EAI Endorsed Transactions
on Collaborative Computing

Research Article ICST.ORG

A Highly Concurrent Replicated Data Structure *EAI Endorsed Transactions*★

Mumtaz Ahmad *, Abdessamad Imine [1] and Mahfoud Houari [2]

[1]LORIA/INRIA, Nancy Grand Est, France

[2]Abou-Bekr Belkaïd Université, Tlemcen

Abstract

Well defined concurrent replicated data structure is very important to design collaborative editing system, particularly, certain properties like out-of-order execution of concurrent operations and data convergence. In this paper, we introduce novel linear data structure based on unique identifier scheme required for indexed communication. These identifiers are real numbers holding specific pattern of precision. Based on the uniqueness and the total order of these identifiers, here, we present two concurrency control techniques to achieve high degree of concurrency according to strong and lazy happened-before relations. Our data structure preserves data convergence, yields better performance and avoids overheads as compared to existing approaches.

Received on 26 April 2015; accepted on 30 June 2015; published on 21 December 2015

Keywords: collaborative editing, optimistic replication, concurrent data structures.

doi:10.4108/XX.X.X.XX

1. Introduction

In collaborative editing, a group of individuals may edit shared document simultaneously. Collaborative editing tools are designed to provide environment in which authorized users edit a shared document. A user or participant may know others working on the shared document, and watch the changes/modifications (in real time) performed by other users. Using collaborative editing tools, multiple users are able to make changes at the same time. A group of users could be in the same location or dispersed geographically [1, 2, 8, 9, 11]. Such tools prefer to manipulate shared objects that own linear structure in which each element (e.g. character, line, object or paragraph) is indexed by an identifier (say position number). Each user is supposed to have copy (local) of the shared document so that the availability of data could be ensured. In general, the collaboration is supposed to be performed as follows: each user's operations (e.g. inserting a new element or deleting existing element at certain position) are locally executed in nonblocking manner and then are propagated to all other sites so that all copies of the document could be updated. Due to replication and arbitrary exchange of operations, convergence maintenance in a decentralized manner is a challenging problem. Traditional concurrency control techniques, such as (pessimistic/optimistic) locking and serialization, turned out to be ine ective because they may ensure consistency at the expense of responsiveness and loss of operations [2, 6, 10]. Another technique, called Operational Transformation (OT) is proposed in [2]. Generally, it consists of application-dependent transformation algorithm which modifies the parameters (*e.g.* position numbers) of operations to execute them are propagated to all other sites so that all copies of the document could be updated. Due to replication and arbitrary exchange of operations, convergence maintenance in a decentralized manner is a challenging problem. Traditional concurrency control techniques, such as (pessimistic/optimistic) locking and serialization,

*Corresponding author Email: mumtazzahmed@gmail.com

turned out to be ineffective because they may ensure consistency at the expense of responsiveness and loss of operations [2, 6, 10]. Another technique, called Operational Transformation (OT) is proposed in [2]. Generally, it consists of application-dependent transformation algorithm which modifies the parameters (e.g. position numbers) of operations to execute them regardless of reception order. Identifying elements by position numbers is not suÿcient to ensure data convergence using OT approach. It is also claimed that all previously proposed transformations fail to achieve such convergence [4, 5].

1.1. Related work

A comparison of several approaches to the problem of collaboratively editing a shared text is presented by Ignat et al. [3]. Operational transformation (OT) [2, 9] considers collaborative editing based on non-commutative operations. To this end, OT transforms the arguments of remote operations to take into account the e ects of concurrent executions. To execute concurrent operations in either order, OT requires two correctness conditions which remain diÿcult to satisfy. Imine et al. [4, 5] prove that all previously proposed transformations fail to satisfy these conditions. More recently, Weiss et al. [12, 13] and Preguiça et al. [7] proposed a new data type, called CRDT (Commutative Replicated Data Types), for collaborative editing. Weiss et al. proposed the Logoot CRDT which uses a sparse n-ary tree rather than Treedoc's dense binary tree [7]. In Logoot, a position identifier is a list of (long) unique identifiers, and Logoot does not flatten. Also, Logoot has a high overhead compared to Treedoc. The approaches presented in [7, 12, 13] have some inconveniences:

1. The data structure may grow indefinitely because the deletion operation has no physical effect on the state. Indeed, they mark deleted elements as tombstones in order to converge all replicas.

2. The position identifiers are very long; sometimes the size of identifier can exceed the size of document.

Unlike [7, 12, 13], we use very dense identifiers (real numbers) to uniquely identify all elements inside the shared document and, instead of tree data structure, we describe the shared document by a simple sequence data structure. Moreover, we remove elements without using tombstones. Another recent work [14] uses rational numbers to uniquely identify the elements of the shared document but the size of the document increases during collaboration sessions as the removed elements are only hidden (they use a form of tombstones).

Furthermore, approaches in [7, 12, 13] are to be revised because we observe that proposed algorithms for generating unique position identifiers do not support some critical situations, that leads to cause problems like replica divergence and order preservation over identifiers. As an example, we analyse these two approaches by proposing identifiers exchange scenario

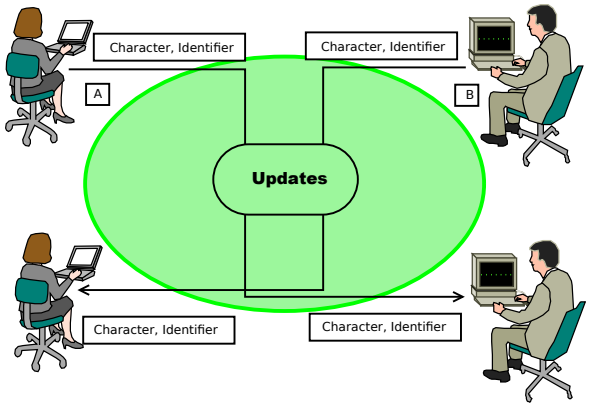

Figure 1. Identifiers exchange scenario

(Figure 1) in which two users A and B insert character concurrently and interchange identifiers.

Let after exchange of points (updates), two identifiers (by Weiss et al. approach [12, 13]) are

$$p = \langle 0, s_A, c \rangle$$

and

$$q = \langle 0, s_B, c \rangle$$

(with s_A and s_B are identifiers of sites for users A and B respectively, such that $s_A < s_B$).
Now, suppose that, user B inserts only one new character between characters having identifiers p and q, then new identifier is $\langle 1, s_B, c \rangle$. This new identifier takes the place as given below which explores an

$$p = \langle 0, s, c \rangle < \langle 1, s, c \rangle < \langle 0, s, c \rangle = q$$

ambiguous order alteration problem in the automatic storage buffer of identifiers.

Similarly, constructing an example for Preguiça et al. approach [7], let user A inserts character 'c' and user B inserts character 'd' and user C inserts character 'e' in a shared document.

One of the possible choices of constructing tree is: node e is the right child of node c and node d is the left child of e. Now let users start concurrent operations between character 'c' and character 'd' then new nodes will be mini-nodes as siblings.

Let dismbiguator of user B be

$$(counter, siteID) = (c1, 2)$$

at the time of insertion of character 'd'. Let he/she intends to insert another character 'f' between c and d, there must be increment in counter, let it be $(c2, 2)$. To make insertion possible, it must holds

$$(c2,\ 2) < (c1,\ 2)$$

but

$$c2 > c1 \text{ (}\textit{normal increment in counter}\text{)}.$$

It means insertion is not possible or problem of order alteration occurred. Further:

$$\text{POSID of d} = 10(0,\ c1,\ 2)$$

$$\text{POSID of f} = 10(0,\ c2,\ 2)$$

$$\text{POSID of c} < \text{POSID of d (assumed.)}$$

Thus, after exchange of identifiers, we observe that, new identifier computed for one of the users does not lie within the previous two identifiers or this new identifier is not comparable by definition (propose by authors) of comparison for two identifiers, consequently it may cause divergence. Treedoc approach is a modified form of approach proposed by Weiss et al. [12, 13] and complicated to implement, for example, to find order, one has to make walk of the tree. Technique to reduce overheads consists in removing information about identifiers that may cause serious problems in retrieving these identifiers from storage buffer. Our approach is simple and avoid these drawbacks.

1.2. Contributions

This paper presents a novel concurrent replicated data structure for collaborative editors, in which, each element is identified by a unique real number. Based on a specific pattern of precision, our unique identifier scheme guarantees order preservation (compatible with the order of the elements) and achieves easily data convergence. Moreover, to each user (or peer), we assign a unique real value as an identifier, also generated under specific precision. A shared document is supposed to be mapped on an interval $I = [a, b]$ with $0 \leq a < b$ for $a, b \in \mathbb{R}$. For the operations performed by users in the network, corresponding unique identifiers are computed over the interval I such that these identifiers are assigned to elements (*e.g.* characters or lines) of the shared document or object. Based on the uniqueness and the total order of our identifiers, we present two concurrency control techniques to achieve high degree of concurrency according to strong and lazy happened-before relations. The first technique relies on time-stamp vectors and it allows the concurrent operations to be executed in either order. This technique is well-suited for collaborative editors where the number of users is fixed. The second technique relies on lazy happened-before relation and it enables us to extend the concurrency even for operations generated by the same user. Using this technique, a collaborative editor can be deployed easily on P2P networks as it can supports dynamic groups where users can leave and join at any time. We validate our data structure with a performance evaluation which shows that our unique identifier scheme is appropriate for linear data structure.

The current manuscript is organized as follows. Section 2 describes the ingredients of our concurrent replicated data structure. Section 3 presents our technique to generate unique identifiers and a view of editing and modifying a document. In Section 4 we suggest two concurrency control techniques in order to use our concurrent data structure. Section 5 gives a performance evaluation of our data structure. Section 1.1 compares with previous work and conclusion is described in Section 6.

2. Introducing New Concurrent Replicated Data Structure

This section consists in introducing new data structure for concurrent editing. It is known that collaborative editors manipulate shared objects that own a linear structure [2, 9, 11]. This structure can be modeled as a sequence of elements from any data type. For instance, an element may be regarded as a character, a paragraph, a page, an XML node, etc. In [11], it has been shown that this linear structure can be easily extended to a range of multimedia documents, such as MicroSoft Word and PowerPoint documents. We consider a shared, replicated document as a sequence of elements and mapped the document to an interval of

real numbers. An element is simply identified by a *real number* in the interval.

2.1. Unique Position Weights

We consider a shared document as an ordered set of elements indexed by unique position identifiers that are real numbers. We call these identifiers *position weights* to distinguish them from the traditional position numbers. The position weights have the following properties:

- Each element in the document (can be thought of as separate storage buffer) has a weight in the corresponding interval used to generate new weights.
- Two elements in the document have two different weights: we can always order two different elements.
- The weight of an element is volatile: any position weight can be removed and inserted again at any time without allowing weight redundancy inside the document.
- Order of position weights is compatible with the order of elements: the set of position weights is totally ordered and consistent with the position numbers of the shared document.

For instance, the position wieghts are denoted with ω, ω', ω_1, ω_2, ... etc. Moreover, position weights or identfiers hold a strict order relation "$<$". To insert a new element between two existing holding position weights ω_1 and ω_2, such that $\omega_1 < \omega_2$, requires only to compute a new position weight ω_{new} in such a way that $\omega_1 < \omega_{new} < \omega_2$. Since the position weights are real numbers that would require theoretically infinite precision. Therefore, their machine representation are carefully used in order to preserve the property (*i.e.*, avoiding weight redundancy). In Section 3, we will present method to compute position weights based on the assumptions described above.

2.2. Shared Data

Shared data structure could be considered as a sequence of pairs (*element*, *weight*) where the elements are ordered by their corresponding weights. Users are able to modify replicas of the data structure by performing any of the following editing operations:
(i) `Insert`($element$, ω_{new}), inserts new element in the document by associating new weight.
(ii) `Remove`(ω_{exist}) removes an element with an existing weight (ω_{exist}) such that this weight is to be recreated next time.

Multiple users are able to edit shared document concurrently and the operations may replayed on each site as soon as received. Unique position weights guarantee the convergence even if the operations performed at different sites in different orders.

3. Practical Implementation

This section describes method to create unique identifiers. These identifiers are real numbers, follow a specific pattern and have low storage overhead as compared to recent available approaches (best to our knowledge) [7, 12, 13].

Definition 1. We define precision as the number of digits following the decimal point of a value (rounded to decimal places/to significant digits), *e.g.*, the precision of the values 12.34600 and 12.345 is 5 and 3 respectively.

3.1. Developing the Basic Rules

In this section we explain basic assumptions made to develop the method.

Mapping a Document to an Interval. To identify each element by a unique position weight, we associate the shared document to an interval I in such a way that 0 and 1 correspond to the begin and the end of the document, respectively. For simplicity we take the initial real interval as $I = [0, 1] = \{x \in \mathbb{R} \mid 0 \leq x \leq 1\}$. Note that the real interval I can be taken up to desired positive length.

Rounding a Value. To create unique identifiers, as a first step, we introduce Function 1 that rounds a given value according to definition 1.

Function 1 takes two parameters, for example, "$Round\,a\,value(x,\, p_r)$" rounds the decimal part of an expression 'x' to the $p_r th$ decimal place and "$round(x)$"

Function 1:(Round a value)

Function: *Round a value*
Input: Value to be rounded, Desired precision
Output: Rounded Value
1 **begin**
2 Let dp := Desired precision;
3 x := Value to be rounded;
4 prec $\longleftarrow 10^{dp}$;
5 $y := \frac{round(x * \text{prec})}{\text{prec}}$;
6 Rounded value $\longleftarrow y$;
7 **return** *Rounded value*;
8 **end**

(see line 5, Function 1) rounds an expression 'x' to the nearest integer.

For instance, $round(-2.4)$ returns -2 and $''Round\ a\ value(15.0766647,\ 4)''$ returns 15.0767 by performing computation over *Maple* 12. We denote $''Round\ a\ value(x,\ p_r)''$ by $\left|x\right|_{p_r}$, *i.e.*, value x rounded over precision p_r by the Function 1.

Let $x \in I$ be a real number. We say that x is correctly rounded to *d-decimal number*, which is denoted by $x^{(d)}$, if the rounded error is $|\epsilon| \leq \frac{1}{2} \times 10^{-d}$. When the rounded error $|\epsilon| = \frac{1}{2} \times 10^{-d}$.

Precision Pattern. To create unique identifiers, we introduce precision control technique keeping the following assumptions.

- $\mathbf{A}_1$: We denote default precision by $''p_d''$ (that commonly taken by programming language) over which we perform computations.

  For example, in Maple, the precision can be fixed by the global variable "Digits" and floating point arithmetic is done in decimal with rounding, so one can set $p_d = Digits$.

- $\mathbf{A}_2$: We denote rounding precision taken for small positive real numbers ϵ by $''p_\epsilon''$, and value for ϵ is taken as user/site's identifier.

- $\mathbf{A}_3$: We denote final rounding precision taken to compute unique identifier by $''p_r''$ and is kept less than both of p_ϵ and p_d. Moreover if we round user/site identifier over $''p_r''$, the resulting value is negligible and has no significant effect on position weights.

By summarizing above assumptions, we get the following inequality

$$p_r < p_\epsilon < p_d \tag{1}$$

and we keep inequality 1 as the basic principle to create unique identifiers and to perform computations accordingly.

3.2. Creating Position Weights for Insertion

Now, suppose that the default precision and the rounding precision are fixed according to the assumptions. To insert an element between two elements p and q, it requires only information about weights of p and q. It is known that the classical midpoint formula computes midpoint of two real values 'say' a and b as $(a + b)/2$. To compute each time different and finite many midpoints for the same interval, it requires certain modifications. As many users compute all possible midpoints over interval I, we are interested in, that:

- Weights computed for a user's modifications must be different from weights computed for all operations performed by other users;
- Set of weights computed computed for operations related to each user must be an ordered set.

For a certain rounding precision (say with a fixed d decimal places) and a user identifier δ created by keeping the required conditions, we modify the classical midpoint formula such that $\forall\ x, y \in I$ with $x < y$, as given below

$$f(x, y) = x + (\frac{y - x}{2})^{(d)} - \delta \tag{2}$$

where $x = x^{(d)}$, $y = y^{(d)}$ (x and y are rounded to d-decimal places). Notice that the weight computed in this way will not be equidistant from the weights x and y due to the subtraction of δ which is a small real value. Function 2 gives how to compute new position weight for a given user (*i.e.*, with user identifier δ).

Example 1. Consider an empty document with corresponding interval $I = [\,0,\ 1\,]$ and rounding precision

Function 2: *middle*(How to compute new point between two weights)

Function: *middle*
Input: $\omega_1, \omega_2 \in I$ (with $\omega_1 < \omega_2$), user identifier δ.
Output: $\omega \in I$.
Requires: $\omega_1 < \omega < \omega_2$

1 **begin**
2 $a \longleftarrow \omega_1^{(d)}$
3 $b \longleftarrow \omega_2^{(d)}$
4 $\nabla \longleftarrow a + (\frac{b-a}{2})^{(d)}$
5 $\omega \longleftarrow \nabla - \delta$
6 **end**
7 **return** ω

with $d = 1$. Let $\delta_1 = 0.004$ and $\delta_2 = 0.00001$ be two user identifiers. According to Function 2, both users δ_1 and δ_2 obtain the same value $\nabla = 0.5$ in Line 4. But, as δ_1 and δ_2 are different and $\delta_1^{(d)} = \delta_2^{(d)} = 0$, each user computes a different ω in Line 5. This line ensures always to compute unique and different position weights as it subtracts δ which is different from one user to another. Indeed, user δ_1 (resp. δ_2) obtains new weight $\omega = 0.496$ (resp. $\omega = 0.49999$) between $\omega_1 = 0$ and $\omega_2 = 1$. Both new weights are different and not equidistant from $\omega_1 = 0$ and $\omega_2 = 1$.

3.3. Editing a Document and Behavior of Identifiers

The main idea of our approach is to provide a non-conflicting execution between concurrent editing operations. We ensure eventual consistency (*i.e.*, the final state of replicas is identical at all sites), provided that every site executes every operation in an order consistent with some happened-before order (See Section 4 which presents two forms of happened-before order). When the data type is a sequence of elements (such as a text document), the out-of-order execution between insertions in the sequence can be obtained with a unique and totally ordered identifiers for each element. Our approach is based on associating a position weight to each element. These position weights are unique and totally ordered. Figure 2 presents an overview of, how a single user make insertion, corresponding to unique identifiers, starting with an empty document. First position weight is

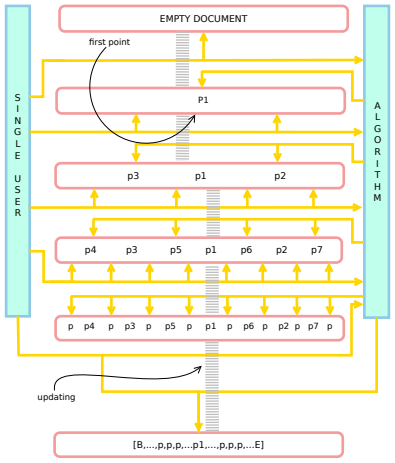

Figure 2. Situation for single user.

presented by p_1, then all other weights are presented by $p's$. Note that, new position weights could be on the left or right side of the existing weight. In Figure 2, 'updating' denotes updates of the document after each modification (insertion).

Example 2. Suppose that multiple users are participating in collaborative edition, we explain in this example, how the corresponding identifiers are generated. Such situation is described in figure 3. In the model that we proposed, a user can modify the document by inserting or removing elements (*e.g.*, lines or paragraphs). To perform this task, corresponding weights are created and removed. Millions of weights can be created with chosen value of user identifier (*i.e.*, δ) and by selecting appropriate rounding precision. These weights can be created locally as well as based on the remote identifiers of the elements during the exchange of elements between the users in the network. To compute new weights, same criteria has to follow for all users in the network. Suppose that a shared document marked with 'Beg' and 'End' mapped to an interval $[0, 1]$ and 'updating' action updates modification to all participants. Suppose $d = 1$ for the rounding precision. Suppose that there are three users U_1 (assigned a user identifier 0.007), U_2 (assigned a user identifier 0.004) and U_3 (assigned a user identifier 0.001) start editing the shared document at three different sites (site 1, site 2 and site 3) as shown in the Figure 3. Let user U_3 inserts first character 'A' in the empty document then the first

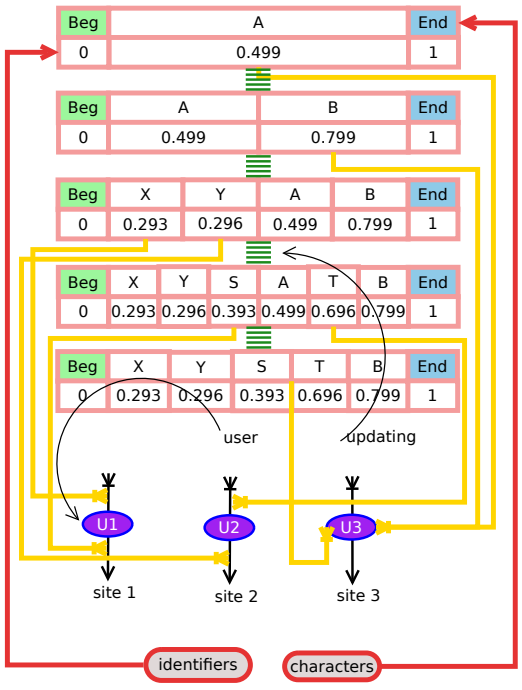

Figure 3. Multiple users are in action

associated weight 0.499 is computed. Again, user U_3 inserts a second character 'B' between character 'A' and 'End', this time the corresponding weight between 0.499 and 1 is computed as 0.799. Now, document is updated and at each site sequence of characters 'AB' is appeared. Let user U_1 and user U_2 intend to insert two characters 'X' and 'Y' between 'Beg' and 'A' concurrently. Notice that weights for both characters are to be computed between 0 and 0.499 and they are 0.293 and 0.296 respectively. Again, updates are performed and each user have sequence of characters 'XYAB'. Now, user U_1 inserts character 'S' between 'Y' and 'A', weight for 'S', 0.393 is computed using neighboring weights of characters 'Y' and 'A'. Similar insertion of character 'T' between characters 'A' and 'B' by user U_2 is attempted. Weight for 'T', 0.696 is computed using neighboring weights of characters 'A' and 'B'. At the same time user U_3 removes character 'A'. Updates are executed and each site has a sequence of characters 'XYSTB'. Notice that removal of 'A' does not affect the insertion by other users because of each site has its own replicas.

4. Two Concurrency Control Techniques

A stable state in a collaborative editor is achieved when all generated editing operations have been performed at all sites. Let o_1 and o_2 be two editing operations. A collaborative editor is *consistent* iff it satisfies the following properties:

- *Causality preservation:* if o_1 happens before o_2 then o_1 is executed before o_2 at all sites.
- *Convergence:* when all sites have performed the same set of operations, the copies of the shared document are identical.

To satisfy the above consistency criteria, we present in this section two concurrency control techniques in order to manipulate our concurrent replicated data structure. Each technique implements strong/lazy happened-before relation and ensures the convergence property.

4.1. Concurrency Control with Strong Causality Relation

Let o_1 and o_2 be operations generated at sites i and j, respectively. We say that o_2 *causally depends* on o_1, denoted $o_1 \rightarrow o_2$, iff:

- $i = j$ and o_1 was generated before o_2; or,
- $i \neq j$ and the execution of o_1 at site j has happened before the generation of o_2.

Operations o_1 and o_2 are said to be *concurrent*, denoted by $o_1 \parallel o_2$, iff neither $o_1 \rightarrow o_2$ nor $o_2 \rightarrow o_1$. As a long established convention in collaborative editors [2, 9], the *time stamp vectors* are used to determine the causality and concurrency relations between operations. A time-stamp vector is associated with each site and each generated operation. Every time-stamp is a vector of integers with a number of entries equal to the number of sites. For a site j, each entry $V_j[i]$ returns the number of operations generated at site i that have been already executed on site j. When an operation o is generated at site i, a copy V_o of V_i is associated with o before its broadcast to other sites. $V_i[i]$ is then incremented by 1. Once o is received at site j, if the local vector V_j "dominates"[1] V_o, then o is

[1] We say that V_1 dominates V_2 iff $\forall\, i$, $V_1[i] \geq V_2[i]$.

ready to be executed on site j. In this case, $V_j[i]$ will be incremented by 1 after the execution of o. Otherwise, the o's execution is delayed.
Let V_{o_1} and V_{o_2} be time-stamp vectors of o_1 and o_2, respectively. Using these time-stamp vectors, the causality and concurrency relations are defined as follows:

- $o_1 \rightarrow o_2$ iff $V_{o_1}[i] < V_{o_2}[j]$;
- $o_1 \parallel o_2$ iff $V_{o_1}[i] \geq V_{o_2}[j]$ and $V_{o_2}[j] \geq V_{o_1}[j]$.

Given our concurrent data structure to describe a shared document, all concurrent editing operations (`Insert` and `Remove` operations) are executed in any order provided that the above causality relation is respected. Unfortunately, the time-stamp vectors do not enable dynamic groups (*i.e.*, users may join or leave the group at any time) since each time-stamp is a vector of integers with the number of entries is equal to the number of users.

4.2. Concurrency Control with Lazy Causality Relation

The time-stamp vectors implement a false causality relation. Indeed, some editing operations performed by a user could be permuted without effect on the state of the document. For instance, two successive insertions can be executed in any order because their position weights are unique totally ordered. As no two different users produce the same position weight, an insert must happen-before a removal with the same position weight. They can never be concurrent. Consequently, the only causality relation to be preserved is $\texttt{Insert}(element, \omega_e) \rightarrow \texttt{Remove}(\omega_e)$. All other operations, not constrained by this relation, can be performed in either order. Using only this simple causality relation enables us to deploy a collaborative editor on P2P networks for supporting dynamic groups.

Next, we present two scenarios which show that exchanging naively editing operations may cause consistency issues. For each scenario, we illustrate the consistency problem and we sketch a solution for this problem.

First Scenario. Consider the scenario given in Figure 4.(a) where two users remove simultaneously the same character e added by operation o_1. When the remove operation o_2 (resp. o_3) arrives at site 2 (resp. site 1), it cannot performed because the position weight ω_e does not exist. Should we consider o_2 and o_3 as not causally ready? If yes, o_2 and o_3 will be never causally ready as we have no information whether or not ω_e has been added. Consequently, the waiting queue would increase drastically.

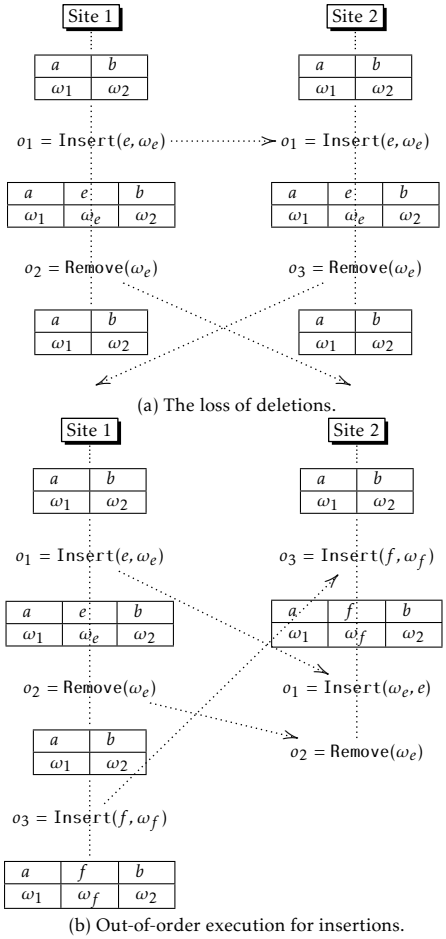

Figure 4. Consistency Issues.

To overcome this problem, we propose that each site maintain a log where the local remove operations will be stored. In this way, o_2 and o_3 are logged respectively at sites 1 and 2. When o_3 arrives, it is easy to verify inside the local log of site 1 that ω_e has been already removed. Thus, o_3 can be ignored. Similarly, the same processing is carried out for o_2 at site 2.

Second Scenario. As shown in Figure 4.(b), suppose a user adds character "e" between "a" and "b" (operation

o_1), removes it (operation o_2) and next adds at the same position another character f (operation o_3). What happens if o_3 arrives before o_1 at site 2? Note that the position weights ω_e and ω_f are computed by the same user (or site) on the same adjacent position weights, ω_1 and ω_2. In this case, ω_e and ω_f are equal. This redundancy breaks the structure of our shared document. Indeed, in this case, the same position weight indexes two different characters "e" and "f". Moreover, at site 2, o_3 may remove either "e" or "f". This can lead to inconsistency situation.

To still create different position weights, we propose to add a monotonically increasing counter that is incremented by each local insertion operation. Hence, we redefine the set of position weights as $\mathcal{W} = \{(\omega, c) \mid \omega \in I \text{ and } c \in \mathbb{N}\}$ the set of which are totally ordered by the relation $\prec$ such that $(\omega_1, c_1) \prec (\omega_2, c_2)$ iff

- $\omega_1 < \omega_2$, or;
- $\omega_1 = \omega_2$ and $c_1 < c_2$.

Note that the first component ω will be still computed according to Function 2. In this case, characters "e" and "f", generated at site 1, will have two different weight positions (the same ω but with different counters).

5. Performance Evaluation

To verify the effectiveness of our approach, an experimentation study has been conducted using a text document as a shared data with various sizes. In collaborative editors, shared data is represented to user as a linear structure and insertion and deletion of elements are based on their position numbers in this structure (*i.e* data view). In our case, we store for each element in the shared Based on the uniqueness and the total order of our identifiers, we present two concurrency control techniques to achieve a high degree of concurrency according to strong and lazy happened-before relations. The first technique relies on time-stamp vectors and it allows the concurrent operations to be executed in either order. This technique is well-suited for collaborative editors where the number of users is fixed. The second technique relies on lazy happened-before relation and it enables us to extend the concurrency even for operations generated by the same user. Using this technique, a collaborative editor can be deployed easily on P2P networks as it can supports dynamic groups where users can leave and join at any time. We validate our data structure with a performance evaluation which shows that our unique identifier scheme is appropriate for linear data structure. data its position weight. To choose the adequate structure, we investigate the implementation of two versions of our data structure, either based on linear structure or on binary tree structure. We denote by n the size of the current state, and by *list* and *tree* the structures used in both versions.

5.1. Local Insertion / Local Deletion

Whatever the used structure, the following operations are performed to insert a new element e (delete an existing one) at position i in the view:

- Search the adjacent position weights ω_i and ω_{i+1} in the case of insertion, and the position weight ω_e corresponding to the position i of the element to be removed.
- Execute the operation (`Insert`(e, ω_e) (or `Remove`(ω_e).
- update the view.

Since insertion/deletion is performed over a given position in the view, a linear structure has the advantage that adjacent *position weights* ω_i and ω_{i+1} (or ω_e in the case of deletion) are returned by $list[i]$ and $list[i+1]$ (or by $list[i]$), respectively, in a constant time. However when using a tree structure, to return either the adjacent weights ω_i and ω_{i+1} of the element to insert (or the ω_e of the element to be removed), the ascending list of all position weights stored in the tree must be computed in $O(n)$ time to extract the required weights.

The second step consists in inserting/deleting position weight in $list/tree$. The third step corresponds to inserting/deleting element e in the shared data and to refresh user view. The complexity time of the third step is $O(n)$ in each structure. But operation `Insert`(e, ω_e)/`Remove`(ω_e) in the second step is executed in $O(n)$ over $list$ and in $O(log(n))$ over $tree$.

5.2. Remote Insertion / Remote Deletion

At the reception of a remote insertion/deletion operation of element e (with weight ω_e), we proceed as follow:

- Search the position of ω_e in $list/tree$.
- Execute the operation (`Insert`(e,ω_e) (or `Remove`(ω_e).
- Update the view.

Since $list$ is an ordered set of weights, thus for a given element weight ω_e its position in $list$ is computed in $O(log(n))$ time. It is used first to insert/remove element in $list$ and also to update the view. In case of tree structure, the correspondence between a given weight ω_e and its position is found by first computing the ascending list of all weights stored in the $tree$ ($O(n)$ time), and next the position is returned by a binary search executed in $O(log(n))$ time over the computed list.

	Best-case		Worst-case	
	List	Tree	List	Tree
Local operation	*cst*	$n + log(n)$	$2.n$	$2.n + log(n)$
Remote operation	$log(n)$	$n + 2.log(n)$	$2.n + log(n)$	$2.(n + log(n))$

Table 1. Linear/Tree structure complexity time.

In Table1 we summarize the overall of all complexity time over $list/tree$ in the cases of $local/remote$ operations and either in the *worst-case* (inserting element at the beginning of the view/deleting the first element) or in the *best-case* (inserting element at the end of the view/deleting the last element).

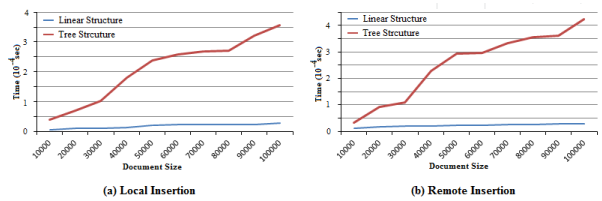

Figure 5. Local/Remote operation evaluation time.

In each structure, all local operations (resp. remote operations) have the same complexity time. In Figure 5 we present the evaluation of the *insertion* over a text document with size varying from 10000 to 100000 elements (we take an element as a small paragraph). It is clear that a linear structure based-implementation is more efficient. Tree structure offers good performance in term of search, insertion and deletion of data defined with its position weight (case of remote operation). There operations are often performed in $O|log(n)|$ time. However, the poor performance encountered in a tree structure is due to the correspondence, for a given element, between its associated position weight and its position in the view. As approaches in [7, 12, 13] are based on tree structure, it is clear that their implementations will present poor performance.

6. Conclusion

This paper presented a new data structure that is well-suited for linear structure-based shared documents (such as text documents) in collaborative editors. To ensure a high degree of concurrency, we proposed a new technique to uniquely identify elements inside the shared document. These identifiers are simply real numbers which are manipulated under a precision control in order to avoid the problem of infinite precision. This technique is quite simple and guarantees the uniqueness of these identifiers. According to two (strong and lazy) forms of happened-before relation, we proposed two concurrency control procedures: the first procedure allows the concurrent operations to be executed in either order. The second one enables us to extend the concurrency even for operations generated by the same user as our identifiers are unique and totally ordered. We performed a performance evaluation which shows that our unique identifier scheme is well-suited for linear data structure. In future work, we intend to investigate the impact of our work when undoing operations. Furthermore, we plan to extend our unique identifier scheme to other data structures such as trees and graphs.

References

[1] Bharat Veer Bedi, Marc Stanley Carter, Martin J Gale, Lucas William Partridge, and Andrew James Stanford-Clark. Method and system for collaborative editing of a

document, June 14 2011. US Patent 7,962,853.

[2] Clarence A. Ellis and Simon J. Gibbs. Concurrency Control in Groupware Systems. In *SIGMOD Conference*, volume 18, pages 399–407, 1989.

[3] Claudia-Lavinia Ignat, Gérald Oster, Pascal Molli, Michèle Cart, Jean Ferrié, Anne-Marie Kermarrec, Pierre Sutra, Marc Shapiro, Lamia Benmouffok, Jean-Michel Busca, and Rachid Guerraoui. A comparison of optimistic approaches to collaborative editing of Wiki pages. In *Collaborative Comp.: Networking, Apps. and Worksharing (CollaborateCom)*, number 3, White Plains, NY, USA, November 2007.

[4] Abdessamad Imine, Pascal Molli, Gérald Oster, and Michaël Rusinowitch. Proving correctness of transformation functions in real-time groupware. In Kari Kuutti, Eija Helena Karsten, Geraldine Fitzpatrick, Paul Dourish, and Kjeld Schmidt, editors, *ECSCW*, pages 277–293. Springer, 2003.

[5] Abdessamad Imine, Michaël Rusinowitch, Gérald Oster, and Pascal Molli. Formal design and verification of operational transformation algorithms for copies convergence. *Theor. Comput. Sci.*, 351(2):167–183, 2006.

[6] Du Li and Rui Li. Ensuring Content Intention Consistency in Real-Time Group Editors. In *IEEE ICDCS'04*, Tokyo, Japan, March 2004.

[7] Nuno Preguiça, Joan Manuel Marquès, Marc Shapiro, and Mihai Leţia. A commutative replicated data type for cooperative editing. *Distributed Computing Systems, International Conference on*, 0:395–403, 2009.

[8] Haifeng Shen and Mark Reilly. Personalized multi-user view and content synchronization and retrieval in real-time mobile social software applications. *Journal of Computer and System Sciences*, 78(4):1185 – 1203, 2012.

[9] Chengzheng Sun and Clarence (Skip) Ellis. Operational transformation in real-time group editors: Issues, algorithms, and achievements, 1998.

[10] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving Convergence, Causality-preservation and Intention-preservation in real-time Cooperative Editing Systems. *ACM Trans. Comput.-Hum. Interact.*, 5(1):63–108, March 1998.

[11] Chengzheng Sun, Steven Xia, David Sun, David Chen, Haifeng Shen, and Wentong Cai. Transparent adaptation of single-user applications for multi-user real-time collaboration. *ACM Trans. Comput.-Hum. Interact.*, 13(4):531–582, 2006.

[12] Stéphane Weiss, Pascal Urso, and Pascal Molli. Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks. In *ICDCS*, pages 404–412. IEEE Computer Society, 2009.

[13] Stephane Weiss, Pascal Urso, and Pascal Molli. Logoot-undo: Distributed collaborative editing system on p2p networks. *IEEE Transactions on Parallel and Distributed Systems*, 21:1162–1174, 2010.

[14] Qin Wu, Calton Pu, and Joao Eduardo Ferreira. A Partial Persistent Data Structure to Support Consistency in Real-Time Collaborative Editing. In *ICDE Conference*, volume 18, pages 399–407, 2010.