CRATE: Decentralized Real-Time Editing in Browsers

Brice Nédelec, Pascal Molli, Achour Mostefaoui
LINA, 2 rue de la Houssinière
BP92208, 44322 Nantes Cedex 03
first.last@univ-nantes.fr

ABSTRACT

Distributed real-time editors such as Google Docs, or Etherpad allow users distributed in space and organizations to collaborate easily with only web browsers. Yet, main stream editors rely on central servers with issues in privacy, single point of failure and scalability. Decentralized editors settle privacy problems but scalability issues remain. None of existing algorithms provide satisfying complexities in both communication and space. In this paper, we propose CRATE; a working decentralized real-time editor that supports large number of users working on large documents. In particular, CRATE achieves a sublinearly upper-bounded communication complexity in $\mathcal{O}((\log d)^2 \ln m)$ where d is the document size and m the membership size. Thanks to WebRTC, CRATE can be easily deployed on a network of browsers. We evaluate CRATE on large-scale experiments in the Grid'5000 testbed involving uptill 600 participants. As expected, we observe a logarithmic progression of the generated traffic according to the size of the shared document and the number of participants.

Keywords

Document authoring, distributed collaborative editing, optimistic replication, polylog sequence encoding, conflict-free replicated data types, adaptive peer sampling, WebRTC

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 on privacy, censorship, economic intelligence. It also raises scalability issues in term of number of participants. Despite that small groups currently constitute the main range of users, events such as massive online lectures, TV shows, conferences gather larger groups. Google Docs supports large groups but only the first fifty users can edit, next users have their rights limited to document reading. We think that real-time editors should allow editing at anytime and anywhere, whatever the number of participants. Real-time editing sessions can be highly dynamic; even if only few authors are editing a document simultaneously, the editing session includes a much larger group of potential writers. In this paper we focus on building a real-time editor that supports privacy and that adapts gently from small groups to large groups.

Decentralized real-time editors [17, 24, 26] do not require intermediate servers and by the same solve privacy issues. However, scalability issues remain. Addressing scalability requires finding a good trade-off between communication, space and time complexities. Achieving a sublinear communication complexity compared to the editing session size is crucial for supporting large groups. In the best case, if we have n participants, sending a typed character to others requires to contact $log(n)$ participants. But consistency maintenance of documents requires each message to piggyback additional information which greatly impacts the communication complexity.

Decentralized algorithms of operational transformation [26] require piggybacking a state vector in order to detect concurrent operations. Unfortunately, state vectors grow linearly compared to the number of members that ever participated in the authoring.

Conflict-Free Replicated Data Types [21] (CRDTs) such as WOOT [17] piggyback a constant-size identifier which is optimal. Nonetheless, their local space complexity grows monotonically which eventually leads to the need of costly garbage collecting. Other algorithms [18, 31] limit their local space consumption. Nevertheless, they piggyback variable-size identifiers that can grow linearly compared to the document size. In such case, they eventually need to run a costly consensus algorithm to balance documents [34]. Finally, Algorithm LSEQ [14] aims to avoid such consensus by sublinearly upper-bounding the space complexity of its identifiers. It conjectured a polylogarithmic growth of the identifiers size $\mathcal{O}((\log d)^2)$ where d is the document size.

In this paper, we propose CRATE, a scalable real-time editor that runs on a network of browsers. CRATE relies on WebRTC and browser-to-browser communication to provide an easy access for end-users. Compared to state-of-the-art, it presents a new original trade-off which balances the load between space, time, and communication complexities. The contributions of this paper are threefold:

- Compared to previous work [15], we demonstrate the upper bounds on space and time complexities of LSEQ. It constitutes an original trade-off for building decentralized real-time editors.

Algorithms	Communication	Space
Operational transformation [23, 26]	$\mathcal{O}(\boldsymbol{W} \ln R)$	$\mathcal{O}(\boldsymbol{H.W})$
Tombstones CRDTs [1, 7, 17, 19, 29, 32, 33]	$\Omega(\ln R)$	$\Omega(\boldsymbol{H})$
Variable-size array-based CRDTs [30]	$\mathcal{O}(\boldsymbol{I} \ln R\|)$	$\mathcal{O}(\boldsymbol{I^2 + W})$
Variable-size tree-based CRDTs [18]	$\mathcal{O}(\boldsymbol{I} \log \boldsymbol{I} \ln R)$ or $\mathcal{O}(\log I \ln R\|)$	$\mathcal{O}(I \log I + \boldsymbol{W})$ or $\mathcal{O}(I + \boldsymbol{W})$
This paper [15]	$\mathcal{O}((\log I)^2 \ln R)$	$\mathcal{O}(I \log I + \boldsymbol{W})$

Table 1: Communication and space complexities of decentralized approaches. W is the number of writers, R is the number of replicas (readers and writers), H is the number of operations in the historic (insertions and deletions), and I is the number of insertions. Bottlenecks of each approach are highlighted.

- Compared to previous work [16], we deploy the adaptive membership protocol Spray in web browsers thanks to WebRTC. We observe that Spray's adaptivity allows handling high dynamicity of real-time editing sessions.
- We conducted experimental studies to validate the complexities of Crate. The experiments took place in the Grid'5000 testbed and involved uptill 600 real web browsers opened to edit a shared document. At a rate of 100 insertions per seconds, the document size reached above 1 million characters. As expected, we observed a logarithmic growth of the traffic compared to the number of participants, and a polylogarithmic growth of the traffic compared to the size of the document. This result confirms that Crate does not require any costly garbage collection mechanism.

The remainder of this paper is organized as follows: Section 2 reviews the related work with an emphasis on the complexity trade-off proposed by other approaches. Section 3 follows with a description of Crate and its core components. Section 4 highlights the scalability of Crate while validating the complexity analysis of LSeq and Spray. Finally, Section 5 concludes the paper and discusses about perspectives.

2. RELATED WORK

Real-time distributed collaborative editors consider multiple participants, each hosting a copy of a shared sequence of characters. A participant updates at any time its local copy by inserting or deleting a character. Then, the operation is eventually delivered to all other members. Finally, delivered operations are re-executed [20]. Consistency requires that all members eventually converge to an identical state, i.e., when the system is idle, all copies become similar [4]. Consistency also requires ensuring intention preservation, i.e., effects observed at generation time must be re-observed at re-execution time regardless of concurrent operations [25]. Consistency finally requires ensuring causality, i.e., the re-execution order of operations follows the *happen before* relationship [11] stated at generation.

Several complexities characterize decentralized 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, including state descriptors, logs, etc.
- Communication complexity: complexity of messages transiting the network.

Solving scalability issues requires finding a balanced trade-off between communication, space and time complexities. Among others, the communication complexity is the most discriminant. It requires $\mathcal{O}(m.\ln R)$ where $(\ln R)$ is a minimal multiplicative factor induced by the broadcasting mechanism which depends of the session size R including both readers and writers; where m is the message embedding metadata necessary to maintain consistency. To scale, the space complexity of these metadata must be sublinear.

Operational transformation (OT) allows building both centralized and decentralized real-time editors. At generation, the processing time of operations is constant. At integration, OT transforms the received operations against concurrent ones which were generated on the same state. An integration algorithm such as COT [26] or SOCT2 [27] along with transformation functions (ensuring transformation properties) guarantee a consistent model. The integration time depends of concurrency and differs among the algorithms. For instance, COT's integration time complexity is exponential. It can reduce it to linear at the expense of space complexity. SOCT2's integration time complexity is quadratic. Whatever the trade-off, OT's integration algorithms rely on concurrency detection which costs, at least, a state vector [6] the size of which grows linearly compared to the number of members W who ever participated in the authoring. Since each message embeds such vector, decentralized OT approaches are not praticable in large groups subject to churn where people join and leave the network frequently and freely. As for space complexity, OT approaches require an historic of operations H, each operation being linked to their state vector, hence $\mathcal{O}(W.H)$. Such monotonically growing historic can be cut at the price of consensus which, once again, does not scale. Thus, OT approaches are the best in humble confined environment like local area networks. However, their performance degrades when the network size grows and becomes less predictable.

Conflict-free replicated data types (CRDTs) for sequences constitute the latter approaches which solve concurrent cases by providing commutative and idempotent operations. As such, and compared to OT, the causality tracking cost is drastically reduced since it only requires tracking semantically related operations. For instance, the removal of a particular element must follow its insertion. The com-

mutative property of operations ensures the consistency of the system. CRDTs provide commutativity by associating a unique and immutable identifier to each element. Defining a total order among the identifiers allows retrieving the sequence of characters.
CRDTs propose interesting trade-offs since 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, the identifiers consume space which, in turns, impacts the communication complexity.

Tombstone-based CRDTs such as WOOT [17] associate a constant size identifier $\mathcal{O}(1)$ to each element but whose removals of elements only hide them to the users. Therefore, removed elements keep consuming space leading to a monotonically growing document, hence $\mathcal{O}(H)$. Destroying removed elements requires running a costly consensus algorithm to determine if everyone received the removal and agree on definitely throw out the element. Such algorithm is prohibitively costly and does not scale in number of members, especially in network subject to churn. Since the structure contains the full history of the document, such approach do not require further information to track causally related operations.

Variable-size identifiers CRDTs truly destroy elements targeted by removals. However, they allocate identifiers the size of which is determined at generation. The allocation function becomes crucial to maintain identifiers under acceptable boundaries. Unfortunately, they depend of 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. But with an identical strategy, writing the same sequence 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 insertions in the sequence, i.e., $\mathcal{O}(I)$. If the structure stores each identifier in a flat array, it grants fast access at the price of quadratic space complexity $\mathcal{O}(I^2)$. If it factorizes common parts of identifiers as a tree structure, it achieves better space complexity. Yet, in the worst case, a costly distributed consensus is still required to balance the structure [34]. 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 worth noting that, contrarily to OT, the vector is kept local.
Algorithm LSeq [15] aims to avoid such consensus by sublinearly upper-bounding the space complexity of variable-size identifiers. It conjectured a polylogarithmic progression of the identifiers size $\mathcal{O}((\log I)^2)$. Section 3.3 describes its functioning. In this paper, we demonstrate this upper-bound and we state the conditions under which it applies.

3. CRATE

Crate (standing for CollaboRATive Editor) is a distributed and decentralized real-time editor running in web browsers. Figure 1 depicts Crate's architecture with four layers: (i) communication: includes the editing session membership mechanism and the information dissemination protocols. (ii) causality: includes the causality tracking structure that guarantees a delivery order of operations reflecting a form of causality.

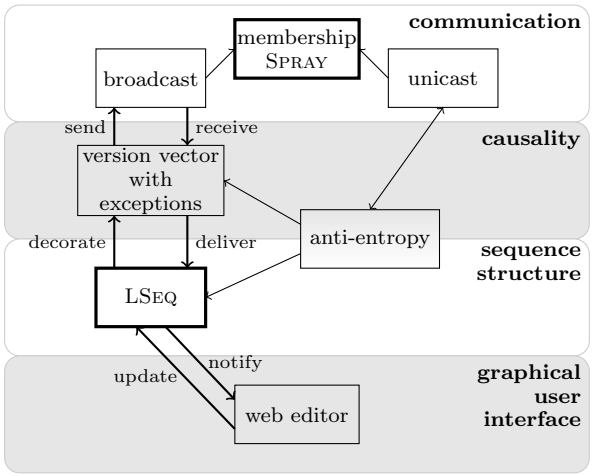

Figure 1: The four layers of Crate's architecture.

(iii) sequence structure: includes the structure that guarantees a global total order among elements of the sequence. (iv) graphical user interface: includes the editor as a graphical entity that users can interact with inside web browsers. The left part of the figure depicts the common process chain: when the user performs an operation on the document, the operation is applied to the shared sequence which creates an LSeq identifier. Then it decorates the result of the operation with causality tracking metadata. Finally, Crate broadcasts it using the neighborhood provided by the Spray random peer sampling protocol. Conversely, when Crate 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 shared sequence which notifies the graphical user interface of the changes. The right part of the figure corresponds to the catch up strategy where a peer may have missed operations due to dropped messages, or simply because the peer worked on offline mode for a while. Therefore, it regularly asks to its neighborhood the missing operations using the differences of version vectors.
Our contributions comprise:

- a fully detailed architecture of a decentralized real-time editor,
- an original usage of Spray applied to the scalable dissemination of messages in the real-time editing context,
- an analysis of space, time, and communication complexities of the replicated sequence structure LSeq.

The rest of this section reviews each layer and details the components inside them.

3.1 Communication

To collaboratively edit a document, users must establish a form of communication between them. It firstly requires accessing the editing session. It secondly requires building a network of communication channels. It thirdly requires the members to use it to spread the changes performed on shared documents.

The first access to an editing session transits a signaling server. A member wishing to share the document provides a *uniform resource locator* (URL) that contains (i) an

address to the CRATE's files with the web application code, (ii) and a reference to the editing session. Note that the latter must be unique (for the signaling server and during the sharing time) and immutable (to one document corresponds one editing session). For instance, the URL `http://chat-wane.github.io/CRATE?snow-crab` targets the Github file server, and asks to the signaling server about the editing session called *snow-crab*. The signaling server chooses at random an available contact among sharers of this editing session. After a round-trip of messages, the first WebRTC connection is established, i.e., a connection from browser-to-browser that enables real-time communication. Since the joiner has access to the editing session through its contact, it does not need the signaling server any longer, hence, it disconnects from it. The member establishes other WebRTC connections using a membership protocol.

The membership protocol, called SPRAY [16], is a random peer sampling protocol [10] the primary target of which is WebRTC. As such, the range of users includes small devices (e.g. smartphones, tablets, etc.) and establishing a connection requires a three-way handshake. These constraints invite to maintain a small number of connections.
Using SPRAY, each member owns a set of neighbors which dynamically grows and shrinks to reflect the network size. Without any global knowledge, (i) it provides each member with a neighborhood of logarithmic size compared to the global network size; (ii) it quickly converges to a topology exposing similarities with random graphs [8]. Among others, (a) it balances the load among members by repeatedly averaging over time the size of neighborhoods pairwise; (b) it becomes robust to random crashes or unexpected departures of members; (c) the shortest average distance to reach all peers stays small.
SPRAY divides the life-cycle of a member into three phases: the joining, the exchanges, and the leaving. They respectively aim to increase, to retain, and to decrease the number of connections following a logarithmic progression.

The information dissemination protocol [5] aims to propagate the changes performed by users on their shared document. Any operation must reach all members (broadcast) to guarantee eventual consistency. When a user performs an operation, CRATE prepares a message including the result of the operation and sends it to the whole network using its neighborhood. Neighbors receiving such message forward it to their own neighbors. Hence, messages reach all participants transitively. To guarantee termination and to limit the flooding, each member forwards each message to their neigbhors only once by using a version vector with exceptions (cf. Section 3.2).
Compared to state-of-art [9, 10, 28], SPRAY provides a neigbhorhood reflecting the network size instead of a constant size neighborhood set at start (commonly oversized to handle large networks). As such, information dissemination protocol on top of SPRAY adapts the load to the network size.
The information dissemination protocol impacts the communication complexity at each member:

$$\mathcal{O}(m. \ln |\mathcal{R}|) \tag{1}$$

where m is the message size determined by the layers below, and $|\mathcal{R}|$ is the number of replicas in the network including both writers and readers of the shared sequence.

3.2 Causality tracking

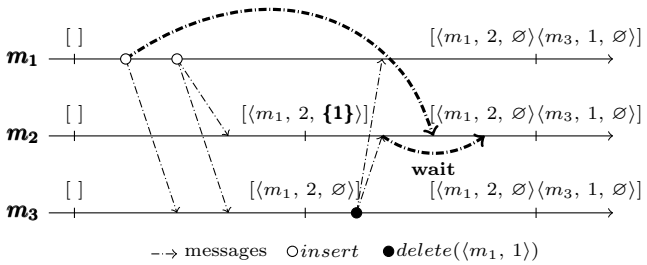

Figure 2: Causality tracking example.

To guarantee the exactly once delivery of operations, and the causal delivery of semantically related operations, CRATE uses a version vector with exceptions [12, 13].

Version vector with exceptions store for each member (i) an integer denoting the maximum counter of operations originated from this site and (ii) a list of integers denoting the exceptions, i.e., the operations known as not received yet.
A unique member identifier along with a monotonically growing counter allows differentiating each operation. Thus, when a member performs a change to its shared document, it increments its local counter. Then it decorates the message with its counter and identifier. Upon reception, CRATE checks in the version vector with exceptions if it already received the operation earlier. In this case, it simply discards the operation. Otherwise, it checks if the operation depends on another one. In this case, CRATE delivers the operation if this other operation is delivered. Otherwise, it puts the operation in a buffer awaiting for the dependency to arrive. Upon delivery, it integrates the operation identifier to the version vector with exceptions.
Figure 2 depicts an editing session involving 3 users. The version vector with exceptions starts empty. Member m_1 inserts two characters in its document and broadcasts the corresponding messages. Member m_3 quickly receives both operations. Since it did not receive these operations before, and since they do not depend of any other operation, it integrates the operation identifiers to its causality structure. It also delivers the operation to the shared sequence structure (cf. Section 3.3). In the meantime, Member m_2 only receives the second operation. Consequently, it marks the first operation of m_1 as exception and still integrates the received operation. Then, m_3 removes the first character inserted by m_1 and broadcasts it. While m_1 delivers the removal immediately, m_2 waits since the targeted operation belongs to the exceptions. Once it receives the missing first operation of m_1, the exception disappears and the delete operation is performed.
The local upper-bound on space complexity is:

$$\mathcal{O}(|\mathcal{W}|) \tag{2}$$

where $|\mathcal{W}|$ is the number of writers, i.e., users who modified the document at least once. Such structure only requires to piggyback the identifiers of operation. Hence, the upper-bound on communication complexity is:

$$\mathcal{O}(o. \ln |\mathcal{R}|) \tag{3}$$

where o is the operation size determined by the shared se-

quence structure (cf. Section 3.3), and the rest is determined by the communication layer (cf. Section 3.1).

The anti-entropy protocol periodically checks if the local replica diverges from another neighbor's one at random. It aims to retrieve missing operations that could have been lost during transmissions. For this purpose, a simple difference between vectors suffices.

For instance, in the prior example depicted by Figure 2, Member m_2 receives the second operation of m_1 before its first. To catch up, m_2 could send its version vector with exceptions to one of its neighbors chosen at random. Assuming that it picks m_3, the latter detects that, compared to its own vector, the remote member missed the first operation of m_1. Then, it sends it back to m_2 along with its own vector. Finally, m_2 follows the normal process for the received operations, and additionnaly merges its vector with the received one.

Such protocol does not require any additional local metadata. However, the communication cost is prohibitively high which encourages to perform this reconciliation protocol with great care:

$$\mathcal{O}(|\mathcal{W}| + |\mathcal{W}| + x.o) \tag{4}$$

where the first $|\mathcal{W}|$ designates the vector contained in the initiating message, and $|\mathcal{W}| + x.o$ the response to this message where $x.o$ are the missing operations.

3.3 Shared sequence

To guarantee eventually consistent [4] replicas of the shared document, CRATE uses a conflict-free replicated data type for sequences [21, 22] with the allocation function LSEQ [14, 15] for its unique and immutable identifiers.

LSEQ stands for polyLogarithmic identifier allocator for SEQuences. It allocates identifiers the size of which is polylogarithmically upper-bounded (compared to the document size) and determined at generation (unique and immutable). Thus, each identifier comprises a path and a globally unique marker called disambiguator. We note the former as a list $[p_1.p_2 \dots p_k]$ where p_k belongs to Integers. The union of paths produces a tree. For instance, the factorization of paths [13.37] and [13.42] produces a tree where the element 13 has two children: 37 and 42. The challenge consists in keeping the identifiers size growth under a sublinear upper-bound on space complexity.

An alphabetical order maintains a dense total order among elements. It uses both path and disambiguators composing the identifier. While the former aims to give a fast and easy way to create and order elements relatively from each other. The objective of the latter is twofold: (i) ordering concurrent operations that happen to have an identical path and (ii) allowing the allocation of new identifiers in-between them. For instance, let us consider the sequence of characters QT with the respective paths [3] and [4], and their respective authors m_x and m_y. When Member m_1 inserts W between those elements, the boundaries [3] and [4] do not allow any additional path of size 1. Assuming a maximum arity of 10, the new path is allocated in the range from [3.0] to [3.9]. In this example, the resulting path is [3.6]. In the meantime, another member m_2 inserts R between Q and T which happens to result in the path [3.6] too. To ensure a total order, disambiguators are associated with each path composed of a monotonically growing counter and a globally unique site identifier. Here, the identifier of Element W is composed of the path [3.6] and disambiguator $[\langle m_x, 1\rangle, \langle m_1, 1\rangle]$ and the Element R is composed of the path [3.6] and disambiguator $[\langle m_x, 1\rangle, \langle m_2, 1\rangle]$. The comparison starts to examine the first concatenations which are equal. Then, the comparison concerns the path of the second level of the tree which are equals too. Hence, it examines the disambiguators. Since $m_x < m_y$, it determines that W is located before R. The resulting sequence is QWRT.

The rest of this section reviews LSEQ's internal components and provides its complexity analysis.

To be application independent , LSEQ uses two sub-allocation strategies designed for antagonist purposes. Indeed, the editing behavior of users is unpredictable, i.e., getting the *a priori* knowledge of where and in which order the operations will be performed is impossible. Hence, instead of characterizing the editing behaviors of users with complex machine learning, LSEQ uses two sub-allocation functions designed to handle trivial editing sequences, knowingly left-to-right editing and right-to-left editing. It assumes that the editing behavior is either one of these two, or a composition of them.

We refer to left-to-right and right-to-left editing as monotonic editing since they come from the repeated insertions of characters at an adjacent position of the previously inserted one.

The allocation functions allocate paths in order to save allocation space for the upcoming operations. In that spirit, the function designed for left-to-right editing allocates paths close of the leftmost available path. Thus, when new characters arrive, presumably at the right of the lastly inserted character, the allocation function still has a large number of available paths. Hence it does not need to increase the depth of the tree.

For instance, assuming a 10-ary tree, a left-to-right allocation function, and a user writing QWERTY starting from Q to Y. When Character Q arrives, the function allocates a path in $[\mathbb{N}_{<10}]$. To leave available paths for future insertions, the function allocates a path close of the bound [0]. In this example, the resulting path is [2]. Then, Character W arrives, the functions allocates a path between Q's path [2], and the virtual boundary [10]. Once again, it chooses a path close of the previous path. It results in the path [5]. Incrementally, we obtain paths [6], [7], [9] for E R T, respectively. When Character Y arrives, there is no room for another path of size 1. Consequently, the depth of the tree increases to receive the new path allocated in $[9.\mathbb{N}_{<10}]$. Identically to prior insertions, the path is allocated close of the previous bound which is [9.0], resulting in [9.1].

Nevertheless, this allocation strategy is not sufficient to handle any editing behavior. When the assumption of left-to-right editing does not hold, performance can be dramatically impacted.

For instance, assuming a 10-ary, a left-to-right allocation function, and a user writing QWERTY starting from Y to Q. As for the prior example, the function allocates [2] for Y. When T arrives, it allocates a path between [0] and [2] resulting in the path [1]. When R arrives, the tree has no room for a new path at depth 1. As consequence, the path is allocated in $[0.\mathbb{N}_{<10}]$. Since it keeps allocating paths close from the previous bound, the resulting path is [0.1]. When E arrives, the tree must grow again etc.

A hash function chooses among the two sub-allocation functions. The choice is made randomly following a uniform

Editing behavior	Space	
	Identifier	Sequence
Random editing	$\mathcal{O}(\log I)$	$\mathcal{O}(I \log I)$
Monotonic editing	$\mathcal{O}((\log I)^2)$	$\mathcal{O}(I \log I)$
Worst case	$\mathcal{O}(I^2)$	$\mathcal{O}(I^2)$

Table 2: Upper-bound on space complexity of LSeq. Where I is the document size.

Editing behavior	Time			
	Local		Remote	
	ins	del	ins	del
Random	$\mathcal{O}(\sqrt{\log I})$	$\mathcal{O}(1)$	$\mathcal{O}(\log I + \sqrt{\log I})$	
Monotonic	$\mathcal{O}(\log I)$	$\mathcal{O}(1)$	$\mathcal{O}((\log I)^2 + \log I)$	
Worst case	$\mathcal{O}(I)$	$\mathcal{O}(1)$	$\mathcal{O}(I)$	

Table 3: Upper-bound on time complexity of LSeq. Where I is the document size.

distribution. As such, it does not favor any editing behavior and provides the independence of the allocation function from any editing behavior.
Using antagonist sub-allocation functions forces all peers to make identical choices [14]. To reach that goal, they all use a similar hash function initialized with a common seed and shared within the document. When a user inserts a new element in the sequence, LSeq firstly processes the depth of the new path, and secondly defers the allocation to the function designated by the hash of the depth.

Scalability is achieved thanks to an exponential tree [2, 3] representing the shared document. Such tree has the particularity that each element has twice as much children as its parent. Hence, paths are sequence of numbers where each concatenation belongs to a subset of Integer the size of which doubles compared to the prior concatenation. For instance, if a path of size 1 is chosen among $[\mathbb{N}_{<32}]$, a path of size 2 is chosen among $[\mathbb{N}_{<32}.\mathbb{N}_{<64}]$ etc.
An exponential tree with a root of maximum arity r has paths of size k chosen among $[\mathbb{N}_{<r}.\mathbb{N}_{<2r} \ldots \mathbb{N}_{<2^{k-1}r}]$. Due to the subsets growth, the binary representation of paths requires one additional bit to encode each concatenation. For instance, a path chosen among $[\mathbb{N}_{<32}]$ is encoded on $log_2(32) = 5$ bits, a path chosen among $[\mathbb{N}_{<32}.\mathbb{N}_{<64}]$ is encoded on $log_2(32) + log_2(64) = 11$ bits, etc.
An exponential tree with a root of maximum arity r has paths of size k encoded on:

$$\sum_{i=0}^{k-1}(\log_2(r) + i) = k^2 + (log_2(r) - \frac{1}{2})k = \mathcal{O}(k^2)\ bits \quad (5)$$

A balanced exponential tree reaching a depth k (i.e., all its branches are filled) holds uptill:

$$\sum_{i=1}^{k} 2^{(i^2+i)/2}\ elements \quad (6)$$

An exponential tree reaching a depth k with only one branch filled per level holds uptill:

$$2^{k+1} - 2\ elements \quad (7)$$

An exponential tree reaching a depth k with only one element per level (worst case) holds uptill:

$$k\ elements \quad (8)$$

The complexity analysis of LSeq stems from this counting. Table 2 shows both the space complexity of each identifier and the space complexity of whole tree structure representing the sequence. When the tree is balanced, its depth is upper-bounded by $\mathcal{O}(\sqrt{\log I})$. Since the identifier encoding grows quadratically compared to the depth of the tree, it results in $\mathcal{O}(\log I)$. From the same reasoning, the identifiers of monotonic editing are polylogarithmic. In the worst case, where the tree has one element per level, identifiers grows quadratically. When the tree structure factorizes identifiers, it results in $\mathcal{O}(I \log I)$ since each element holds a single concatenation of the path the size of which grows logarithmically. In the the worst case, the last identifiers contains the full tree, hence, $\mathcal{O}(I^2)$.
Table 3 shows the time complexity of operations of LSeq divided between the local and remote parts of the optimistic replication. The local insertion is in charge of allocating an identifier. Since, the balanced tree resulting from random editing reaches a depth of $\mathcal{O}(\sqrt{logI})$ and requires as much comparisons to create the new identifiers. When they are integrated, it requires as much binary searches to locate their correct position. A tree filled by monotonic editing grows logarithmically. Hence, the time complexity upper-bound is higher than the balanced structure. Finally, the worst case requires to consider the path of each element to create the new identifiers. Then, at integration, it requires one comparison per level (i.e. per element) to find its location. The local delete operation is only in charge of broadcasting the identifier to all participants, hence, the constant time complexity.

As summary, Figure 3 depicts the functioning of LSeq resulting in two trees after scenarios creating the sequence $QWERTY$: (i) the left-to-right editing sequence [(Q, 0), (W, 1), ...] and (ii) the right-to-left editing sequence [(Y, 0), (T, 0), ...]. In both cases, the exponential tree of LSeq starts with an arity 32 and doubles it at each level. It uses the $frontEditing$ and $endEditing$ sub-allocation functions which are sub-allocation functions designed for right-to-left and left-to-right monotonic editing behaviors. The hash function designates the $frontEditing$ sub-allocation function to the first level of the tree, and the $endEditing$ sub-allocation function to the second level. Since the first level of the tree uses the function $endEditing$, the scenario involving the left-to-right editing sequence results in a tree of depth 1. On the other hand, the antagonist scenario shows LSeq identifiers that quickly reach a level of the tree where the sub-allocation function is designed to handle the right-to-left editing behavior.

3.4 Graphical user interface

To perform their changes on the shared document, users have a visual representation of their respective local copy in their web browser. Since CRDTs for sequences provide functions relying on identifiers rather than indexes, Crate provides a look-up function which gets the identifiers at a designated index, and converse. Thus, when users perform their insertions, Crate gets the identifiers at the targeted position and generates the new identifier. When it receives

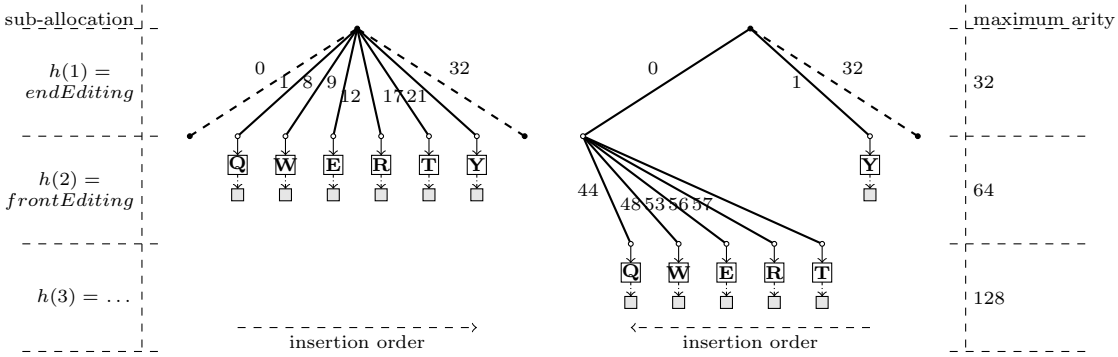

Figure 3: LSeq tree handling two monotonic editing behaviors.

Editing behavior	Time Look-up
Random editing	$\mathcal{O}(2^{\sqrt{\log I}})$
Monotonic editing	$\mathcal{O}(I)$
Worst case	$\mathcal{O}(I)$

Table 4: Upper-bound on time complexity of the look-up on a LSeq structure. Where I is the document size.

an identifier, it firstly inserts it into the tree structure and secondly get its index to notify the view.

Each element in the tree keeps track of its number of children. Such information allows quick withdrawing of ranges of elements. Table 4 shows the time complexity of the look-up operation. When the tree structure is balanced, only a small fraction of the tree need to be browsed to find the index of elements. On the other hand, the structure generated by monotonic editing leads to a linear look-up. Fortunately, the latter case does not happen at local insertions since Crate can lazily store the last generated identifier to create the new one. If a user starts to edit each time at different positions, the structure starts to be balanced, and the look-up becomes more efficient relatively to the document size. Upon reception, since users only have a partial view of the whole document, the look-up of the received operation may fall into a range of element not loaded by the view.

4. EXPERIMENTS

The objective of this experiment section is threefold:

- To show the evolution of the average neighborhood size of members of an editing session. Using Spray, we expect a logarithmic growth compared to the global size of the network.
- To confirm the complexity analysis of LSeq. Concerning the size of identifiers, the common monotonic editing behavior should lead to a polylogarithmic growth compared to the document size. Concerning the time complexities, they should be logarithmic except for the look-up. This latter should be linear after monotonic editing behavior.
- To show 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 number users and number of operations.

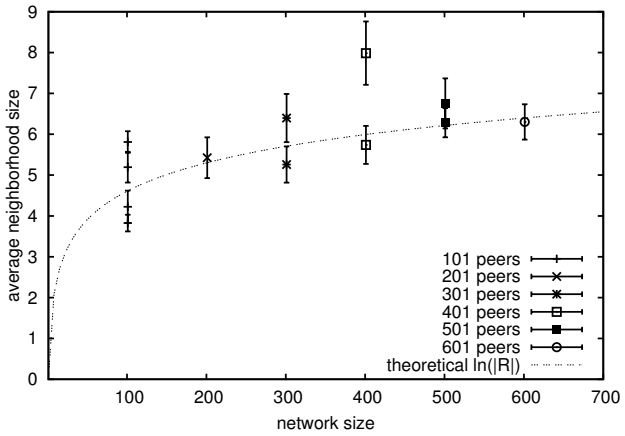

Figure 4: Average neighborhood size of each peer over the global network size.

We conducted the experiments using a Javascript implementation directly usable within modern web browsers[1]. The experiments ran on the *Grid'5000* testbed involving uptil 601 instances of the editor distributed among 101 machines.

Objective: To show the evolution of the average neighborhood size of members of an editing session. The Spray protocol aims to provide a logarithmically scaling neighborhood compared to the global network size. Furthermore, the protocol builds a dynamic network where the load is balanced among members.

Description: During a run, peers dynamically modify their neighborhood. Each peer periodically reports its neighborhood size. Then, we average all these values and measure the variance. The average value defines how connected is a member to the network, impacting (among other) on the information dissemination efficiency. The variance defines

[1]https://github.com/Chat-Wane/CRATE

how spread are the values; a small value means a good load balancing among members. Runs comprise from 101 members to 601 members with 100 members increments, i.e., 6 different runs. The first member creates the editing session which is progressively joined by the other writers (1 joiner per 5 seconds). Each member starts sharing the document as soon as its joining. Hence, outsiders can join the network through one of them chosen at random.

Results: Figure 4 shows the results of this experiment. The figure displays the network size on the x-axis and the average neighborhood size on the y-axis. It shows the average value of each experiment along with the theoretical expectation: a natural logarithm on the global network size. Each average value also carries the variance measurement. As expected, we observe that the average neighborhood size grows logarithmically compared to the size of the network despite being scattered around the theoretical curve. Figure 4 also shows that the variance is small around the average values. Thus, members have an even neighborhood size. In other terms, the load is well-balanced among members when they need to disseminate information to the whole network. At the opposite of centralized solutions, it builds a topology where no member is more important than another in term of connectivity. As such, the network is more resilient to random crashes or unexpected departures.

Reasons: When an outsider wants to join the editing session, it picks a random sharer as first contact. SPRAY assumes that the neighborhood size of this particular member is already logarithmic compared to the network size. Therefore, it uses this assumption to create additional connections that increase logarithmically in number. Yet, the averages neighborhood size does not follow exactly the theoretical expectation curve. Indeed, the number of neighbors of each member strongly depends of the first contact. However, the specific average value is not necessarily reached, especially considering that, while the theoretical expectation is a floating number, the neighborhood size belongs to the integers. This fact also impacts the variance. Indeed, the SPRAY protocol exchanges the neighborhood of member over time. During the exchanges, it aims to balance the neighborhood size of each peer. Yet, the average value may not be an integer value. Therefore, the neighborhood size of the members constantly fluctuates around the average value.

Objective: To confirm the time complexity of the operations of LSEQ. The time complexity part of Table 3 summarizes our expectations.

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. To 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 look-up, the local part of an insertion, the remote part of an insertion, and the remote part of a deletion. 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 does a lot of on-the-fly optimization which we limit in order to show the real time contribution of each operation.

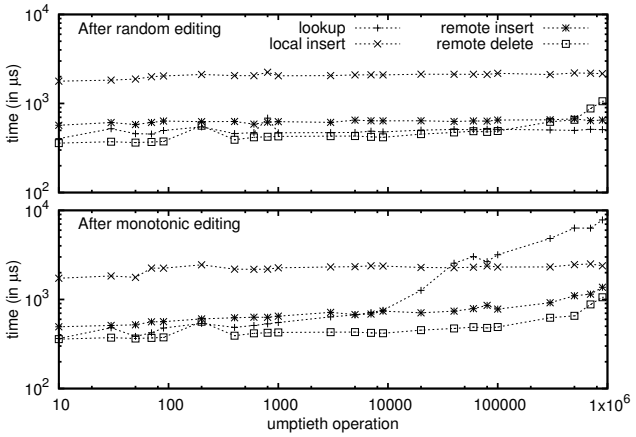

Figure 5: Time performance of operations.

Results: Figure 5 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. 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 on a structure filled with insertions following a monotonic editing behavior. We observer 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 look-up execution time, and a slower growth for both the remote insertions and the remote deletions.

Reasons: 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 look-up 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 look-up 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.

Objective: To confirm the space complexity of the identifiers generated by LSEQ. On the common monotonic editing behavior, we expect a polylogarithmic growth compared to the document size.

Description: The executions follow the previous experiment about the neighborhood size. Thus, the experimentation comprises 6 runs with networks including 101, 201, 301, 401, 501, 601 members. The editing session members are repeatedly inserting new characters at the end of the

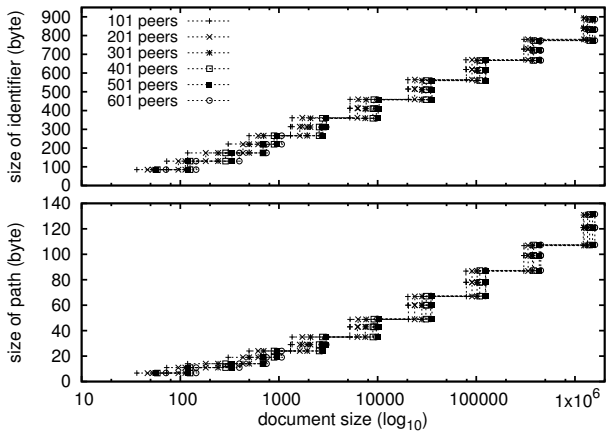

Figure 6: Average size of identifiers.

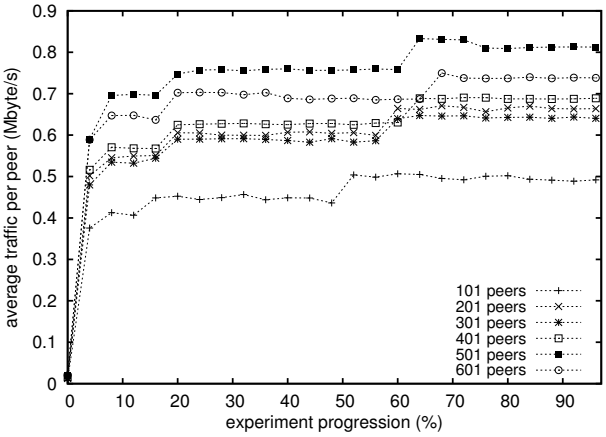

Figure 7: Average traffic per second at each peer during the experiments.

document. The rate of these operation is 100 operations per second distributed among members. Thus, the experiment involving 101 members forces each member to perform 1 insertion per second, while the experiment involving 601 members forces each member to perform 1 insertion every 6 seconds. During the runs, each author reports the byte-size of newly generated identifiers along with the byte-size of its path in the exponential tree. site, counter, starting values of the exponential tree.

Results: Figure 6 shows the result of this experimentation. The x-axis denotes the document size (which is equivalent to the number of insert operations since the members do not perform removals) on a decimal logarithmic scale. Thus, the document grows from empty to above a million characters. The y-axis of the top part of the figure denotes the byte-size of the identifiers while the bottom part of the figure denotes the byte-size of the path only. First, we observe that all plots are very close from each other. Hence, the size of identifiers does not depend of the number of participants. Second, we observe that the identifiers grows polylogarithmically compared to the size of the document. Indeed, the bottom part of Figure 6 shows that the increment step of the generated paths is slowly increasing during the experiment. It empirically exposes a small surlinear behavior when the x-axis is on a logarithmic scale. Hence the polylogarithmic curve. Finally, we can observe that LSeq generates paths the size of which is small comparatively to the whole identifier that includes site identifiers and counters to guarantee uniqueness.

Reasons: The curves of the byte-size are close because the identifiers of LSeq do not depend of whom created it nor the number of authors involved in the document editing. They only depend of the position of insertions which globally corresponds to the editing behavior. In this case, the highlighted editing behavior is a monotonic left-to-right editing, i.e., repeated insertions at the end of the document. This kind of behavior tends to unbalance the tree by filling only one branch per level of the tree. Fortunately, since the tree exponentially grows (each element has twice as much children as its parent), the path growth slowly decreases over insertions. Nevertheless, it costs one additional bit to encode each concatenation of the path, hence the path growth on the bottom part of the figure. The identifiers are much heavier than the path they carry because they also includes a list of pairs to guarantee the uniqueness of identifiers and a global total order. Each member generates its own globally unique identifier (with high probability) without relying on any remote service. As such, they require large identifiers that does not collide with remote members.

Objective: To show 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 number of users and number of operations.

Description: As for prior experiments, this experimentation concerns networks from 101 members to 601 members, each coauthoring a document of over a million of characters by repeatedly inserting new characters at the end of the document. Figure 4 displays the average neighborhood size of members of each run. Figure 6 displays the identifiers size of each run. This experiment measure average traffic of members in Megabyte per second. For the recall, 100 operations are performed per second uniformly distributed among participants.

Results: Figure 7 shows the result of this experiment. The x-axis denotes the experiment progression. The y-axis shows the average outgoing traffic generated by members (in Megabyte per second). As expected, we observe three results: (i) It confirms the results of both prior experiments about the neighborhood size (i.e., a logarithmically growing/shrinking neighborhood size compared to the network size), and the size of identifiers (i.e. a polylogarithmic growth compared to the size of the document). (ii) Being adaptive, the Spray membership protocol grants the ability to disseminate information scaling with the network dimension. (iii) The growth of identifiers constitutes an important part of the traffic as well. Hence, Crate scales in terms of number of users and number of operations.

Reasons: When an outsider joins the network, it injects a number of connections roughly logarithmic compared to the current network size. When it generates an operation, it sends its result to its neighborhood. Since this result (i.e. the identifier) grows polylogarithmically compared to

the document size, and since the neighborhood grows logarithmically compared to the network size, it multiplies the polylogarithm with the logarithm. When a member receives such operation, and it receives it for the first time, it sends it to its neighborhood. Therefore, for each operation performed by any member, the rest of the members will send the result of the operation to all their respective neighborhood once. As such, the network load is balanced among users. Furthermore, it scales in number of editing session members and document size.

5. CONCLUSION

In the context of collaborative editing, this paper proposed an identifier allocation function LSeq for building sequence replicated data structures. The obtained sequences enjoy good space and time complexities. The proof on space complexity demonstrates logarithmic, polylogarithmic, and quadratic for, respectively, random, monotonic, and worst-case insertions. Using this allocation function and interval version vectors to track causality, we developed a distributed collaborative editor called Crate. This editor is fully decentralised and scales in terms of the number of peers, concurrency, and document size. We validated the approach on a setup close of the real life conditions and using extreme parameters: low startup values, high number of insertions, high number of peers and high latency. The experiments confirmed the space complexity of LSeq and the scalability of Crate. As such, it alleviates the issues brought by centralised approaches: single point of failure, privacy issues, economic intelligence issues, limitations in terms of service, etc. Also, it alleviates the scalability issues of decentralised approaches. As such, it can be seen as a serious competitor for current trending editors (e.g. Google Docs, SubEthaEdit, ...), allowing massive collaborative editing without any service providers. More specifically, it opens the field to a new range of distributed applications such as massive editing of online courses, or collaborative reviewing of co-located events, webinars etc.

Acknowledgement

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

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

6. 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, États-Unis, Sept. 2011.

[2] A. Andersson. Faster deterministic sorting and searching in linear space. In *Foundations of Computer Science, 1996. Proceedings., 37th Annual Symposium on*, 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] P. Bailis and A. Ghodsi. Eventual consistency today: Limitations, extensions, and beyond. *Commun. ACM*, 56(5):55–63, May 2013.

[5] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Trans. Comput. Syst.*, 17(2):41–88, May 1999.

[6] B. Charron-Bost. Concerning the size of logical clocks in distributed systems. *Inf. Process. Lett.*, 39(1):11–16, July 1991.

[7] N. Conway. *Language Support for Loosely Consistent Distributed Programming*. PhD thesis, University of California, Berkeley, 2014.

[8] P. Erdős and A. Rényi. On random graphs i. *Publ. Math. Debrecen*, 6:290–297, 1959.

[9] A. Ganesh, A.-M. Kermarrec, and L. Massoulié. Peer-to-peer membership management for gossip-based protocols. *IEEE Transactions on Computers*, 52(2):139–149, Feb 2003.

[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] 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.

[14] 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 Proceedings*. CEUR-WS.org, Sept. 2013.

[15] B. Nédelec, P. Molli, A. MostÃl'faoui, and E. Desmontils. LSEQ: an Adaptive Structure for Sequences in Distributed Collaborative Editing. In ACM, editor, *13th ACM Symposium on Document Engineering*, Sept. 2013.

[16] 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.

[17] 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*, pages 259–268. ACM, 2006.

[18] N. Preguiça, J. M. MarquÃĺs, 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.

[19] 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.

[20] Y. Saito and M. Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, Mar. 2005.

[21] 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.

[22] 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.

[23] M. Suleiman, M. Cart, and J. Ferrié. Serialization of concurrent operations in a distributed collaborative environment. In *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work: The Integration Challenge*, GROUP '97, pages 435–445, New York, NY, USA, 1997. ACM.

[24] C. Sun and C. Ellis. Operational transformation in real-time group editors: issues, algorithms, and achievements. In *Proceedings of the 1998 ACM conference on Computer supported cooperative work*, CSCW '98, pages 59–68, New York, NY, USA, 1998. ACM.

[25] 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 Transactions on Computer-Human Interaction (TOCHI)*, 5(1):63–108, 1998.

[26] 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.

[27] N. Vidot, M. Cart, J. Ferrié, and M. Suleiman. Copies convergence in a distributed real-time collaborative environment. In *Proceedings of the 2000 ACM Conference on Computer Supported Cooperative Work*, CSCW '00, pages 171–180, New York, NY, USA, 2000. ACM.

[28] S. Voulgaris, D. Gavidia, and M. van Steen. Cyclon: Inexpensive membership management for unstructured p2p overlays. *Journal of Network and Systems Management*, 13(2):197–217, 2005.

[29] 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.

[30] 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 Conference on Distributed Computing Systems, 2009*, pages 404–412. IEEE, 2009.

[31] 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.

[32] 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.

[33] 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.

[34] 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.