A scalable sequence encoding for collaborative editing

Brice Nédelec, Pascal Molli, Achour Mostefaoui

CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2016; **00**:1–20
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

A scalable sequence encoding for collaborative editing

Brice Nédelec*, Pascal Molli*, Achour Mostéfaoui*

LINA, 2 rue de la Houssinière,
BP92208, 44322 Nantes Cedex 03, France
first.last@univ-nantes.fr

SUMMARY

Distributed real-time editors made real-time editing easy for millions of users. However, main stream editors rely on Cloud services to mediate sessions raising privacy and scalability issues. Decentralized editors tackle privacy issues but scalability issues remains. We aim to build a decentralized editor that allows real-time editing anytime, anywhere, whatever is the number of participants. In this paper, we propose an approach based on a massively replicated sequence data structure that represents the shared document. We establish an original tradeoff on communication, time and space complexity to maintain this sequence over a network of browsers. We prove a sublinear upper bound on communication complexity while preserving an affordable time and space complexity. In order to validate this tradeoff, we built a full working editor and measured its performance on large scale experiments involving up till 600 participants. As expected, the results show a traffic increasing as $\mathcal{O}((\log I)^2 \ln R)$ where I is the number of insertions in the document, and R the number of participants. Copyright © 2016 John Wiley & Sons, Ltd.

KEY WORDS: real-time collaborative editing; optimistic replication; replicated sequences

1. INTRODUCTION

Google Docs made real-time editing in browsers easy for millions of users. However, Google mediates real-time editing sessions with central servers raising issues of privacy, censorship, and economic intelligence. Decentralized real-time editors [20, 27, 29] do not require intermediate servers and by the same settle privacy issues. However, scalability issues in terms of the number of participants, remain. Despite that small groups currently constitute the main range of users, events such as massive open online courses (MOOC), TV shows, conferences gather larger groups [6]. Google Docs supports large groups but only the first 50 users can edit; next users have their rights limited to document reading, not even in real-time. We think that real-time editors should allow users to edit at anytime and anywhere, regardless of the number of participants. Even if only a small subset among millions of participants are writing, all participants of the editing session should be able to read and write in real-time whenever they want. In 2013, Coursera gathered 41000 participants for a MOOC entitled "Fundamentals of Online Education: Planning and Application". However, the course relied on Google tools that only allow a limited number of users to edit simultaneously. The result was a "disaster" according to journalists [19, 26]. This example clearly demonstrates that collaborative editors should be designed for large groups. [30] reports a similar issue in a context of massively distributed authorship.

In order to be used in a crowdsourcing context such as MOOC, a scalable real-time editor should support large groups i.e. it should support thousands of users editing or reading in real-time documents containing millions of characters. Addressing scalability requires finding a good

*Correspondence to: LINA, 2 rue de la Houssinière, BP92208, 44322 Nantes Cedex 03, FRANCE, E-mail: first.last@univ-nantes.fr

tradeoff between communication, space and time complexities. Among others, achieving a sublinear communication complexity compared to the number of participants is crucial for supporting large groups.

We consider variable-size conflict-free replicated data types [24] for sequences such as Logoot [34] and Treedoc [21] as the most promising approaches to handle large groups. However, as the size of unique and immutable allocated identifiers associated to each character of the sequence grow linearly, their performance decreases. It eventually affects the responsiveness of the editor. A sublinear upper bound on the size of identifiers solves the problem and enables real-time editing in crowdsourcing context. In previous works [16, 17], we proposed an allocation strategy and observed a sublinear upper-bound empirically. However, there is no formal proof of such property.

The contributions of this paper are threefold:

- Compared to previous works [16, 17], we prove the upper bounds on space and time complexities of LSEQ and the conditions upon which they apply. The size of the identifiers grows up to $\mathcal{O}((\log I)^2)$ where I is the number of insertions in the document. This result opens the way for building decentralized and scalable real-time editors.
- To measure the overall performance of a real working system, we built CRATE: a real-time decentralized editor running in web browsers [15]. Compared to Google Docs, CRATE enables real-time editing regardless of the number of participants. To better preserve privacy of users, CRATE ensures that shared documents only exist within browsers participating to the editing session. In this paper, we describe the complete architecture of CRATE and review the contribution of each component to the scalability of the system.
- To validate our complexity analysis, we provide experiments characterizing LSEQ's behavior and we compare it to the state-of-the-art approaches [21, 33]. Results show that LSEQ achieves an original tradeoff between time and space complexities. In the Grid'5000 testbed, we launched CRATE editing sessions involving up to 600 connected web browsers. The generated traffic inherits from the scalability of both LSEQ and messages dissemination [18]. The resulting documents reach millions of characters. It validates the scalability of a full working editor built on top of LSEQ.

The remainder of this paper is organized as follows: Section 2 reviews the related work. Section 3 describes the necessary background to understand the replicated data types for sequences. Section 4 describes the principle and functioning of LSEQ. In particular, it provides its complexity. Section 5 describes the architecture of decentralized real-time editors. The experiments of Section 6 highlight its scalability while validating the complexity analysis. Section 7 concludes the paper.

2. RELATED WORK

Real-time distributed collaborative editors consider multiple participants, each hosting a copy – or replica – of a shared sequence of characters. The participants can update their copy by inserting or deleting characters at any time. Then, the editor eventually delivers the change to all participants. Finally, each editor integrates the received operations [23].

According to [28], collaborative editing requires to ensure the CCI properties:

- Convergence: replicas integrating a same set of operations converge to an equivalent state [5, 25], i.e., users see an identical document;
- Causality: if an operation precedes [11] another operation, the integration of the former always precedes the integration of the later;
- Intention preservation: effects observed at generation time must be re-observed at integration time regardless of concurrent operations.

Four complexities characterize collaborative editors:

- Generation time: complexity to generate locally an operation.
- Integration time: complexity to execute remotely an operation.
- Space complexity: complexity to store a local copy of the shared sequence.
- Communication complexity: complexity of messages transiting the network.

DOI: 10.1002/cpe

Solving scalability issues requires finding an original balance between communication, space and time complexities. The challenge consists in providing a sublinear upper bound on communication complexity without making other complexities impracticable.

Operational transformation (OT) constitutes the mainstream of both centralized and decentralized real-time editors. OT ensures the CCI properties [28]. At generation, the processing time of operations is constant. At integration, OT transforms the received operations according to concurrent ones which were generated on the same state. An integration algorithm such as COT [29] or SOCT2 [31] along with transformation functions (ensuring transformation properties) guarantee a consistent model. The integration time depends on concurrency and differs among the algorithms. For instance, COT's integration time complexity is exponential. It can be reduced to linear at the expense of space complexity. SOCT2's integration time complexity is quadratic in terms of number of concurrent operations. Regardless of the tradeoff, OT's integration algorithms rely on concurrency detection which costs, at least, a state vector [7] the size of which grows linearly compared to the number of members W who ever participated in the authoring. Since each message carries such a vector, decentralized OT approaches are not practicable in large and dynamic groups where people frequently join and leave the editing session.

Conflict-free replicated data types (CRDTs) for sequences [24, 25] constitute the latter approaches which solve concurrent cases by providing commutative, associative, and idempotent operations. The causality property of CCI that precludes editors to reach large scale can be relaxed. Contrarily to OT, CRDTs only require to track semantically related operations. For instance, the removal of a particular element must follow its insertion.

Intentions for sequences are defined with the help of a dense order [34]. A sequence is a set of elements paired with a dense order, i.e., elements are ordered and inserting between two elements is always possible.

- Inserting the element e at position i in the sequence becomes inserting the element e between the element at position $i-1$ and the element at position i;
- Deletion an element at position i in the sequence removes the element from the set without impairing the order of other elements.

CRDTs implements this dense total order by associating a unique and immutable identifier to each element of the sequence.

CRDTs propose an interesting tradeoff, for they can balance complexities depending on the type of structure that represents the document. In particular, increasing the generation time of operations to decrease the integration time is profitable since an operation generated once is re-executed many times. Nevertheless, identifiers consume space and impact the communication complexity.

Tombstone-based CRDTs [1, 9, 20, 22, 32, 35, 36] such as WOOT [20] associate a constant size identifier $\mathcal{O}(1)$ to each element but removals of elements only hide them to users. Therefore, removed elements keep consuming space leading to a monotonically growing replica, hence $\Theta(I)$ where I is the number of insertions performed on the document. Destroying removed elements requires running consensus algorithms to determine if everyone received the removal and agrees on definitely throw out the element. Such algorithms are prohibitively costly and do not scale in terms of number of members, especially in networks subject to churn [13].

Variable-size identifiers CRDTs [17, 21, 33] such as Logoot [33] truly destroy elements targeted by removals. Since removals truly erase information from the structure, these approaches require a local state vector compacting their history, hence an additional $\mathcal{O}(W)$ on space complexity. It is worth noting that, contrarily to OT, the vector stays local. Variable-size identifiers approaches allocate identifiers the size of which is determined at generation. The allocation function becomes crucial to maintain identifiers under acceptable bounds. Unfortunately, they depend on the insert position of elements. For instance, writing the sequence `QWERTY` left-to-right allocates the identifiers [1] to `Q`, [2] to `W`, [3] to `E`, ..., [6] to `Y`. Continuing with such insertion pattern will slowly deplete all available identifiers of that size leading to larger identifiers such as [65535.1], [65535.2], etc. With an identical allocation strategy, writing the same sequence `QWERTY` right-to-left allocates the identifiers [1] to `Y`, [1.1] to `T`, [1.1.1] to `R`, ... We observe a quick growth of identifiers depending on the editing behavior. In both cases, the growth is linear compared to the number of

DOI: 10.1002/cpe

insertions I in the sequence, i.e. $\mathcal{O}(I)$. Both Treedoc [21] and Logoot [33, 34] suffer from such growth of identifiers which negatively impacts the generated traffic. To provide small identifiers, these approaches eventually require balancing the structure, i.e., relocating identifiers. This requires a global agreement which is akin to run a distributed consensus protocol [37]. Unfortunately, these protocols do not scale.
The algorithm LSEQ [17] aims to avoid such consensus by sublinearly upper bounding the space complexity of variable-size identifiers. [17] conjectured a polylogarithmic progression of its identifiers size $\mathcal{O}((\log I)^2)$ compared to the number of insertions in the document. This paper proves the complexity upper bounds and states the conditions under which it applies.

LogootSplit [4] is a CRDT for sequences that allocates an identifier to each string; splitting strings if needed. Therefore, it allocates fewer identifiers. At worst, it allocates as many identifiers as Logoot, Treedoc or LSEQ. It constitutes an orthogonal improvement that applies to any of these variable-size allocation functions. Hence, LSEQSplit would exhibit the improvements of both "Split" and LSEQ.

3. PRELIMINARIES

Sequence replicated data structures (*sequence* for short) are the closest structures that can implement a shared document.

Definition 1 (Document)
A document is a series of k characters $d(H) = \alpha_1.\alpha_2 \dots \alpha_k$ produced by a series of operations H.

Definition 2 (Replicated sequence)
Let $d(H)$ be the document produced by the series of operations H. A replicated sequence of $d(H)$ is a structure $r(H)$ with a projection π such that $\pi(r(H)) \rightarrow d(H)$.

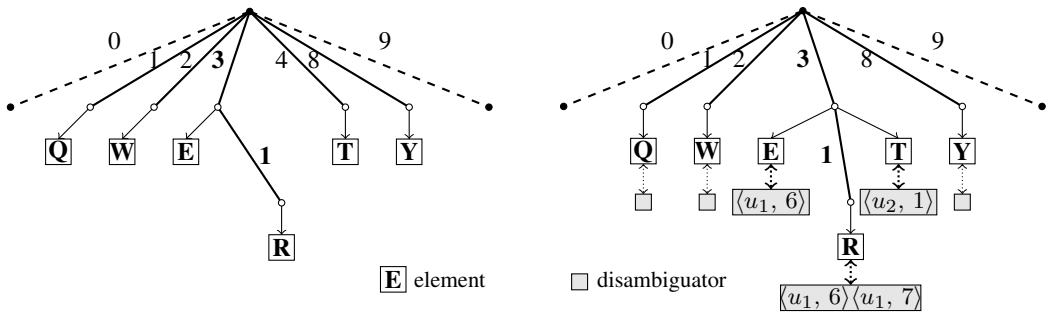
We focus on replicated sequences [24, 25] that provide two commutative operations: INSERT and DELETE. Operations can be integrated in any order as long as the deletion of an element follows its insertion. Assuming that the replicas integrated the same set of operations, they converge to an identical state, i.e., users see the same document [25].
Each element of the sequence is associated to a unique and immutable identifier. Using a dense total order on identifiers, we project the set of pairs $\langle identifier, element \rangle$ to a sequence of elements. Since elements are not explicitly assigned to an offset, a function LOOKUP retrieves the identifier at a targeted position in the sequence, and converse.

Each time a user types a character, the INSERT operation generates an identifier for the new element. For instance, let us consider a sequence `QWTY` with the unique, immutable, and totally ordered integer identifiers 1, 2, 4, and 8 respectively. A collaborator inserts `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`, for the identifiers 3 and 4 are contiguous. The space of identifiers must be enlarged to handle the new insertion. If we consider identifiers as decimal numbers, we can associate 3.1 to the character `R`. Since $3 < 3.1 < 4$, the sequence of characters is, as expected, `QWERTY`. Inserting a new character between `E` and `R` would result in another space extension : $3.0.X$, where X is an integer.

Such growing identifiers are called variable-size identifiers. The main objective is to keep the growth of the size of the identifiers under a sublinear boundary.

Encoding a position in the sequence. Each INSERT operation generates an identifier that encodes the position of the new character in the sequence using a dense order. Concatenations of comparable elements constitute a mean to represent such dense order. The union of these identifiers creates a tree. Its nodes store the elements of the sequence and its edges are labeled such that a path from the root to a node containing an element represents the main part of the latter's identifier. For instance, the character `R` in the previous example is accessible following the path composed of the edges labeled 3 then 1. This represents the *path* of an identifier.

DOI: 10.1002/cpe



(a) The tree representing the sequence is built by the union of identifiers and mainly uses paths to order its characters.

(b) The tree uses disambiguators to maintain an equivalent state, even in presence of concurrent insertions resulting in same paths. For simplicity sake, only the disambiguators of E, R, and T are displayed.

Figure 1. Examples of 10-ary trees containing the sequence of characters QWERTY.

Definition 3 (Path)
A path is a series of integers noted $[\ell_1.\ell_2 \dots \ell_e]$ chosen in a set $\mathcal{P}$ paired with a dense order $(\mathcal{P}, <_{\mathcal{P}})$. Each integer ℓ_i composing the path is chosen in a set $\mathbb{N}_{<X_i}$. The binary representation of a path of size e requires $\sum_{i=1}^{e} \lceil \log_2 X_i \rceil$ bits.

Figure 1a shows the underlying 10-ary tree representing a sequence. As 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 ⟨[2], W⟩ and ⟨[4], T⟩ results in the following pair: ⟨[3], E⟩. Then, inserting R between E and T forces to start a new level, for there is no room at the first level of the tree for any paths between these bounds. In this example, the resulting path is [3.1]. Using the total order of paths, we retrieve the sequence QWERTY.

Disambiguation of concurrent cases. The order among paths $(\mathcal{P}, <_{\mathcal{P}})$ is a total order when a single collaborator edits. However, it becomes a partial order when the editing session involves several collaborators and concurrent operations exist. For instance, two collaborators inserting a character at a same position in the sequence at a same time may end up with the same allocated path. In such case the order of characters is not strictly defined and may break the convergence property. We need a disambiguator in each identifier to preserve their uniqueness. This guarantees a total order among identifiers, hence convergent replicas.

Definition 4 (Disambiguator)
A disambiguator is a globally unique marker chosen in a set $\mathcal{D}$ paired with a total order $(\mathcal{D}, <_{\mathcal{D}})$. Disambiguators usually comprise unique site identifiers along with Lamport clocks [11].

Figure 1b depicts a tree containing 6 elements with only 5 distinct paths. The collaborator u_1 inserts QW. Then, the collaborators u_1 and u_2 concurrently insert E and T respectively. This happens to generate an identical path: [3]. To solve any ambiguity about the order of characters, each identifier includes a disambiguator. Let a disambiguator be a series of pairs of unique site identifiers and counters: $[\langle s_1, c_1\rangle.\langle s_2, c_2\rangle \dots \langle s_e, c_e\rangle]$ where e is the size of the path. The disambiguator $[\langle u_1, 6\rangle]$ is associated with E – meaning u_1 already performed 3 operations before inserting Q, W, and E; the disambiguator $[\langle u_2, 1\rangle]$ is associated with T – meaning this is the first operation of u_2. The disambiguator of the character R inserted between E and T is $[\langle u_1, 6\rangle.\langle u_1, 7\rangle]$.

Allocating variable-size identifiers. Unique and immutable identifiers comprise both a path and a disambiguator in order to preserve a dense total order, even in presence of concurrent operations.

Definition 5 (Variable-size identifier)
A variable-size identifier is a triple $\langle p, \alpha, d\rangle$ chosen in $\mathcal{I}$ where p is a path chosen in $\mathcal{P}$, α is an element chosen in an alphabet $\mathcal{A}$, and d is a disambiguator chosen in $\mathcal{D}$. The set $\mathcal{I}$ is paired with a dense total order $(\mathcal{I}, <_{\mathcal{I}})$.

DOI: 10.1002/cpe

In Figure 1b, the dense total order is also lexicographic: at each level we compare the paths, then the site identifiers, then the counters. It is defined as :

$$\begin{aligned} t_i < t_j &\iff (p_i < p_j) \vee ((p_i = p_j) \wedge (s_i < s_j)) \vee ((p_i = p_j) \wedge (s_i = s_j) \wedge (c_i < c_j)) \\ t_i = t_j &\iff \neg(t_i < t_j) \wedge \neg(t_j < t_i) \\ id_i <_{\mathcal{I}} id_j &\iff \exists(m > 0)(\forall n < m),\ (t_n^i = t_n^j) \wedge (t_m^i < t_m^j) \end{aligned}$$

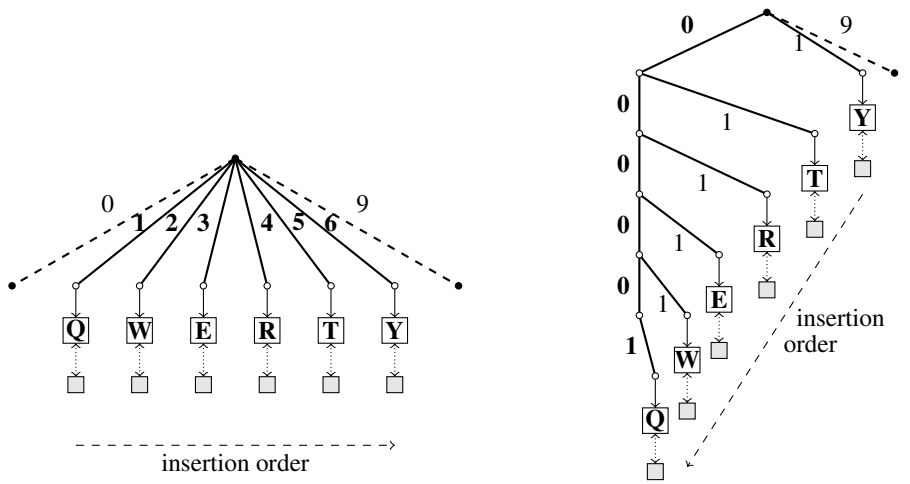
In this example, the paths order most of characters. Nevertheless, E precedes T because $p_1^{\mathrm{E}} = p_1^{\mathrm{T}}$ and $s_1^{\mathrm{E}} < s_1^{\mathrm{T}}$. Then, u_1 inserts Y at the end of the sequence. Finally, u_1 inserts R between E and T. Since E and T have an identical path, there is not enough room for new insertions at this level. The allocation function chooses a path $[3.X]$ where $0 < X < 10$. By copying the disambiguator of E at the first level, it ensures that the new identifier will follow E and precede T.

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

```
1:  INITIALLY:
2:      T ← ∅;                                          ▷ structure of the CRDT for sequences
3:
4:  LOCAL UPDATE:
5:      on insert (previous ∈ I, α ∈ A, next ∈ I):
6:          let ⟨p, q⟩ ← CONVERT2PATH(previous, next);
7:          let newPath ← ALLOCPATH(p, q);
8:          let newDis ← ALLOCDIS(p, newPath, q);
9:          BROADCAST('insert', ⟨newPath, α, newDis⟩);
10:     on delete (i ∈ I):
11:         BROADCAST('delete', i);
12:
13: RECEIVED UPDATE:
14:     on insert (i ∈ I):                              ▷ once per distinct triple in I
15:         T ← T ∪ i;
16:     on delete (i ∈ I):                              ▷ after the remote insert(i) is done
17:         T ← T \ i;
```

Algorithm 1 shows the general outlines of these sequences. It divides the operations into the local and remote parts of the optimistic replication scheme. When a user types (resp. removes) a character, the editor immediately executes the local part of the INSERT (resp. DELETE) operation. The editor executes the remote part of operations as soon as it receives it and that its precondition is checked.
Algorithm 1's core lies in the local part of the insertion where it generates a path and a disambiguator. Firstly, to simplify readability in the rest of this paper, CONVERT2PATH immediately gets rid of the disambiguator contained in each identifier to keep paths only. It translates the identifiers into the closest paths that maintain their order using only the order among paths, i.e., $(\mathcal{I}, <_{\mathcal{I}}) \rightarrow (\mathcal{P}, <_{\mathcal{P}})$. For instance, considering the identifiers of Figure 1b $id_{\mathrm{R}} = \langle [3.1], \mathrm{R}, [\langle u_1, 6\rangle.\langle u_1, 7\rangle]\rangle$ and $id_{\mathrm{T}} = \langle [3], \mathrm{T}, [\langle u_2, 1\rangle]\rangle$, comparing these identifiers results in $id_{\mathrm{R}} <_{\mathcal{I}} id_{\mathrm{T}}$; but using only paths results in $id_{\mathrm{T}} <_{\mathcal{P}} id_{\mathrm{R}}$. Implicitly, id_{T} is the path $[4]$, for its first triple t_1^{T} is greater than the id_{R}'s one; and any path allocated between $[3.1]$ and $[4]$ builds an identifier between id_{R} and id_{T}.
Secondly, ALLOCPATH allocates a new path between these bounds. It should choose among the paths with the smallest size for performance sake. For instance, between $[3.1]$ and $[4]$, the available paths are $[3.2]$, $[3.3]$, ..., $[3.9]$. ALLOCPATH chooses among these.
Thirdly, ALLOCDIS generates the disambiguator to associate with the path in order to guarantee that the new identifier consistently fall between the adjacent identifiers that served its allocation following the dense total order among identifiers. For instance, considering that ALLOCPATH chose the path $[3.2]$, ALLOCDIS copies the id_{R}'s first pair of the disambiguator $\langle u_1, 6\rangle$ to which it concatenates its own unique marker: $\langle u_3, 1\rangle$. The resulting identifier is $id_e = \langle [3.2], e, [\langle u_1, 6\rangle.\langle u_3, 1\rangle]\rangle$. As expected, we obtain $id_{\mathrm{R}} <_{\mathcal{I}} id_e <_{\mathcal{I}} id_{\mathrm{T}}$.
Finally, the editor broadcasts id_e to all editors. Minimizing the size of id_e is important, for it directly impacts the generated traffic.

DOI: 10.1002/cpe



(a) Optimal case. (b) Worst case.

Figure 2. Two trees filled with the resulting identifiers of two different permutations resulting in an identical sequence QWERTY. The function ALLOCPATH allocates the leftmost branch in the tree. All paths of the optimal case have a length of 1 while the tree of the worst case grows up to a depth of 6.

Choosing the smallest path. The most critical part of sequences with variable-size identifiers consists in creating the paths. The function that allocates identifiers must provide the smallest possible paths without impairing future allocations. As shown in Algorithm 1 at Lines 9 and 11, each identifier is broadcast to all editors. As consequence, their size directly impacts the communication complexity of the editor.

As illustrated in Figure 2, the allocation of paths without an *a priori* knowledge of the final sequence is a non-trivial problem. Suppose that ALLOCPATH allocates the leftmost branch available at the lowest depth possible. Suppose two insertion orders resulting in an identical sequence of characters QWERTY. In the first case, we insert Q at position 0, followed by W at position 1, followed by E at position 2, etc. In the second case, the sequence starts empty and we insert Y at position 0. Then, we insert T. However, as the intended word is QWERTY, we insert T before Y. Thus, we insert T at position 0 shifting Y to position 1, etc.

In Figure 2a, the insertion order exactly follows the expectations of the allocation function. The depth of the tree never grows. The execution of operations remain efficient.

In Figure 2b the insertion order goes against the expectations of the allocation function. The depth of the tree increase at each insertion. Indeed, as an element gets the smallest value at its level, there is no room for a new element at the same level, hence the creation of a new level. The depth of the tree grows very fast decreasing the efficiency of operations.

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 corpora [21, 33]. However, there exist human edited documents that do not correspond to this kind of editing [17]. The problem is the following:

Problem statement 1 (Size of allocated identifiers)
Let $s(I) = id_1.id_2 \dots id_I$ be a sequence of identifiers, and $s(I+1) = s(I) \cup \text{INSERT}(p, \_, n)$, with $p, q \in s(I)$ and $p <_{\mathcal{I}} q$. Let $|s(I)|$ be the size of the binary representation of the sequence. The function INSERT must allocate identifiers such that: $|s(I+1)| - |s(I)| < \mathcal{O}(I)$

Large scale collaborative editors need an allocation function that provides identifiers with a sublinear space complexity compared to the number of insertions whatever is the editing sequence that produced the document. Such allocation function would avoid the need for consensus algorithm [13] and would make CRDT-based editors a practicable alternative to the current mainstream editors.

DOI: 10.1002/cpe

4. LSEQ: A POLYLOGARITHMIC PATH ALLOCATOR

LSEQ (polyLogarithmic SEQuence) is the name of the proposed allocation function. Our prior works [16, 17] empirically showed that LSEQ allocates identifiers with a sublinear upper bound on space complexity. However, we did not provide any complexity analysis to support these observations.

This section starts by describing the allocation strategy. Then, it provides the proof of the polylogarithmic growth of LSEQ's identifiers and states the conditions upon which this applies. The complexity analysis also includes the space complexity of a replicated structure, and the time complexity of operations provided by such structure.
As for the communication complexity, the next section describes an LSEQ-based editor that benefits from these identifiers.

4.1. Allocation of paths

When a user types a character, the editor executes the local part of the INSERT operation (see Line 5 of Algorithm 1). This function allocates an identifier comprising a path, the element, and a disambiguator. The most important choice concerns the path (see Line 7 of Algorithm 1). The function ALLOCPATH chooses the path that encodes the position of the new element regarding its adjacent elements in the sequence using a dense space. For the sake of performance, it aims to keep the paths small.

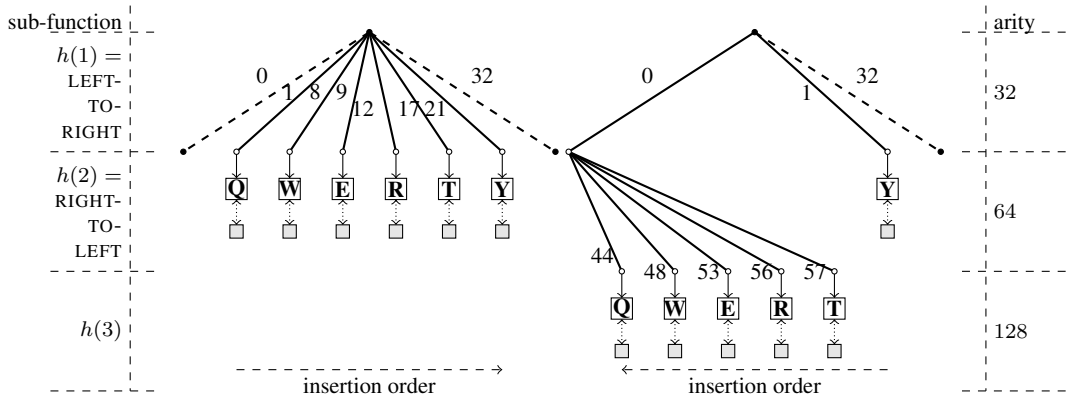
Algorithm 2 Allocation of paths

```
1:  let boundary ← 10;                                              ▷ Any constant
2:  let h : ℕ → (𝒫 × 𝒫 → 𝒫);                                       ▷ get sub-allocation function

3:  function ALLOCPATH(p, q ∈ 𝒫) → 𝒫
4:      let ⟨depth, _⟩ ← GETDEPTHINTERVAL(p, q);
5:      return h(depth)(p, q);                  ▷ Delegates the call to a sub-allocation function
6:  end function

7:  function LEFT-TO-RIGHT(p, q ∈ 𝒫) → 𝒫
8:      let ⟨depth, interval⟩ ← GETDEPTHINTERVAL(p, q);          ▷ #1 Get the depth of the new path
9:      let step ← min(boundary, interval);                  ▷ #2 Maximal space between two identifiers
10:     return SUBPATH(p, depth) + rand(0, step);                          ▷ #3 Create the new path
11: end function

12: function RIGHT-TO-LEFT(p, q ∈ 𝒫) → 𝒫
13:     let ⟨depth, interval⟩ ← GETDEPTHINTERVAL(p, q);                                ▷ #1
14:     let step ← min(boundary, interval);                                            ▷ #2
15:     return SUBPATH(q, depth) − rand(0, step);                                      ▷ #3
16: end function

17: function GETDEPTHINTERVAL(p, q ∈ 𝒫) → ℕ × ℕ      ▷ Which level has enough space for 1 path
18:     let depth ← 0; interval ← 0;
19:     while (interval < 2) do
20:         depth ← depth + 1;
21:         interval ← SUBPATH(q, depth) − SUBPATH(p, depth);
22:     end while
23:     return ⟨depth, interval⟩;
24: end function
```

Algorithm 2 shows the instructions of LSEQ that implements ALLOCPATH, and Figure 3 shows the tree filled with the identifiers generated by this algorithm on the scenarios presented in Figure 2. Figure 3a describes the left-to-right insertions of characters resulting in `QWERTY` i.e. [(`Q`, 0), (`W`, 1), ...]. Figure 3b describes the right-to-left insertions of characters resulting in `QWERTY` i.e. [(`Y`, 0), (`T`, 0), ...].

As shown in Figure 3, there exists two major differences between LSEQ and the state-of-the-art [21, 33]:

DOI: 10.1002/cpe



(a) Left-to-right editing behavior. (b) Right-to-left editing behavior.

Figure 3. Example of LSEQ's exponential trees with two antagonist editing behaviors to create the sequence of characters QWERTY. Contrarily to the example of Figure 2, the depth of trees does not grow linearly.

- The set of possible paths in LSEQ can be represented as an exponential tree [2, 3] instead of a tree of constant arity. An exponential tree allows each node to have twice as many children as its parent. For instance, in Figure 3, the root of the tree can have up to 32 children, each of these children can have up to 64 children, etc. This allows to give more space where it is required.
- Each level of the tree has a different strategy to allocate paths. We can see on Figure 3a that level $h(1)$ allocates paths from left to right. The allocated paths increase from 1 to 21. We also see on Figure 3b that level $h(2)$ allocates from right to left. The allocated paths decrease from 57 down to 44.

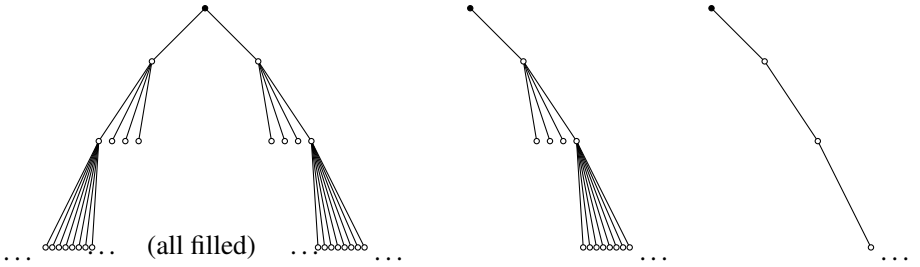
If the strategy for a level suits the insertion order, then the allocation is efficient. Otherwise, the level is sacrificed (e.g. the first level in Figure 3b) and the next level is efficiently filled. Thanks to the exponential growth of paths over levels, LSEQ compensates the sacrificed levels. Combining an exponential tree with different allocation strategies solves our problem statement.

Algorithm 2 firstly processes the size of the new path by progressively exploring the available paths between the adjacent paths. For instance, in Figure 3b, the character T needs a path between the paths [0] and [1]. There is not enough room for a new path between [0] and [1], but there are 63 available paths between [0.0] and [1.0]. Thus, the new path will comprise 2 concatenations: $[0.X]$ where X is yet to determine.

Secondly, a hash function h delegates the choice of path to a sub-allocation function. The hash function returns a result depending on the size of the new path. For instance, in Figure 3, the sub-allocation function of paths of size 1 is LEFT-TO-RIGHT, the sub-allocation function of paths of size 2 is RIGHT-TO-LEFT, etc. The results of this hash function must be identical regardless of the editor [16]. For instance, all editors of an editing session use the sub-allocation function LEFT-TO-RIGHT to choose the paths of size 1. In addition, the choices between the sub-allocation functions must follow a uniform distribution, for we do not know the future editing behaviors, and we do not want to favor any.

Thirdly, LSEQ uses one of its two sub-allocation functions. One is designed to handle left-to-right editing (Line 7), i.e., repeated insertions at the right of the newest inserted element (see Figures 2a and 3a); while the other is designed to handle right-to-left editing (Line 12), i.e., repeated insertions in front of the newest inserted element (see Figures 2b and 3b). To achieve their design, they leave more available paths at the right (resp. at the left) of the new path for the future insertions assumed at the right (resp. at the left) of the newest element. Each sub-allocation function has 3 instructions. It starts by processing the number of available paths between the adjacent paths. For instance, there are 63 available paths between [0.0] and [1.0]. Then, it shrinks the range of allocation using a $boundary$ value. Without such variable, each new allocation would consume half the available space in average making it inefficient to handle left-to-right or right-to-left editing. With this variable set to 10, the

DOI: 10.1002/cpe



(a) LSEQ tree filled using random editing (b) LSEQ tree filled using monotonic editing. (c) Worst case LSEQ tree.

Figure 4. LSEQ tree filled with the studied editing behaviors and in the worst case.

function only considers 10 among the 63 available paths. Then, depending on the design of the sub-allocation function, it chooses among the paths in the processed range starting from one of the adjacent paths. For instance, in Figure 3b, LEFT-TO-RIGHT chooses the path of Y among the paths [1], [2], ..., [10]; RIGHT-TO-LEFT chooses the path of T among the paths [0.54] [0.55], ..., [0.63]. Finally, the sub-allocation function uses SUBPATH to truncate or prolong the path in argument to reach the desired size. In Figure 3b, when inserting T, RIGHT-TO-LEFT starts from the next path [1] prolonged by SUBPATH to [1.0] and chooses a random path among the 10 preceding paths. The randomness aims to leave a small gap between paths – to handle users' minor corrections – even in presence of concurrent insertions. The resulting path of T is $[1.0] - 7 = [0.64] - 7 = [0.57]$.

In Figures 3a and 3b, the exponential tree of LSEQ starts with a maximum arity 2^5 and doubles it at each level. Also, it uses the left-to-right and right-to-left sub-allocation functions at the first and the second level of the tree respectively. Since the first level of the tree uses the function designed for left-to-right editing, the scenario involving the left-to-right editing sequence results in a tree of depth 1. On the other hand – and contrarily to the allocation function presented in Figure 2 – the antagonist scenario only leads to a tree of depth 2. Indeed, LSEQ quickly reaches a level of the tree where the sub-allocation function is designed to handle the right-to-left editing behavior.

4.2. Complexity analysis

In text editing, most of the editing behaviors can be empirically summarized as a composition of two basic editing behaviors:

- The random editing behavior where the author inserts new elements at what appears random positions in the sequence. For instance, this behavior mostly arises when syntactic corrections are performed, e.g. the author writes QWETY and realizes that the R is missing. She adds the missing character in a second time.
- The monotonic editing behavior 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 character Q to the last letter of the word Y. On the opposite, there exist documents mainly edited at the beginning (e.g. `http://en.wikipedia.org/wiki/Template_talk:Did_you_know` [17]). This behavior characterizes both left-to-right and right-to-left editing.

As a consequence, we focus on the space complexity analysis of LSEQ to random and monotonic editing behaviors. It is worth noting that monotonic editing behavior represents an unfavorable case since it tends to unbalance the underlying tree storing the replicated sequence. As soon as the participants start to edit at different positions – which is closer from a real-life use case – the tree becomes more balanced. The analysis does not include an average case, for it requires knowing the average distribution of the position of edits performed by humans which is obviously complex.

The complexity analysis also includes a worst case analysis. In the worst case, each new identifier is bigger than the previous one. To produce such worst case in LSEQ, the insertions must saturate the small gap left between identifiers (see Lines 10 and 15 of Algorithm 2). The editing behavior that produces such worst case with LSEQ is the following: (1) The user types A followed by B. The

DOI: 10.1002/cpe

Table I. Upper bounds on space complexity of LSEQ, Logoot and Treedoc. Where I is the number of insertions performed on the replicated sequence.

EDITING BEHAVIOR	LSEQ SPACE		LOGOOT / TREEDOC SPACE	
	IDENTIFIER	SEQUENCE	IDENTIFIER	SEQUENCE
Random editing	$\mathcal{O}(\log I)$	$\mathcal{O}(I \log I)$	$\mathcal{O}(\log I)$	$\mathcal{O}(I)$
Monotonic editing	$\mathcal{O}((\log I)^2)$	$\mathcal{O}(I \log I)$	$\mathcal{O}(I)$	$\mathcal{O}(I)$
Worst case	$\mathcal{O}(I^2)$	$\mathcal{O}(I^2)$	$\mathcal{O}(I)$	$\mathcal{O}(I)$

gap left between `A` and `B` corresponds to the $boundary$ (see Line 1 of Algorithm 2); (2) The user types $boundary$ characters between the two lastly inserted characters. At least the last identifier has grown; (3) Repeat the second step. The editing behavior that produces such worst case with the state-of-the-art [21, 33] is the following: (1) The user types a character at the beginning of the document; (2) Repeat the first step.

The complexity analysis depends of the chosen structure to represent the replica. In this section, we focus on a tree structure but other structures (e.g. lists [33]) exposing different tradeoffs could be used. Nevertheless, the space complexity of identifiers – the main contribution of LSEQ – stays the same regardless of the chosen structure.

Space complexity. We distinguish the space complexity of each identifier from the space complexity of the tree. The former is important since it directly impacts the communication complexity, for that each identifier is broadcast to all editors; The latter is important since it represents the replicated document stored locally by each editor.

As stated in Section 4.1, paths require $\mathcal{O}(e^2)$ bits to be encoded, where e is the depth of the element in the tree. Fortunately, the depth is upper-bounded depending on the editing behavior that filled the exponential tree.

The random editing behavior fills the tree at random. The repeated insertions progressively fill the gaps in the tree without favoring any position in the document. Over insertions, the lowest branches of the tree become filled of elements and the depth of the tree grows slowly (see Figure 4a). Thus, the random editing behavior balances the tree. Being exponential, the tree stores $\sum_{i=1}^{k} 2^{(i^2-i)/2}$ elements, where k is the depth of the tree. The depth of paths is upper-bounded by $\mathcal{O}(\sqrt{\log I})$ concatenations, where I is the number of insertions. Since paths require $\mathcal{O}(e^2)$ bits to encode, paths have an optimal logarithmic space complexity $\mathcal{O}(\log I)$. This result applies to all variable-size identifier allocators [21, 33]. A tree structure factorizes common parts of identifiers. Overall, the space complexity is not the sum of identifiers but $\mathcal{O}(I \log I)$.

The monotonic editing behavior fills only one branch of the tree (see Figure 4b). Yet, since the maximum arity grows over depths, the growth of the tree tends decelerate over insertions. The tree stores up till $2^{e+1} - 1$ elements, where e is the depth of the tree. Hence, the number of concatenation composing a path is $\mathcal{O}(\log I)$, where I is the number of insertions. Since the binary representation of paths increases quadratically, the space complexity of paths taken individually is upper-bounded by $\mathcal{O}((\log I)^2)$. Overall, the space complexity of the tree remains upper-bounded by $\mathcal{O}(I \log I)$.

In the worst case, each insertion increases the depth of the tree. Thus, after I insertions, the tree comprises e elements, where e is the depth of the tree (see Figure 4c). The space complexity of paths grows quadratically and each new allocated path contains the full tree. For instance, the first allocated path is [31], the second one is [31.63], the third one is [31.63.127], etc. Overall, the space complexity of the tree is $\mathcal{O}(I^2)$ too.

Table I summarizes the space complexity of LSEQ. In particular, the expected growth of identifiers is bounded between an optimal logarithm and a polylogarithm, hence, a sub-linear upper bound compared to the number of insertions. To provide such improvement, LSEQ sacrifices on its worst case complexity that becomes quadratic. However, this worst case is made difficult to produce, for the two sub-allocations functions with antagonist designs settle each other's deficiency. Furthermore, if a malicious user tries to produce the worst case, the difference in complexity makes it easy to detect, hence, to handle. Table I also shows that compared to state-of-the-art, LSEQ significantly improves the identifiers size but exposes a small overhead on the full structure, i.e.

DOI: 10.1002/cpe

Table II. Upper bounds on time complexity of LSEQ. Where I is the number of insertions performed on the replicated sequence.

EDITING BEHAVIOR	TIME		
	LOCAL		REMOTE
	INS	DEL	INS / DEL
Random editing	$\mathcal{O}(\sqrt{\log I})$	$\mathcal{O}(1)$	$\mathcal{O}(\log I + \sqrt{\log I})$
Monotonic editing	$\mathcal{O}(\log I)$	$\mathcal{O}(1)$	$\mathcal{O}((\log I)^2 + \log I)$
Worst case	$\mathcal{O}(I)$	$\mathcal{O}(1)$	$\mathcal{O}(I)$

from $\mathcal{O}(I)$ to $\mathcal{O}(I \log I)$. The tradeoff is beneficial since the communication complexity of editors inherits from the improvement: as shown in Algorithm 1, each identifier is broadcast to all editors.

Time complexity. Time complexity provides insights about the performance evolution of each operation over insertions. They are divided between their local execution and their remote integration. Similarly to the space complexity, the analysis focuses on two editing behaviors (random and monotonic) to which we add the worst case.

The local insert operation simply consists in building a new path according to two adjacent paths. Therefore, it depends of the depths of the latter which grow depending on the editing behavior. Since the random editing behavior, the monotonic editing behavior, and the worst case respectively lead to a depth growth upper-bounded by $\mathcal{O}(\sqrt{\log I})$, $\mathcal{O}(\log I)$ and $\mathcal{O}(I)$, the time complexity of the local part of the insert operation follows: $\mathcal{O}(\sqrt{\log I})$, $\mathcal{O}(\log I)$ and $\mathcal{O}(I)$.

The local delete operation simply broadcasts the identifier of the element to remove to all participants. Hence, the time complexity is constant $\mathcal{O}(1)$ regardless of the editing behavior.

Both the remote insert and delete operations perform the same instructions to integrate the received result. Consequently, they have an identical time complexity. Random, monotonic, and worst case editing behaviors respectively lead to $\mathcal{O}(\sqrt{\log I})$, $\mathcal{O}(\log I)$ and $\mathcal{O}(I)$ levels in the exponential tree structure. Since the children of each node are ordered by path, we perform binary searches recursively until we find the leaf. During binary searches, the delete operation removes the nodes it explores if they, or their children, do not contain at least another element. For instance, considering two elements associated to the following paths: $[3.1]$ and $[3.1.2]$. If one deletes the first element, the path is kept since the path labeled by 3 then 1 is still required for the second element. Then, if one deletes the second element and the node containing it has no children, the path and nodes are truly removed. The complexity of a binary search depends of the level l at which it is performed: $\mathcal{O}(\log 2^l)$. Repeated binary searches lead to the upper bound $\mathcal{O}(\sum_{i=1}^{e}(\log 2^i))$, where e is the depth of the tree. Replacing the depth e, upper bounds on time complexity are $\mathcal{O}(\log I + \sqrt{logI})$, $\mathcal{O}((\log I)^2 + \log I)$, and $\mathcal{O}(I)$ for random, monotonic, and worst case editing behaviors respectively. In the worst case, the algorithms perform one comparison per element, i.e. per level, in the tree until they reach the deepest leaf.

To interface the user's view of the document with the underlying replicated sequence, an additional lookup operation is necessary. Its goal is to retrieve the identifier of the element at the specified index in the sequence, and converse. Since the structure is a tree, this access is not direct.

To consistently manage the translation between sequence structure and the tree structure, the latter stores with each node the number of elements included in its sub-trees. Each insertion and removal updates these counters as they explore a path. Then, processing an index comes down to count the number of elements in the siblings of explored nodes starting from the beginning or the end of the sequence. Unfortunately, it can be costly depending on the editing behavior that filled the tree.

The random editing behavior leads to an exponential tree of depth bounded by $\mathcal{O}(\sqrt{\log I})$. In such a case, a very small portion of the whole tree will be explored. The upper bound happens when the explored nodes are exactly in the middle of its siblings, for it forces to check at least half of these sibling per depth. Since each node stores twice as much children as its parents. The number of siblings to check doubles at each depth. Overall, the sum of these siblings is equal to the number of node at the deepest level, which is $\mathcal{O}(2^{\sqrt{\log I}})$ elements.

DOI: 10.1002/cpe

Table III. Upper bounds on time complexity of the lookup on a LSEQ structure. Where I is the number of insertions performed on the replicated sequence.

EDITING BEHAVIOR	TIME LOOKUP
Random editing	$\mathcal{O}(2\sqrt{\log I})$
Monotonic editing / Worst case	$\mathcal{O}(I)$

From an identical reasoning, the lookup operation after monotonic editing – which also constitutes the worst case for this operation – is upper-bounded by $\mathcal{O}(I)$. Indeed, only one branch is filled and explored. The counters become useless since all nodes but one have only one child. The exploration directly ends up in the deepest level containing $\mathcal{O}(2^{\log_2 I})$ elements where half of them must be checked leading to $\mathcal{O}(I)$.

Tables II and III summarize the time complexity of operations. The exponential tree structure leads to efficient update operations. Its drawback lies in the lookup operation making the link between the sequence and the tree. In particular, monotonic editing leads to the worst case scenario where the access grows linearly. Fortunately, (i) the view should not be updated if the index of the change falls outside the user's visibility window. Thus, the lookup can stop earlier if it explores the tree outside the scope; (ii) keeping fast accesses to the newest path and its adjacent paths removes the cost of the lookup, for the repeated insertions – if monotonic – alone can maintain these fast accesses up-to-date.

Summary. The complexity analysis reveals the improvements brought by LSEQ as well as their cost. LSEQ sacrifices on its worst case complexity to improve on other editing behaviors that are considered more likely to happen in collaborative editing. The most important result is the polylogarithmic space complexity of identifiers, for it applies on communication complexity. In comparison, state-of-the-art allocators [21, 33] only provided linearly upper-bounded identifiers. Other analysis such as the local space consumed by the replicated structure, and the time complexity of provided operations shows that the tree as replicated structure is efficient. Nonetheless, based on preferences, one could choose an array – as a flat version of the tree – to improve the performance of operations at the cost of increased memory usage [33]. Section 6 validates our complexity analysis on experiments. The next section describes the other components necessary to build a decentralized collaborative editor that scales.

5. DECENTRALIZED REAL-TIME EDITOR

Figure 5 describes the decentralized editors' architecture that comprises 4 layers where each can constitute an obstacle to scalability: (i) the communication layer includes the editing session membership mechanism and the information dissemination protocols; (ii) the causality layer includes the causality tracking structure and the catch up mechanism; (iii) the sequence layer includes a convergent replicated structure representing the document; (iv) the user interface layer includes the editor as a graphical entity that users can interact with.

The left part of the figure depicts the common process chain: when a user performs an operation on the document, the operation is applied to the distributed sequence. Then it decorates the operation with causality tracking metadata. Finally, the editor broadcasts it using the neighborhood provided by the membership protocol. Conversely, when the editor receives a broadcast message, it checks if the operation is causally ready to be delivered. Once the condition is verified, it applies the operation to the distributed sequence which notifies the graphical user interface of the changes. The right part of the figure corresponds to the catch up strategy where a member may have missed operations due to dropped messages, or simply because the user worked offline for a while. Therefore, when the editor is online, it regularly asks to its neighborhood for the missing operations.

CRATE [15] is a real-time decentralized editor running directly within web browsers and available at `https://github.com/Chat-Wane/CRATE`. This section describes the model of CRATE:

DOI: 10.1002/cpe

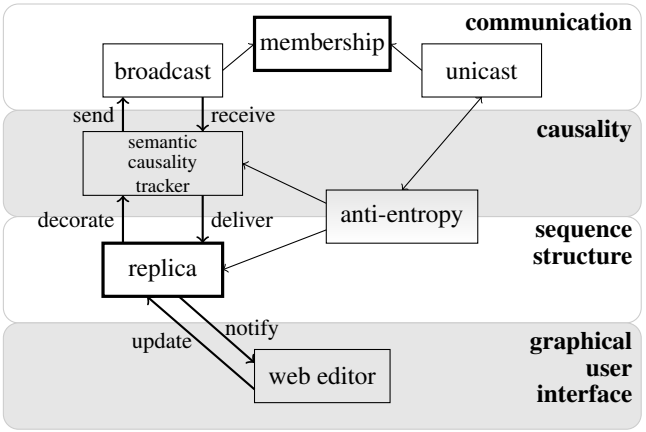


Figure 5. The four layers of decentralized editors' architecture.

communication, causality, and sequence. It reviews each component and its contribution to the system. This composition provides a tradeoff favoring communication complexity. The system scales well in terms of number of participants and document size. Section 6 validates this claim through experiments.

Communication. Editing sessions can gather from small to large groups during their life time, e.g., massive open online courses (MOOC) may start with a large number of students which can quickly decrease due to a lack of interest [6]. Also, editing sessions vary from their size according to the scope of documents. For instance, a document describing a personal project and the visibility of which is limited to friends gathers significantly less people than a document about a large event, such as a collaboratively written report about a conference. The communication layer should transparently handle any editing session size in a scalable manner.

CRATE uses SPRAY [18] to automatically adapt its functioning to the editing session size. SPRAY is a random peer sampling protocol [10] that incrementally builds and maintains the neighborhood of editors. Thus, each editor has a small set of editors to communicate with. The size of sets scales logarithmically compared to the number of current participants in the editing session. If the editing session starts with 10 participants, they have 2.3 neighbors in average. If the editing session grows to 1000 connected participants, they have 6.9 neighbors in average. If the editing session shrinks to 10 participants again, they have 2.3 neighbors in average again.

To broadcast operations, the editor makes extensive use of neighborhoods to propagate changes. Indeed, when a user performs a change, the editor sends it to the editors in its neighborhood. Each neighbor is in charge of integrating and forwarding the change to its own neighbors. Changes transitively reach all editors very quickly, for the average shortest path from an editor to others remains very small. The communication complexity at each editor is upper-bounded by $\mathcal{O}(M \ln(R))$, where M is the message size, and R is number of replicas connected during the propagation.

Causality tracking. To preserve consistent replicas, the same outcome must result from the generation of the operation and its integration [28]. Often, the behavior of an operation depends of others previously integrated. For instance, the generation of a removal operation requires the targeted element to exist, hence, its insert operation to be integrated. These *happens before* relationships [11] constrain the integration order. Unfortunately, the more the order is constrained, the costlier it becomes. Accurately tracking causal relations of one operation with all others requires at least $\mathcal{O}(W)$ both locally and in communication overhead, where W is the number of participants that ever wrote in the document [7]. Such communication overhead confines its usage to small editing sessions.

CRATE uses a version vector with exceptions [12, 14]. Each operation is uniquely identified by a unique site identifier along with a counter. Each broadcast message includes such identifier. For

DOI: 10.1002/cpe

each editor, (i) an integer denotes the maximal counter of received operations that originated from this editor and (ii) a set of integers denotes the exceptions, i.e., the operations known as not yet received from this editor.

This causality tracking structure tracks only the semantically related pairs of operations (e.g. the removal of an element with its insertion). If the operations arrives to an editor out of order, the removal waits for the corresponding insertion. On the opposite, it immediately integrates received insertions. The structure also serves as a tool to identify differences between replicas when an editor needs to catch up with the current state of the document in the live editing session. Each editor periodically performs an anti-entropy [8] round ensuring that no operations went missing due to an unreliable network or offline writing.

While the local overhead implied by such structure is upper-bounded by $\mathcal{O}(W)$, the communication overhead is constant $\mathcal{O}(1)$.

Shared sequence. Sun et al. state that collaborative editing requires convergence, causality, and intention preservation [28]. A replicated structure for sequences aims to provide eventually convergent [25] copies of the document while providing meaningful operations preserving intention. Thus, users visualize the same document and can modify it consistently.

CRATE uses a conflict-free replicated data type for sequences [25] to represent its documents. Such sequence types rely on unique and immutable identifiers. CRATE uses LSEQ to provide the identifiers. Delete operations truly remove the targeted elements from the underlying structure. Furthermore, the size of these identifiers are expected to be bounded between a logarithmic bound and a polylogarithmic bound compared to the number of insertions in the sequence. Being sublinear, the structure does not require balancing which needs consensus algorithms that do not scale [13].

Communication complexity. The logarithmic multiplicative factor of broadcast, the constant cost of causality tracking, and the cost of identifiers, lead to a communication complexity bounded between $\mathcal{O}((\log I).(\ln R))$ and $\mathcal{O}((\log I)^2.(\ln R))$ depending on the editing behavior, where I is the number of insert operations performed on the sequence, and R the number of replicas in the editing session during the message propagation.

6. EXPERIMENTS

The objective of this experiment section is twofold:

- To confirm the complexity of LSEQ. Monotonic editing behavior should lead (i) to a polylogarithmic growth of identifiers compared to the number of insertions; (ii) to a logarithmic increasing of execution time except for the lookup which is expected to be linear.
- To show the impact on the generated traffic on a real decentralized collaborative editor. We expect that both the identifier size and neighborhood size impact the traffic. Since the former grows polylogarithmically, and the latter grows logarithmically, we expect the traffic to scale in terms of the number of users and the number of operations.

6.1. Space complexity

Objective: To confirm the space complexity of LSEQ's identifiers and to highlight the improvement over state-of-the-art.

Description: We consider two editing behaviors: (i) random which consists in inserting elements at random position in the sequence; (ii) left-to-right which consists in inserting elements at the end of the sequence. We measure the average bit size of paths allocated by Treedoc [21], Logoot [33], and LSEQ [17]. The generated documents reach half a million characters.

Result: Figure 6 shows the results of the experiment. The x-axis denotes the size of the document. The y-axis denotes the average size of generated paths. The top figure shows the measurements of the random editing behavior while the bottom figure shows the measurements of the left-to-right editing behavior. As expected, we observe that random editing leads to logarithmic

DOI: 10.1002/cpe

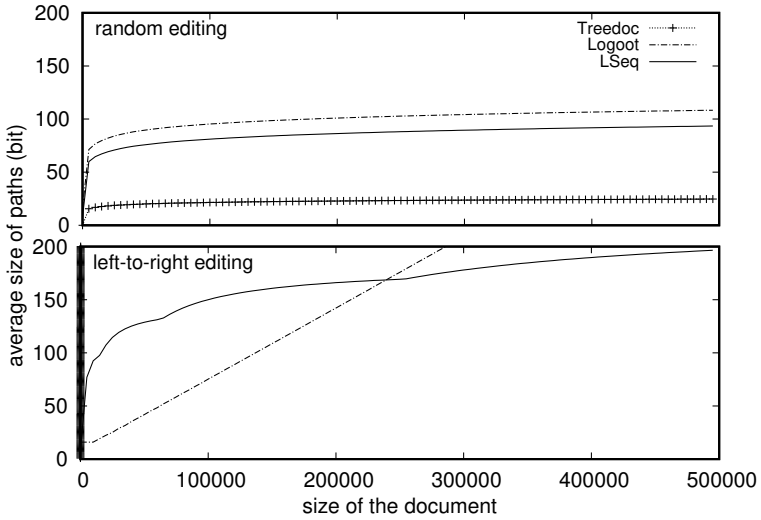


Figure 6. Size of paths allocated by Treedoc, Logoot, and LSEQ.

growth of paths whatever the allocation function. In addition, Treedoc allocates the smallest paths, followed by LSEQ, followed by Logoot. Nonetheless, on left-to-right editing, paths allocated by Treedoc grow extremely quickly to such an extent that its plot merges with the y-axis. Logoot's paths grow slower but still linearly compared to the number of insertions in the document. We observe that LSEQ exposes a better space complexity compared to the state-of-the-art, which eventually leads to smaller allocated paths.

Reason: The random editing leads to a balanced underlying structure, hence the logarithmic progression of paths allocated by the three strategies. In this case, Treedoc is better because it has an arity of 2^1, for it is a binary tree. In this setup, LSEQ starts with an arity of 2^8 while Logoot has an arity of 2^{16}, hence LSEQ allocating smaller paths than Logoot in random editing. The left-to-right editing behavior tends to unbalance the tree over insertions. In Treedoc, monotonic editing is the worst case scenario where each new path contains all paths allocated before it. Logoot is designed to handle left-to-right editing. It allocates paths but keeps space for upcoming insertions. Still, since its maximum arity is constant, the number of characters in a branch is constant, hence the linear growth of paths. On the opposite, LSEQ doubles its arity at each level of the tree. The branch can store twice as much elements as its parent. Hence a growth that scales sublinearly compared to the number of insertions in the document.

6.2. Time complexity

Objective: To confirm the time complexity of LSEQ's operations. We expect good scalability of operations except for the lookup after monotonic editing.

Description: This experiment involves one user who performs operations on its local copy of the document. The benchmark ran on a MacBook Pro with 2.5 GHz Intel Core i5, with Node.js 4.1.1 on Darwin 64-bit. For each operation, we create a document containing $I - 1$ characters and measure the time taken by the I^{th} operation. The operation set includes the lookup, the local part of an insertion, the remote part of an insertion, and the remote part of a deletion – the local part of a deletion only consist of broadcasting the identifier. We perform the measurements multiple times on two kinds of documents. Firstly, a document generated by random editing, i.e. the underlying LSEQ tree is balanced. Secondly, a document generated by monotonic editing, i.e. one branch per level of the LSEQ tree is filled. We focus on tendencies rather than absolute values. Javascript performs on-the-fly optimization which we limit in order to show the real time contribution of each operation.

Result: Figure 7 shows the result of this experiment. The x-axis denotes the number of insert operations performed before the measured operation. The y-axis denotes the average time taken by this operation. Both axis are on a logarithmic scale. The top part of the figure focuses on a structure filled with insertions following a random editing behavior while the bottom part of the figure focuses

DOI: 10.1002/cpe

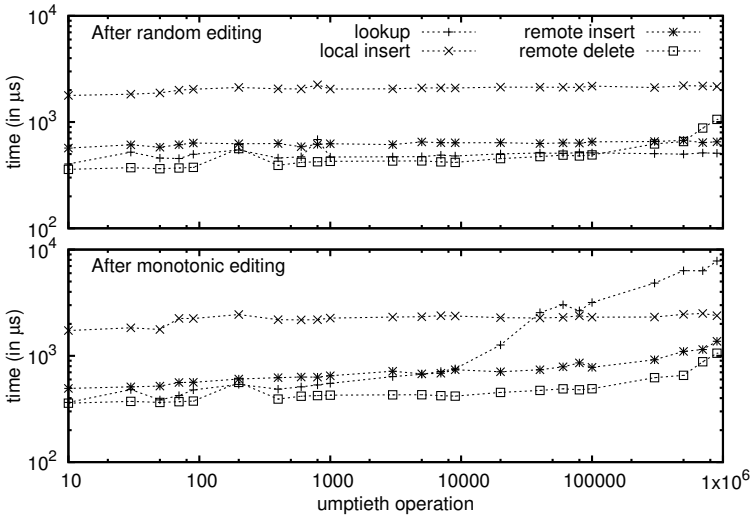


Figure 7. Performance of the umptieth operation performed in LSEQ.

on a structure filled with insertions following a monotonic editing behavior. We observe that the measured values barely grow after random editing, regardless of the operation. On the other hand, while the local insertion after monotonic editing remains stable, we observe a linear growth with the lookup execution time, and a slower growth for both the remote insertions and the remote deletions.

Reason: After the insertions at random positions, the underlying tree of LSEQ is balanced. Therefore, the range of influence of each operation is limited to a small subset of the elements composing the document. For instance, the lookup operation does not need to explore each element of the tree. Instead, it quickly discards a lot of irrelevant branches at each level of the tree because the index does not fall into their range. However, this remark does not hold when the insertions followed a monotonic editing behavior. Indeed, in this case, most elements are located in the one and deepest level of the tree. Thus, the lookup likely crawls to this level and then inspects each elements to count their children and actualize its current index. The remote operations measurements follow the same reasoning: they must perform a binary search at each depth of the tree; since the random editing structure is balanced, the average depth of the search is smaller compared to the structure resulting from monotonic editing behavior. Hence, a lower time complexity.

6.3. Communication complexity

Objective: To show that our LSEQ-based decentralized collaborative editor scales well in terms of number of participants and document size. We expect the traffic to grow polylogarithmically compared to the document size (contribution of LSEQ), and logarithmically compared to the number of participants (contribution of SPRAY).

Description: This experiment was conducted on the Grid'5000 testbed with 101 machines running from 1 to 6 web browsers. The results concern editing sessions involving from 101 members to 601 members. The members of each session collaboratively write a document by repeatedly inserting new characters at the end of the document. We measure the average outgoing traffic of members, the average size and variance of partial views. An arbitrary number of 100 operations per second are uniformly distributed among participants. The editing sessions last 7 hours. The documents reach millions of characters.

Result: Figure 8 shows the result of this experiment. The x-axis denotes the experiment progression. The y-axis shows the average outgoing traffic generated by members. The legend shows the average size and variance of partial views for each editing session. As expected, we observe two results: (i) the growth of each individual plot corresponds to a polylogarithmic increasing due to the paths allocated by LSEQ; (ii) the growth between plot corresponds to a logarithmic increasing due to the partial views maintained by SPRAY. In addition, the load is balanced among participants thanks to SPRAY. Hence, CRATE scales well in terms of number of participants and number of insertions.

DOI: 10.1002/cpe

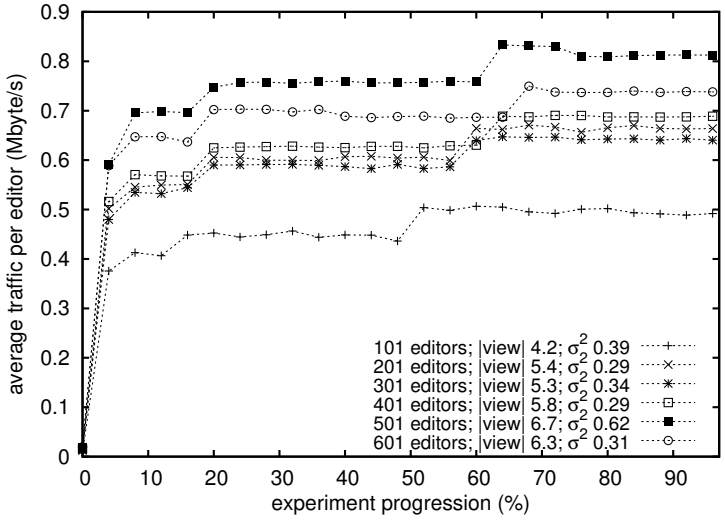


Figure 8. Average traffic per second at each editor during real-time editing sessions.

Treedoc- and Logoot-based collaborative editors would have linear growth, for both Treedoc and Logoot allocate linearly growing identifiers during left-to-right editing (as shown in Section 6.1).

Reason: When an outsider joins the network, SPRAY injects a logarithmic number of connections compared to the network size. Since it does not use any global knowledge, this number is a rough estimation and may lead to a small deviation between the actual value and the expected theoretical value. For instance, the editors of the editing session involving 601 members have a smaller view in average than the editors of the editing session involving 501 members (6.3 against 6.7). As consequence, the generated traffic of the former is smaller than the generated traffic of the latter. Nevertheless, regardless of the editing session, the traffic is well balanced among participants, for the variance in view sizes remains small.

Each time an insertion is performed, an identifier is created and broadcast. Each editor sends each identifier to all the editors included in its partial view exactly once. Since the identifiers grow polylogarithmically compared to the number of insertions (thanks to LSEQ), and since the views grow logarithmically compared to the network size (thanks to SPRAY), the resulting traffic is the product of both this polylogarithm and this logarithm.

7. CONCLUSION

In this paper, we demonstrated that LSEQ makes large-scale decentralized editors a practicable alternative to mainstream editors. LSEQ-based editors such as CRATE enable real-time editing at anytime, anywhere, whatever the number of participants, without third parties.

We proposed an original tradeoff on time, space, and communication complexities. The balance mainly consists in providing sublinear communications while maintaining affordable memory consumption and efficient operations. We validated this new tradeoff on large-scale experiments involving up till 600 web browsers interconnected.

Editors smoothly adapt their behavior to editing sessions without any global knowledge. Thus, they become useful not only for small groups but also in large events such as conferences, or massive online lectures, that can gather a large number of participants.

For future work, we intend to study the behavior of users when they are part of large editing sessions. Are current user interfaces descriptive enough to handle such scenarios? CRATE allows clients to contribute to their editing sessions by sharing their resources (e.g. bandwidth). It makes real-time editing cheap for any web applications, for web application providers do not spend further resources. For instance, CRATE could be embedded in Wikipedia to solve peaks of concurrent changes resulting in conflicts.

DOI: 10.1002/cpe

Finally, we demonstrated how a well-known collaborative application can be provided without collaboration providers. We aim to explore if other collaborative applications such as distributed calendars or crowdsourcing platforms can be deployed on a network of browsers.

ACKNOWLEDGEMENTS

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. M. Ahmed-Nacer, C.-L. Ignat, G. Oster, H.-G. Roh, and P. Urso. Evaluating CRDTs for Real-time Document Editing. In ACM, editor, *11th ACM Symposium on Document Engineering*, pages 103–112, Mountain View, California, United States, Sept. 2011.
2. A. Andersson. Faster deterministic sorting and searching in linear space. In *Proceedings of 37th Annual Symposium on Foundations of Computer Science, 1996.*, pages 135–141. IEEE, 1996.
3. A. Andersson and M. Thorup. Dynamic ordered sets with exponential search trees. *J. ACM*, 54(3), June 2007.
4. 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.
5. P. Bailis and A. Ghodsi. Eventual consistency today: Limitations, extensions, and beyond. *Commun. ACM*, 56(5):55–63, May 2013.
6. L. B. Breslow, D. E. Pritchard, J. DeBoer, G. S. Stump, A. D. Ho, and D. T. Seaton. Studying learning in the worldwide classroom: Research into edx's first mooc. *Research & Practice in Assessment*, 8:13–25, 2013.
7. B. Charron-Bost. Concerning the size of logical clocks in distributed systems. *Inf. Process. Lett.*, 39(1):11–16, July 1991.
8. 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*, pages 1–12. ACM, 1987.
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, pages 3:1–3:10, New York, NY, USA, 2010. ACM.
10. M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. Van Steen. Gossip-based peer sampling. *ACM Transactions on Computer Systems (TOCS)*, 25(3):8, 2007.
11. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978.
12. D. Malkhi and D. Terry. Concise version vectors in winfs. *Distributed Computing*, 20(3):209–219, 2007.
13. A. Mostéfaoui, H. Moumen, and M. Raynal. Signature-Free Asynchronous Binary Byzantine Consensus with $t<n/3$, $O(n^2)$ Messages, and O(1) Expected Time. *Journal of the ACM (JACM)*, 62:1000–1020, Dec. 2015.
14. M. Mukund, G. Shenoy R., and S. Suresh. Optimized or-sets without ordering constraints. In M. Chatterjee, J.-n. Cao, K. Kothapalli, and S. Rajsbaum, editors, *Distributed Computing and Networking*, volume 8314 of *Lecture Notes in Computer Science*, pages 227–241. Springer Berlin Heidelberg, 2014.
15. B. Nédelec, P. Molli, and A. Mostéfaoui. Crate: Writing stories together with our browsers. In *Proceedings of the 25th International Conference on World Wide Web*, WWW '16 Companion, New York, NY, USA, 2016. ACM.
16. B. Nédelec, P. Molli, A. Mostéfaoui, and E. Desmontils. Concurrency effects over variable-size identifiers in distributed collaborative editing. In *DChanges*, volume 1008 of *CEUR Workshop Proc.* CEUR-WS.org, Sept. 2013.
17. B. Nédelec, P. Molli, A. Mostéfaoui, and E. Desmontils. LSEQ: an Adaptive Structure for Sequences in Distributed Collaborative Editing. In ACM, editor, *13th ACM Symposium on Document Engineering*, Sept. 2013.
18. B. Nédelec, J. Tanke, D. Frey, P. Molli, and A. Mostefaoui. Spray: an Adaptive Random Peer Sampling Protocol. Technical report, LINA-University of Nantes ; INRIA Rennes - Bretagne Atlantique, Sept. 2015.
19. W. Oremus. Online class on how to teach online classes goes laughably awry. `http://www.slate.com/blogs/future_tense/2013/02/05/mooc_meltdown_coursera_course_on_fundamentals_of_online_education_ends_in.html`, february 2013.
20. G. Oster, P. Urso, P. Molli, and A. Imine. Data consistency for p2p collaborative editing. In *Proceedings of the 20th anniversary conference on Computer supported cooperative work, 2006*, pages 259–268. ACM, 2006.
21. N. Preguiça, J. M. Marqus, M. Shapiro, and M. Letia. A commutative replicated data type for cooperative editing. In *2009, ICDCS'09. 29th IEEE International Conference on Distributed Computing Systems*, pages 395–403. Ieee, June 2009.
22. 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*, 71(3):354–368, 2011.

DOI: 10.1002/cpe

23. Y. Saito and M. Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, Mar. 2005.
24. M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Rapport de recherche RR-7506, INRIA, Jan. 2011.
25. M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. *Stabilization, Safety, and Security of Distributed Systems*, pages 386–400, 2011.
26. V. Strauss. How online class about online learning failed miserably. `https://www.washingtonpost.com/news/answer-sheet/wp/2013/02/05/how-online-class-about-online-learning-failed-miserably/`, february 2013.
27. C. Sun and C. Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proc. of the 1998 ACM conf. on Comp. supported cooperative work*, CSCW '98, pages 59–68, New York, NY, USA, 1998. ACM.
28. C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Trans. on Computer-Human Interaction (TOCHI)*, 5(1):63–108, 1998.
29. D. Sun and C. Sun. Context-based operational transformation in distributed collaborative editing systems. *IEEE Transactions on Parallel and Distributed Systems*, 20(10):1454–1470, Oct 2009.
30. B. Tomlinson, J. Ross, P. Andre, E. Baumer, D. Patterson, J. Corneli, M. Mahaux, S. Nobarany, M. Lazzari, B. Penzenstadler, A. Torrance, D. Callele, G. Olson, S. Silberman, M. Stünder, F. R. Palamedi, A. A. Salah, E. Morrill, X. Franch, F. F. Mueller, J. J. Kaye, R. W. Black, M. L. Cohn, P. C. Shih, J. Brewer, N. Goyal, P. Näkki, J. Huang, N. Baghaei, and C. Saper. Massively distributed authorship of academic papers. In *CHI '12 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '12, pages 11–20, New York, NY, USA, 2012. ACM.
31. N. Vidot, M. Cart, J. Ferrié, and M. Suleiman. Copies convergence in a distributed real-time collaborative environment. In *Proc. of the 2000 ACM Conf. on Comp. Supported Cooperative Work*, CSCW '00, pages 171–180, New York, NY, USA, 2000. ACM.
32. S. Weiss, P. Urso, and P. Molli. Wooki: A p2p wiki-based collaborative writing tool. In B. Benatallah, F. Casati, D. Georgakopoulos, C. Bartolini, W. Sadiq, and C. Godart, editors, *Web Information Systems Engineering - WISE 2007*, volume 4831 of *Lecture Notes in Computer Science*, pages 503–512. Springer Berlin Heidelberg, 2007.
33. S. Weiss, P. Urso, and P. Molli. Logoot: a scalable optimistic replication algorithm for collaborative editing on p2p networks. In *ICDCS'09. 29th IEEE International Conf. on Dist. Comp. Sys., 2009*, pages 404–412. IEEE, 2009.
34. S. Weiss, P. Urso, and P. Molli. Logoot-undo: Distributed collaborative editing system on p2p networks. *IEEE Transactions on Parallel Distributed Systems*, 21(8):1162–1174, 2010.
35. Q. Wu, C. Pu, and J. Ferreira. A partial persistent data structure to support consistency in real-time collaborative editing. In *IEEE 26th International Conference on Data Engineering (ICDE), 2010*, pages 776–779, 2010.
36. W. Yu. A string-wise crdt for group editing. In *Proceedings of the 17th ACM international conference on Supporting group work*, GROUP '12, pages 141–144, New York, NY, USA, 2012. ACM.
37. M. Zawirski, M. Shapiro, and N. Preguiça. Asynchronous rebalancing of a replicated tree. In *Conf. Française de Systèmes d'Exploitation (CFSE)*, page 12, Saint-Malo, France, May 2011.

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, eventual consistency, and networks.

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 is co-head of the GDD research team within the LINA Lab. His research interests are on distributed computing: agreement and calculability issues in asynchronous, fault-prone distributed systems and replicated data structures.

DOI: 10.1002/cpe