiment confirmed the observations made in the second one: the concurrency rate and the size of identifiers are correlated, and the sequential execution gives the upper bound on the average size of messages.

6 RELATED WORK

Distributed collaborative editing tools can be divided in two approaches. (1) The Operational Transform (OT) approach [2] 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 $move$, $cut-paste$, ... However, the correctness analysis requires to carefully examine each couple of operations and their parameters. As a consequence, a few OT approaches are actually correct [21]. Furthermore, (i) the decentralized approaches [22] require a version vector to identify the generation context of the received operations and transform the arguments. Thus time and spatial complexity of the downstream part of optimistic replication do not scale in terms of number of operations or number of peers respectively. (ii) The centralized approaches [18] serialize the order of operations on a server. It alleviates the problem relative to the version vector. However, such topology implies a single point of failure and privacy issues. (2) The Conflict-free Replicated Data Type [4], [5] approaches share the computational cost between the local and remote part of the optimistic replication. While CRDTs significantly improve the time complexity compared to decentralized OT approaches, they hide the complexity in the memory usage.

CRDTs for sequences are divided in two classes: (1) The tombstone class of CRDTs [6], [7], [8], [9], [10], [11], [12], [13] marks the delete 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 operations. For instance, in Wikipedia documents subject to vandalism, delete operations are very common. Using this class of CRDTs, the pages could contain few lines and yet use a lot of storage. (2) The variable-size identifiers class of CRDTs [9], [14], [15] assigns an identifier to each element in the document. However these identifiers do not have a fixed-length but deleted elements are truly removed. As a consequence, the space complexity of identifiers only depends of insert operations and is very important.

In the literature, there were two kinds of allocation functions for the identifiers. Let's 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 randomizing 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 in average half the level of identifiers in a branch. (2) The observations made on occidental 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.

As Section 4 demonstrates, the allocation function h-LSEQ [16], [17] improves state-of-the-art allocation functions by lowering the space complexity of identifiers from linear to sub-linear in two over the three studied cases. Furthermore, most of editing behaviours, i.e., monotonic and random, corresponds to these favourable cases or a composition of them.

Using h-LSEQ, a CRDT for sequences still requires a form of causality. However, the causality tracking is still an issue in distributed network subject to churn [23]. The full causality tracking requires piggybacking a version vector with N entries with each message, N being the number of nodes ever involved in the network. Of course, it is impracticable in the context where peers can join and leave the network freely. Some other approaches [24] reuse entries of left nodes but require that leaving nodes notifies it clearly.

A version vector is the smallest structure to accurately characterize causality [25]. Nevertheless, there exist other trade-offs with more lightweight structures at the price of accuracy. In this paper, we made the choice of tracking the semantically related operations accurately. Thus, the resulting h-LSEQ-based CRDT has the space complexity of a version vector locally while not overwhelming the network with causality metadata.

7 Conclusion

In this paper, we refined the h-LSEQ allocation strategy by providing the proof of its space complexity. It has a logarithmic, polylogarithmic, and quadratic upper bound on its space complexity for, respectively, random insertions, monotonic insertions, and worst-case. Using this allocation strategy, we built a CRDT using an interval version vector to minimize the network load. As a consequence, it scales in terms of number of peers involved in the editing and number of insertions. We validated our CRDT under extreme conditions: degenerated parameters, high number of insertions, high number of peers, high latency. The experimentation confirmed the space complexity of h-LSEQ. It also confirmed our expectation on the scalability of the h-LSEQ-based CRDT. Such observations place CRDT-based distributed collaborative editors as a serious concurrent for current trending editors (such as Google Docs, SubEthaEdit, ...) without any service provider or any service limitation. In particular, using such editor alleviates economic intelligence issues brought by third-party hosts.

Future work concerns the release of a truly tree-based implementation. Indeed, the current version uses a data structure which is a linearization of the tree. As a consequence, each element is linked to its identifier without factorizing the common parts of the latter. Therefore, the memory usage is currently higher than the one suggested by the theoretical analysis of Section 4. The network load would remain unchanged.

Ultimately, we intend to create a worldwide event of massive collaborative editing using our h-LSEQ-based CRDT to stress out the system.

Acknowledgement

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] P. R. Johnson and R. H. Thomas, "Maintenance of duplicate databases," RFC 677, Internet Engineering Task Force, Jan. 1975. [Online]. Available: http://www.ietf.org/rfc/rfc677.txt

[2] Y. Saito and M. Shapiro, "Optimistic replication," *ACM Comput. Surv.*, vol. 37, no. 1, pp. 42–81, Mar. 2005. [Online]. Available: http://doi.acm.org/10.1145/1057977.1057980

[3] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and 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.

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

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

[6] M. Ahmed-Nacer, C.-L. Ignat, G. Oster, H.-G. Roh, and P. Urso, "Evaluating CRDTs for Real-time Document Editing," in *11th ACM Symposium on Document Engineering*, ACM, Ed., Mountain View, California, États-Unis, Sep. 2011, pp. 103–112. [Online]. Available: http://hal.inria.fr/inria-00629503

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

[8] G. Oster, P. Urso, P. Molli, and 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.

[9] N. Preguiça, J. M. Marqus, M. Shapiro, and 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.

[10] H.-G. Roh, M. Jeon, J.-S. Kim, and J. Lee, "Replicated abstract data types: Building blocks for collaborative applications," *Journal of Parallel and Distributed Computing*, vol. 71, no. 3, pp. 354–368, 2011.

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

[12] Q. Wu, C. Pu, and 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.

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

[14] L. André, S. Martin, G. Oster, and 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, Oct. 2013. [Online]. Available: http://hal.inria.fr/hal-00903813

[15] S. Weiss, P. Urso, and 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.

[16] B. Nédelec, P. Molli, A. Mostéfaoui, and E. Desmontils, "Concurrency effects over variable-size identifiers in distributed collaborative editing," in *DChanges*, ser. CEUR Workshop Proceedings, vol. 1008. CEUR-WS.org, Sep. 2013.

[17] B. Nédelec, P. Molli, A. Mostfaoui, and E. Desmontils, "LSEQ: an Adaptive Structure for Sequences in Distributed Collaborative Editing," in *13th ACM Symposium on Document Engineering*, ACM, Ed., Sep. 2013. [Online]. Available: http://doi.acm.org/10.1145/2494266.2494278

[18] D. A. Nichols, P. Curtis, M. Dixon, and 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.

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

[20] S. Weiss, P. Urso, and P. Molli, "Logoot-undo: Distributed collaborative editing system on p2p networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 21, no. 8, pp. 1162–1174, 2010.

[21] A. Imine, P. Molli, G. Oster, and 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*, ser. ECSCW'03. Norwell, MA, USA: Kluwer Academic Publishers, 2003, pp. 277–293. [Online]. Available: http://dl.acm.org/citation.cfm?id=1241889.1241904

[22] D. Sun and C. Sun, "Context-based operational transformation in distributed collaborative editing systems," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 20, no. 10, pp. 1454–1470, Oct 2009.

[23] R. Baldoni and M. Raynal, "Fundamentals of distributed computing: A practical tour of vector clock systems," *IEEE Distributed Systems Online*, vol. 3, no. 2, p. 12, 2002.

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

[25] B. Charron-Bost, "Concerning the size of logical clocks in distributed systems," *Inf. Process. Lett.*, vol. 39, no. 1, pp. 11–16, Jul. 1991. [Online]. Available: 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 PhD 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 sptember 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, transacionnal memory, wiki, etc.). How to specify and implement distributed data structures and then intergrate 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.