Supporting Adaptable Granularity of Changes for Massive-scale Collaborative Editing

Luc André, Stéphane Martin and Gérald Oster
Université de Lorraine, F-54000
CNRS, F-54500
Inria, F-54600
Email: {luc.andre,stephane.martin,gerald.oster}@loria.fr

Claudia-Lavinia Ignat
Inria, F-54600
Université de Lorraine, F-54000
CNRS, F-54500
Email: claudia.ignat@inria.fr

***Abstract*—Since the Web 2.0 era, the Internet is a huge content editing place in which users contribute to the content they browse. Users do not just edit the content but they collaborate on this content. Such shared content can be edited by thousands of people. However, current consistency maintenance algorithms seem not to be adapted to massive collaborative updating. Shared data is usually fragmented into smaller atomic elements that can only be added or removed. Coarse-grained data leads to the possibility of conflicting updates while fine-grained data requires more metadata. In this paper we offer a solution for handling an adaptable granularity for shared data that overcomes the limitations of fixed-grained data approaches. Our approach defines data at a coarse granularity when it is created and refines its granularity only for facing possible conflicting updates on this data. We exhibit three implementations of our algorithm and compare their performances with other algorithms in various scenarios.**

Keywords*—*Collaborative editing, consistency maintenance, optimistic replication, Computer-supported collaborative work

I. Introduction

Mass collaboration involves hundreds to thousands of people working towards a common goal. Wikipedia is one of the most emblematic figure of mass collaboration. When breaking news happens, it is common that hundreds of people contribute to the same related Wikipedia pages in a very short amount of time. In such a situation a real-time collaborative editor would improve the collaboration as it allows contributors to edit simultaneously the content. It would avoid the recurring concurrent edits problems that arise when several users edit the same page at the same time: the first contribution is saved while others have to be manually merged by their contributors with the new content; afterwards the merged content is resubmitted again; this process is recursively repeated until no contribution is saved in the meantime. In real-time collaboration changes of one user are automatically integrated into the shared data and immediately pushed to the other users.

Unfortunately current real-time collaborative editing technology is not ready to support that scale of collaboration – large amount of contributors, high velocity of changes. – This is partly due to the underlying replication mechanism which has to merge the concurrent changes in a single content and replicate it in real-time amongst the contributors. Existing replication mechanisms for collaborative editing consider that the shared content is made of a sequence of elements whose granularity is fixed. For instance a text content is generally seen as a sequence of characters or a sequence of lines. A coarse granularity of changes allows to keep low the overhead generated by additional structural metadata that maintains ordering between elements. But when the content of such an element is updated, this element has to be deleted and a new one is inserted. This can lead to duplicated content when two concurrent updates are done on the same element. In this case additional user actions are required to merge this duplication. On the other hand, a finer granularity of changes allows to compute finer merging of concurrent changes but implies a higher overhead. Additionally, the more elements the document is made of, the more computation is needed to apply and merge user changes.

In this paper we propose a new replication mechanism for merging content made of elements with variable granularity. It avoids the case where content is automatically duplicated during merging of concurrent modifications since it allows to insert an element within an existing element. As the content is generally made of coarser elements than in other approaches, our proposed approach performs better and has a smaller memory footprint than existing approaches.

This paper is organized as follows. Section II gives a comprehensive view of existing replication mechanisms suitable for real-time collaborative editing and points out their limitations. Section III presents the model and the algorithms used by our approach. Section IV discusses the correctness of our approach. Section V exposes performance of our approach by means of simulations. Finally, Section VI contains concluding remarks.

II. Background and Related Work

In order to enable high availability and performance in collaborative editing data is replicated. Each user possesses a copy of the edited document. Rather than using traditional pessimistic mechanisms relying on locking or priority-based policies, collaborative editing employs optimistic replication [1] mechanisms. Users can freely edit the document copies at any moment. When a user edits her own copy, she first modifies it locally and then the related changes are propagated to the copies of other users. These changes are integrated when received at remote copies. Optimistic replication lets copies diverge temporarily while operations are still transiting the network but eventually, when all changes are

COLLABORATECOM 2013, October 20-23, Austin, United States

DOI 10.4108/icst.collaboratecom.2013.254123

integrated, it ensures copies convergence. Two main families of optimistic replication algorithms have been proposed: operational transformation and conflict-free replicated data types. In this section we present the main features of these families of algorithms by highlighting their advantages and limitations.

A. Operational Transformation

Operational transformation approach [2] is a family of operation-based optimistic replication mechanisms. It has been proposed in the context of real-time collaborative editing and has been widely studied in the literature [3]–[5]. In this approach local operations are executed immediately on the document copy after they have been generated and remote operations are transformed against concurrent operations upon their reception at other copies. The transformations should be performed in such a manner that the intentions [4] of the users are preserved and the copies of the document converge in the end.

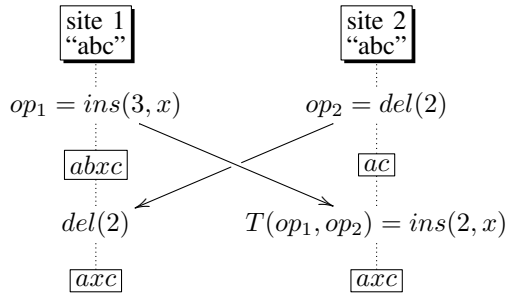


Figure 1: Operational transformation - copies convergence by transforming.

Operational transformation approaches ensure the causality property among the operations. The causal relation between two operations was defined based on the “happen-before” relation [6]: an operation O_a is said to causally precede operation O_b if the site that generated O_b executed O_a before the generation of O_b. The causality property ensures that if a site executed operation O_a before it generated O_b, then all the other sites should execute O_a before O_b.

The main issue of the operational transformation approach is scalability. The transformation of a remote operation is computed regarding all concurrent operations, i.e. operations that changed the state of the document between the remote generation and the local reception. Fig. 1 illustrates an example of two concurrent operations. The insertion is generated at site 1 when the document text is “abc”, but when received at site 2 the text is “ac”. The state of the document is not the same at the generation at site 1 and the reception at site 2, so the received operation needs to be transformed with every operation responsible of this change. Moreover, when an operation is transformed regarding another operation the two operations have to be defined on the same document state. In order to satisfy this condition for transformation additional computations need to be performed. Several algorithms define an exclusion transformation that excludes an operation from the context of another operation [4] or a transposition function that changes the execution order of two operations and transforms them such that the same effect is obtained as if the operations were executed in their initial order and initial form [5].

The more active users are typing, the more concurrent operations are present and the more time is needed to handle each operation. Additionally, in decentralized versions of operational transformation approaches, it is hard to reduce the history size by a garbage collection mechanism. In literature, the fully decentralized algorithms require to satisfy two correctness properties that are not friendly to ensure [7]. For instance, the majority of operational transformation algorithms make use of the position of the character for text editing. In this context, one way to ensure these properties is to keep traces of the deleted elements in the document as in [8], which is quite simple to achieve but constraining in memory. In a more general case, a vector clock [9] usually needs to be sent together with each operation to be able to detect concurrency and causality. This vector contains one entry per user session which can be costly if a large amount of users are collaborating.

B. Conflict-free Replicated Data Types

More recently, a new approach called Conflict-free[1] Replicated Data Type (CRDT) has been proposed [10]–[12]. The purpose of such family of algorithms is to define abstract data types providing a commutative set of operations to update the data. Hence, transformations are not required anymore to integrate concurrent changes since concurrent operations commute - they can be applied in any order. Several algorithms have been proposed; however, in what follows we restrict our discussions to the most representative ones.

WOOT [10] is the first created CDRT algorithm. Its underlying principle is that in a text document, every character is located between two others, so to build a document one just needs to know the neighbours of each character. These neighbours are called $Previous$ and $Next$. To insert a new element B between two existing elements A and D, element B is created, with $Previous(B) = A$ and $Next(B) = D$. Then to insert C between B and D, $Previous(C) = B$ and $Next(C) = D$. Even if now the element next to B is C, $Next(B)$ remains unchanged. This is not an issue, B and D just need to be ordered before C is. Problems arise when concurrent edits of the document happen. If two users insert a string of characters between the same elements at the same time, the deterministic algorithm sorting all these new characters is quite complex and time-consuming. This is emphasized by the fact that, as each element is characterized by other elements, one cannot completely remove elements from the document - they are turned invisible and said becoming *tombstones*. So even inserting between two apparently contiguous elements might trigger the sorting algorithm. The tombstone concept is also a problem itself as the length of the document can only grow over time. It is worth to note that optimized versions (named WOOTH and WOOTO) of this algorithm have been proposed [13], [14]. The causality property is replaced by preconditions: the necessity of existence of neighbours in the case of an insertion and of the element itself in the case of a deletion. As a result the

[1]This approach is also referred as Commutative Replicated Data Types

"happen-before" relation between operations can be violated in some cases.

Based on the idea that a text document is a set of ordered characters, another algorithm named LOGOOT [12] has been proposed. LOGOOT associates each character with a unique identifier, the set of identifiers being *densely ordered* by a specific relation[2]. To insert a new character at a precise position in the document the character is inserted with a new suitable identifier created on purpose. To remove an existing character the corresponding character identified has to be located and then the related character is removed with its associated identifier. In this algorithm, each identifier is a list of triples of integers. The first integer is a priority number, the second one references the user who generated the element – called the unique site identifier –, and the third one is equal to the value of the user's logical clock when this element was created – the clock is incremented each time an operation is generated –. The priority number is used to sort the characters, the site identifier to break ties if two users generated both an element with the same priority, and the clock value, together with the site identifier, ensures that triples are unique. If it is not possible to create a new triple between two others (for instance to insert a character between two existing characters), the new identifier will contain additional triples. This is similar to entries in a dictionary where one can find the word "busy" between the word "bus" and the word "but" even if the letter 't' comes straight after the letter 's' in the alphabet. The main issue of LOGOOT is that a huge amount of insertions in the same part of the document might lead to identifiers formed by very large lists of triples that are memory costly [14].

TREEDOC [11] offers an alternative solution to compute identifiers by modeling the document as a binary tree. Each node contains a character, and the path from the root of the tree to the node is the identifier of the character. The left child of a node was inserted just before the node, and the right child after it. This leads to a more compact representation of identifiers and to a natural dichotomous search, but paths tend to grow fast when the tree is unbalanced. For instance, the tree becomes unbalanced when insertions are performed at the end of the text. Such a case occurs quite frequently as users naturally write continuously from left to right. Moreover, some tombstones are needed as it is not possible to remove one node from the document if this node still has children. However, once a node does not have any children anymore, it can be safely removed. In the case of a concurrent insertion in a node's child subtree during its deletion, it is sufficient to reintroduce the needed nodes as tombstones – as they are known from the identifier which is the path – when receiving the insertion operation.

LOGOOT and TREEDOC do not propose any solution for ensuring causality. They assume that the proposed algorithm can be combined with any mechanism for ensuring causality.

All CRDT algorithms from the literature suffer from the problem of the granularity: a fine granularity (i.e. characters) can lead to more memory consumption and interlaced edits, while a coarse one (lines or paragraphs) is less costly in term of memory but can create duplicated content as an element cannot be updated. Indeed, an update is usually turned into the deletion of the former element followed by the insertion of the updated one. Hence, several concurrent updates of the same element duplicate this element.

An attempt to solve this problem was proposed in [15]. The granularity used is a string of characters. This work additionally introduces offsets as a way to differentiate the parts of a string and to insert a new element into it. In this model, strings have one identifier and one integer called offset, representing the position of the left end of the string in the initial insertion. Strings indeed can be split. In this case several elements with the same identifier are in the document, but their offsets differ. An appropriate property of this technique is that a split of a string does not depend on the user who performed it. If two users concurrently split the same string at the same position, only one new element is created during the split. Consequently, content is not duplicated. This work uses a WOOT-like way to sort elements, using two left and right references to others elements. It theoretically has the same performance problems as WOOT, namely complex integration mechanism and the need to keep tombstones in the document. Besides, this approach is not suitable for real-time editing. Indeed, when users are typing in, characters are usually generated one by one giving the other users a kind of feedback that some edits are in progress. But, in this approach, all characters are buffered locally in order that only one single block of characters is sent to other users. There is no possible way to append characters to an existing string afterwards – without generating additional metadata/identifiers –

C. Summary

Operational transformation approaches do not scale well and nearly all CRDT have issues with the granularity of edits. The sole known algorithm that handles multiple granularities has the same limitations as the algorithm on which it is based on.

Our aim is to propose an algorithm suitable for real-time editing that deals with strings of characters without requiring tombstones. This algorithm should offer support for aggregation of edits at the end or at the beginning of an existing string, allowing small edits to be sent as soon as they are typed in. This feature of real-time collaborative editing should be supported by a non costly solution that does not need additional identifiers, i.e. by creating only one large string once all edits have been sent and received. In what follows we describe our proposed solution.

III. PROPOSITION

A. Overview

The basic idea of our algorithm is to associate one sequence of data (i.e character) with only one identifier, and to generate identifiers that stand into a sequence. As in LOGOOT and TREEDOC algorithms, the set of identifiers must be ordered and dense. A common way to achieve this property is to use sorted lists of elements that can be lexicographically compared. For the sake of comprehension, in this section these elements are assimilated to the alphabet from 'a' to 'z', and the lists of elements – the identifiers – to words. In practice, any finite and large enough ordered set can fit, but different users shall not

[2]In mathematics, a partial order $<$ on a set X is said to be dense if, for all x and y in X for which $x < y$, there is a z in X such that $x < z < y$.

be able to produce the same identifiers. This can be achieved for instance by reserving for each user one specific element to be appended at a precise position in the list.

In our approach, an identifier of a character is of the form $base : offset$ where $base$ represents the first elements except the last one, $offset$ is the last element, and : is the concatenation operator. An identifier of a string is of the form $base[o_1.o_n]$ and it represents all characters identified by $base : o_1$,$base : o_2$,...,$base : o_n$. The character $(m+1)$ of the string is identified by $base : (o_1 + 1)$. The interval $[o_1.o_n]$ can be internally represented by the starting offset and the size of the interval.

For instance, the newly inserted string "HEY" can be represented in LOGOOT [12] as the following sequence of three characters : <daa,H><dab,E><dac,Y>. LOGOOT uses the same character representation as in our approach. But in our approach this string will be represented using the following single element: <da[a.c],HEY>. This further allows to insert new characters at the end of the string in order to obtain the new string "HEY**WO**" without creating an additional identifier, as illustrated with the sequence: <da[a.e],HEYWO>.

The newly generated identifiers are 'dad' and 'dae' and they are appended to the group 'da[a.c]' to create 'da[a.e]'. It is worth noting that the new identifier is unique. In order to ensure that two different users do not generate the same identifier, only the initial creator of the string can modify it. This is in practice not constraining, as it is very frequent that a single user is typing a whole sentence character by character. In this case the creator is the one that updates the string each time a character is added.

The proposed document model can be updated using the two following local functions:

- $insert(pos, str)$ inserts a string str at position pos in document.
- $remove(from, to)$ deletes the part of the text between the two positions $from$ and to (both characters located at $from$ and to are included in the deleted portion).

To insert, we basically need to create a new identifier between two other (uncompressed) identifiers, or in other words to generate a word that lies between two others. It is always possible to find such a word assuming a word never ends by a blank '␣'. It is obvious in some trivial cases such as finding a word between 'abc' and 'def' – at least anything that starts with 'b' or 'c' can fit. In some complex cases, the blank character '␣' is used. We can easily add 'aa' between 'a' and 'b'. However, an element inserted between 'a' and 'aa', would be identified by 'a␣a'. This allows us to have an ordered and dense set of identifiers as needed. Once the new identifier is created, it is sent together with the inserted string to the other users. Inserting into an existing compressed identifier will lead to three elements: the former compressed one is now split into two identifiers – similar to the former one but with the corresponding split interval –, plus a third identifier that lies in the middle of these two.

To delete a part of the document, we simply need to find the (uncompressed) identifiers that stand between the given positions, to store them for an immediate broadcast to the other users, and to delete these identifiers and the associated characters from the document. This can also lead to one split. In our example, if we delete character 'Y' we need to delete the identifier 'dac'. The data structure becomes: <da[a.b],HE><da[d.e],WO>

The two preceding functions $insert$ and $remove$ generate respectively the two following operations that will be broadcast to other users' copies in order to apply the changes on them:

- $add(id, str)$ adds the string str with identifier id to the data model.
- $del(id)$ deletes an interval of text whose identifier is id.

In order to add a new identifier to the data model, we need to find its position among the other (uncompressed) identifiers. This can be easily achieved using dichotomic search, and then the new identifier can be added. The same search procedure is applied in order to find the characters that must be removed in the case of a deletion.

These two operations are executed remotely when received at the other users' copies. On these copies, the initial real locations of the characters in the document might have changed – since the document may have been updated locally in parallel –. But, unique identifiers determine the right place to insert or remove a string.

An identifier of a character that is deleted cannot be re-used. In order to avoid reusing of the same identifier the data structure maintaining the $base$ part of an identifier keeps information about the minimum and maximum $offset$ generated.

Similar to other CRDTs such as LOGOOT and TREEDOC our approach does not propose any solution for ensuring causality. However, causality is not needed and our approach works well without it. If for some reason causality becomes mandatory, our algorithms can be used together with any mechanism that ensures causal delivery of operations such as causal barriers [16].

B. Algorithms

Before describing our algorithms for executing local and remote insertions and deletions, we present some functions used by these algorithms.

The function GenerateBase(idLow, idHigh) is used to generate a new base between the two existing identifiers idLow and idHigh.

```
function GenerateBase(idLow, idHigh)
  low=infiniteIterator(idLow, MIN_VALUE)
  high=infiniteIterator(idHigh, MAX_VALUE)
  newID=[]
  l=low.next()
  h=high.next()
  while (h−l < 2) do
    newID.append(l)
    l=low.next()
    h=high.next()
  end while
  // generate a random number between l and h values and append it
  newID.append(RandomNumber(l, h))
  newID.append(SEPARATOR)
  newID.append(replicaNumber)
  newID.append(SEPARATOR)
```

```
    newID.append(localClock++)
    return newID
  end function
```

An element of an identifier can take any value between MIN_VALUE and MAX_VALUE.

We defined a special iterator that returns element by element the values of the lists and completes that list with an infinite number of MIN_VALUE for the lower identifier (or an infinite number of MAX_VALUE for the higher identifier) when all the values contained in the list have been enumerated. At each step, the two values returned are compared and the lower one is appended to a new list as long as the distance between the two values is smaller than two, i.e as long as we cannot insert a new value between the two values returned by the iterators. When the distance is greater or equal to two (this case will always occurs, since we will eventually compare MIN_VALUE and MAX_VALUE), a new value between the two values is randomly created and appended to the new list. The site number is then added, plus the value of the local clock. This part of the identifier forms the *base*. A string identifier is created by adding an interval to this *base*.

SEPARATOR is a special value used to separate the generated id and the replica number or clock, to ensure the uniqueness of identifier if the last element needs various sites to be represented. For example, the separator avoids the following ambiguous problem: the concatenation of site 12 with clock 1 and site 1 with clock 21 give both 121 without any separator.

The function NumberOfInsertableCharacters computes the number of characters that can be inserted between an identifier and the next identifier in the model. This helps to decide if we need to split a block or not: when we want to insert one large string, the interval of the associated identifier is large, too, i.e lots of uncompressed identifiers compose it. We need to check if these uncompressed identifiers can be compressed into another, or if the next element stands between two of them. In this last case we need to create several compressed identifiers. This function takes as arguments idInsert the identifier of string to be inserted, idNext the identifier of next element in the data structure and length the size of string associated to the identifier that will be inserted. The returned value is the length if there is enough room, otherwise it is the maximum number of characters that can be inserted. There are two cases to consider. The first one is when the *base* part of idInsert is not a prefix of idNext. In this case length can safely be returned since it means that any identifier which has this base as a prefix – and it is the case for every uncompressed identifier represented in idInsert – is smaller than idNext. On the other case, when the base of idInsert is a prefix of idNext, it is necessary to check if the end of the interval is smaller than the value in idNext that stands after the base. If it is smaller it can return length.

```
function NumberOfInsertableCharacters(idInsert, idNext, length)
  if idInsert.length < idNext.length then
    for i from 0 to idInsert.length−1 do
      if idInsert[i] ≠ idNext[i] then
        return length
      end if
    end for
    return idNext[i]−idInsert[i]+1
  else
    return length
  end if
end function
```

In what follows we present the local and remote insertion and deletion procedures.

When a user types in some text in her document the local insertion procedure $insert(pos, str)$ is called which performs the following actions:

1) Search for the identifiers of the characters c_1 and c_2 at positions pos and $pos{+}1$ respectively, i.e. $id_{pos} = base_1 : o_1$ and $id_{pos+1} = base_2 : o_2$.
2) Check if c_1 was generated at this site and the sum of offset o_1 plus size of str is less than MAX_VALUE and c_1 is at the end of a block. In this case, str is appended to the character c_1 and the operation $add(base_1 : (o_1 + 1), str)$ is broadcast to the other user copies.
3) Similarly to the previous step check if c_2 was generated at this site, if it is located at the beginning of a block and if str can be inserted before it inside the block. The corresponding remote add operation is broadcast to the other user copies.
4) Otherwise, check if c_1 is inside a block, split the block in two blocks in this case. In both cases, generate a new identifier between id_{pos} and id_{pos+1} by calling the function GenerateBase. And, generate the corresponding add operation with this new identifier.
5) Return the generated operation.

The local insertion procedure always generates an add operation, that is broadcast to other users' copies in order to apply the changes on these copies.

The remote operation $add(id, str)$, where id is of the form $id = base : offset$, performs the following actions:

1) Search the element with the largest identifier smaller than id ($id_1 = base_1 : o_1$) and the element with the smallest identifier larger than id ($id_2 = base_2 : o_2$). If $base = base_1$ and $o_1 + 1 = offset$, append str to the block with the base $base$. Otherwise if $base = base_2$ and $offset + 1 = o_2$, add str at the beginning of the block with the base $base$. Otherwise create a new element.
2) Check if the string needs to be split with a call to n =NumberOfInsertableCharacters(id,id_2,$str.length$).
3) Eliminate the first n characters from str and assign the remaining characters to $newString$. If $newString$ is not empty, compute $newId = base : (offset + n)$ and call recursively $add(newId, newString)$.

When a user removes some text in her document, the local deletion procedure $remove(from, to)$ is called which performs the following actions:

1) Search the identifiers between position $from$ and to.
2) Generate the delete operations with found identifiers.
3) Delete elements between $from$ and to. If position $from$ or to is in the middle of a block, split this block.
4) Return the corresponding del operation.

The local deletion procedure always generates a del operation that is broadcast to other users' copies in order to apply the deletion of these copies.

The remote operation $del(ids)$ searches elements identified by ids and deletes them.

C. Data Structure

In this section, we present three data structures that can be used to implement our data model: a naive version, a string-based version and a tree-based version. Each of these data structures has its own benefits and drawbacks.

Each data structure contains a base-block composed of: the base of the identifier and the minimum and maximum offsets for this base. In Fig. 2 and Fig. 3 these extrema are indicated by means of letters placed at the right-hand side of the base-block.

*1) Naive Implementation (*LOGOOTSPLITNAIVE*):* In the naive data structure, the algorithm stores the blocks sorted by identifiers in a list such as presented in section III. Each block contains a string and its identifier. When a user deletes or inserts an element inside a block, the algorithm must split the block. It returns two new blocks with different intervals – but with the same *base* part of the identifier – and the related portion of the initial string. Concretely, the data model of this version is an array containing blocks. A dichotomic search can be used to find an element based on its identifier.

The drawbacks of this implementation is the cost of the lookup to find the ith element for local operations. Indeed, it is necessary to count (and sum up) the size of each block from the beginning of the list to find the right position. This is clearly a costly procedure, especially if the position that the algorithm is looking for is at the end of the document.

*2) String-based Implementation (*LOGOOTSPLITSTRING*) :* In the string-based data structure, two arrays are used: one to represent the final string and a second one to store the offset of the character and a link to the *base* part of the identifier.

Fig. 2 illustrates the state of the data model with this string-based implementation after few insertions – the same example was previously presented in section III –. First the string "LLO␣" is inserted between "E" and "W" whose identifiers are respectively 'dab' and 'dad'. The string "ARLD" is inserted at the end of the content and then the first character "A" is deleted.

This structure consumes more memory, but it offers a direct access to block identifiers. It is easy to manage insertions and deletions. Unfortunately, the complexity of an insertion is equal to the size of the array. But as observed during the experimentations, this cost remains low in practice.

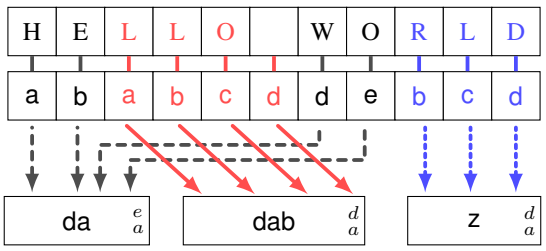


Figure 2: String-based implementation data structures

*3) Tree-based Implementation (*LOGOOTSPLITAVL*):* This data structure is derived from Ropes [17], a data structure for storing and manipulating strings. The key idea is to make a self-balanced tree with subsequences of the document stored in its nodes. The main difference with the original Ropes approach is that in our context no empty nodes exist. The tree is a self-balancing binary search tree implemented using an AVL [18]. This data structure behaves similarly to the naive implementation but each block is now organized in an AVL tree rather than an array.

Each node of the data structure contains:

- a subsequence of the document. It has the same content as the block in naive implementation.
- the offset of the subsequence: The offset of first element of the subsequence.
- a link to the base-block containing the *base* part of the identifier.
- two children nodes, before and after.
- the size of the string contained by this subtree. The size of the left child represents the position of the subsequence in the text.
- the height of the subtree, used to balance the tree.

An element can be searched either based on its absolute position or on its identifier. When the absolute position is provided the tree is traversed based on the sum field of the left child node. When the identifier is provided the tree is traversed by using the *base* part and the offset field of the node. For both searches three possibilities exist: the element is located before, inside or after a node. If the searched element is located before a node we compare it again with the left child of the node and if it is located after the node we compare it again with the right child. If the left or respectively right child of the node does not exist we create a new node and connect it in the case of an insertion, or return an error in the case of a deletion. If the searched element is located inside the node, we split the node in two nodes. The second node becomes the right child of the first one and the searched element is created as the left child of the second node if the operation is an insertion. If the operation is a deletion, the searched element is deleted. A node with empty string is deleted.

On each search for an insertion or a deletion, the target path is kept in order to apply the tree rebalancing procedure.

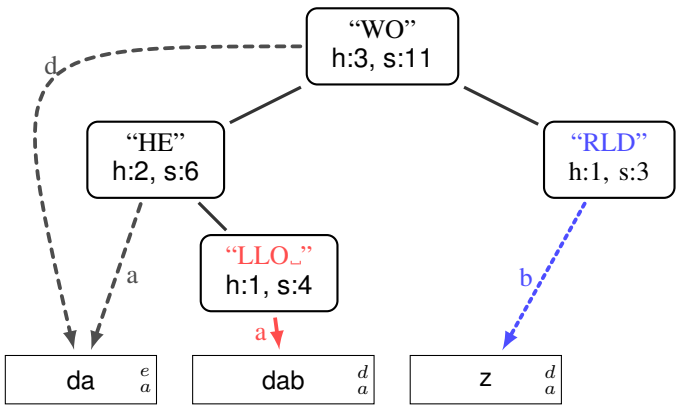


Figure 3: Tree-based implementation data structures

This procedure traverses the tree from the end of the path and performs the following actions until no more nodes exist in the path:

- while $|leftChild.height - rightChild.height| \geq 2$ the tree will be balanced as described in [18]. On each rotation the new father is added to the path for consistency maintenance. This operation needs to be repeated several times since a split during an insertion could create more than one node at a time – contrary to basic AVL.
- update the length of the string contained in the subtree.
- update the height of the subtree with the max of children height plus one.
- pop node from path.

Fig. 3 illustrates the tree-based data structure for the example presented in section III at a site that received first the operation of addition of the string "ARLD" at the end of the string "HEWO", then the deletion of character "A" and then the insertion of the string "LLO␣" between "E" and "W".

IV. Correctness

Our proposal ensures two correctness criteria: *Convergence* and *Intention preservation*.

Definition 1. *Convergence property: When no new updates are generated and all messages (operations) have been delivered to all users' sites, user document copies have the same state.*

Identifiers are unique, the $base$ component of an identifier being composed of a unique site identifier and the number of operations generated at that site. Moreover, identifiers are totally ordered. Therefore, when all operations are received by all sites, the document copies are identical being composed by the same sequence of elements ordered by their corresponding identifiers.

Definition 2. *Intention preservation property:*

1) *Each character inserted between two other characters in the document viewed by a user, needs to keep its relative position between its neighbors during the editing process.*
2) *Two concurrent string insertions at the same position lead to one string followed by another – same order on document copies –, not to a random interleaving of both strings.*

An identifier is uniquely constructed between two neighbor identifiers. Identifiers are sorted in a total order and are never modified during the editing process. Therefore the first point of intention preservation property is ensured.

The second point is ensured since the GenerateBase algorithm first creates a base, then an interval. So two concurrent insertions at the same place have different bases, and so if the base of the first string is smaller than the one of the second, any character of the first string has a smaller identifier than any character of the second, hence the characters are not mixed up.

V. Evaluation

A. Average-case Time Complexity Analysis

We denote by n the size of document, l the number of blocks, d the number of deleted elements and s the string added or deleted. We denote by f the number of times a block was split. We denote by i the size of an identifier. An identifier grows when a block is split or when offset reaches MAX_VALUE. In the worst case the number of blocks l is equal to the size of document n. This case happens when users created a document by inserting the text content character by character. We denote a variable x by x^* if it was implemented in a constant amortized time. The generation of a new identifier id has a cost of $\mathcal{O}(i)$. Copying a string in a new block costs s^*. During insertion or deletion, the cost of finding a base-block in the hash table is l^*.

In the naive implementation, the data model is an ordered array of blocks. The search function for the position of an element needs to sum up the lengths of all blocks preceding that position. The cost of this function is $\mathcal{O}(l)$, l being the number of blocks. The search function for the identifier of an element in the array has a cost of $\mathcal{O}(i \times \log(l))$ where i is the cost to compare two identifiers and $\log(l)$ is the complexity of a dichotomic search. The complexity of local insertion is $\mathcal{O}(l + s^* + i)$ since l is the cost of identifier search, l^* is the cost of the base-block search (l subsumes l^*), i is the cost of the new identifier generation and s^* is the cost of copying the string content. For remote insertion, the complexity is $\mathcal{O}(i \times \log(l) + l^* + s^*)$ where $i \times \log(l)$ represents the cost of identifier search function, l^* is the cost of the block insertion in the array and s^* is the cost of copying the string. However, in non-causal delivery insertion in a block which was split can be delivered before the operation that caused the split. We

TABLE I: SUMMARY OF ALGORITHMS' TIME COMPLEXITY

Algorithm name	Search		Insertion		Deletion	
	of a Position	of an Identifier	Local	Remote	Local	Remote
LogootSplitNaive	l	$i \times \log(l)$	$l + i$	$i \times \log(l) \times f$	$l + f$	$i \times \log(l) \times f$
LogootSplitString	1	$i \times \log(n)$	$s + i$	$i \times \log(n) \times f + s$	f	$i \times \log(n) \times f$
LogootSplitAVL	$\log(l)$	$i \times \log(l)$	$\log(l) + i$	$i \times \log(l) \times f$	$f + \log(l)$	$i \times \log(l) \times f$
Logoot	1	$i \times \log(n)$	$s \times i$	$i \times \log(n) \times s$	s	$i \times \log(n) \times s$
Treedoc	i	i	$s \times i$	$s \times i$	$i \times s$	$i \times s$
Woot	$n + d$	1	$n + d + s$	s	$n + d + s$	s

l: # of blocks, n: size of the document, i: the size of an identifier, d: # of deleted elements, s: size of the string to inserted/deleted, f: # of times a block was split

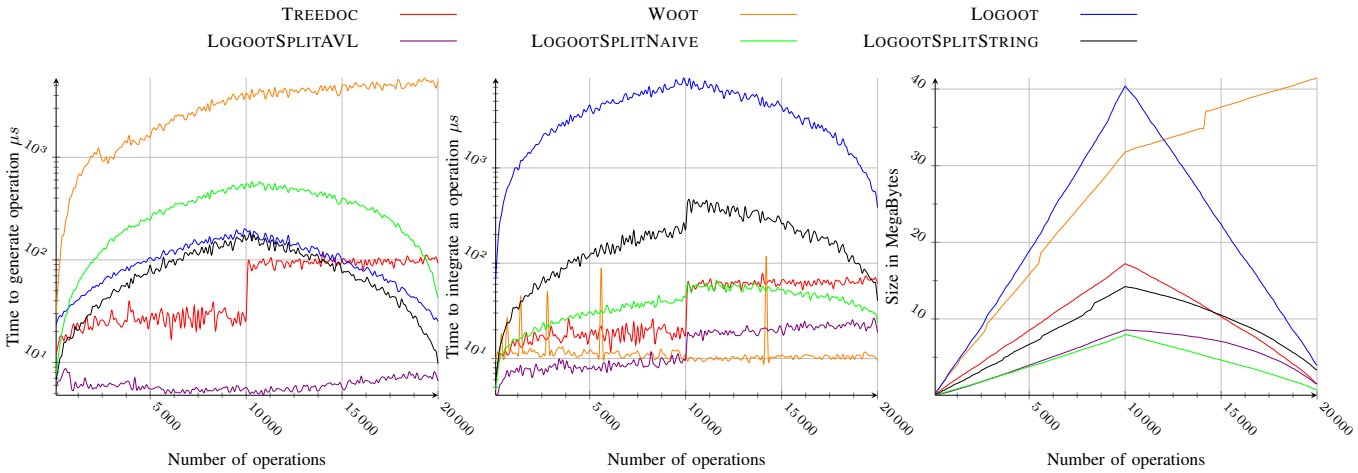
use f to count the fragments of the block, i.e the number of blocks in which it was split. This complexity becomes: $\mathcal{O}((i \times \log(l) + l^* + s^*) \times f)$. Locally a deletion can only delete contiguous blocks. Therefore the cost is $\mathcal{O}(l + f \times i^*)$ where l is the cost of searching the position, l^* the cost of deleting blocks and $f \times i^*$ the cost of copying the identifiers to forge the operation. For remote deletion, the block can be non-contiguous and thus each identifier of a fragment is sent (one by character in the worst case). Therefore the complexity is $\mathcal{O}((i \times \log(l) + l^*) \times f)$. The first product is the cost of search identifier and l^* is the cost of the deletion of a block in an array, and this process is performed f times.

In the string implementation, the search function for a position has a constant complexity as the underlying data structure is an array and each character is linked to its block. The search function for an identifier has a cost of $\mathcal{O}(i \times \log(n))$, because we use dichotomy on each character identifier. The complexity for local insertion is $\mathcal{O}(n^* + s + i)$ since adding a new string and allocate its place costs n^*, generating the offset block and linking it to the new block costs s and generating the new identifier costs i. The complexity of remote insertion is $\mathcal{O}((i \times \log(n) + n^*) \times f + l^* + s)$. The first product is the cost of a searching the identifier and n^* is the cost of inserting using an array copy. Since the string to be inserted can be fragmented, this processing is performed f times. l^* is the cost of searching the base-block in the hash-table, and s is the cost of creating a new block for each character and linking it to the base-block. The local deletion is always made on contiguous blocks which could have been fragmented, thus every fragment identifier must be sent. Therefore the cost of local deletion is $\mathcal{O}(f \times i^* + n^*)$ where $f \times i^*$ is cost of copying the identifier and n^* is the cost of deleting of contiguous characters using an array copy. The remote deletion is the deletion of each fragment, it has a cost of $\mathcal{O}(i \times \log(n) \times f \times n^*)$ where the first product is the cost of searching a fragment identifier and n^* is the cost of deleting in an array using array copy. These two operations are performed for each fragment (f times). In the worst case the algorithm could search every character with one block by character.

In the tree implementation, searching a block from its position has a complexity of $\mathcal{O}(i \times \log(l))$ since each node stores the size of the sub-document at its left and since the tree is self-balanced. The balancing procedure traverses a path in $\log(l)$ and rotates the tree in a constant time as described in AVL tree [18]. By definition, the balancing count is equals to the number of added and deleted nodes. Two nodes can be added by a single operation (split and new node) during an insertion. During a deletion it could be more – until l – if we delete the entire document. The cost of self-balancing is hidden by the search complexity in add operation and this is the contrary in delete operation. The update itself is efficient, insertion is a copy of added string or append an element (s^*). The complexity of local add is $\mathcal{O}(\log(l) + s^* + i)$. $\log(l)$ is the cost of search, s^* is the cost of string copy and i is the cost of generating a new identifier. For same fragmentation reason the complexity of remote add is $\mathcal{O}((i \times \log(l) + s^*) \times f + l^*)$. $i \times \log(l)$ is the cost of identifier search, s^* is the cost of copying the string and l^* is the cost of searching the base-block in the hash-table. The complexity of local deletion is $\mathcal{O}(f \times i^* + \log(l))$. i^* is the cost of copying the identifier, $\log(l)$ is the cost of balancing the tree which is more expensive than search since it executed more than one time (f times). The remote deletion has a complexity of $\mathcal{O}(i \times \log(l) \times f)$ where $i \times \log(l)$ is the cost of searching identifier and balancing and it is performed f times.

TREEDOC is not based on a self-balanced tree and therefore identifiers grow on each insertion when insertions are always performed at the right of the last insertion. As in our proposition identifiers are never rewritten. Consequently, its search function has a complexity of $\mathcal{O}(h)$ in the worst case. TREEDOC is not optimized for strings. The insertion costs $s \times i$ because one identifier is generated for each character. An improvement is to generate the first identifier and make a balanced tree. The complexity becomes $s + i$.

Table I presents a comparison of the time complexity of the different algorithms. For the sake of simplicity, provided complexities do not include amortized costs.



(a) Generation time (b) Remote integration time (c) Memory occupation

Figure 4: Experimentation results for random trace

B. Experimentation

1) Description: To evaluate performance of our proposed approach, we implemented our algorithms in Java and integrated them into the JBenchmarker project[3]. The experimentations were executed on Oracle JVM 7 powered by Ubuntu 11.04 on Xeon 5160 processor (4MB cache, 3GHz, 1333 FSB) workstation with 8GB of RAM. Our implementation is not multi-threaded, therefore each algorithm uses only one core. We measured the time of operation generation (when a user modifies the document), the integration time (when another user receives a modification) and the memory occupation. The time is measured by System.nanotime() and the memory occupation is measured by means of a dedicated Java library[4]. We launched each algorithm with the same sequence of local operations on a text document. Such sequences are called traces. No traces of real-time massive collaborative editing with a large number of users that edit concurrently shared data are available. For instance, Wikipedia traces are already serialized and do not capture concurrency between user edits. We therefore randomly generated different types of traces for evaluating the performance of our algorithms.

Two main types of traces were generated. The traces contain two types of local operations $insert(pos, string)$ and $remove(from, to)$. In the first trace insert/remove operations were generated at random positions with a random string of 50 characters. In the second trace insert operations of random strings of 50 characters were generated contiguously at the end of the document and remove operations of 50 characters were generated randomly. For each experiment, 100 sites are virtually generating operations.

Each trace is composed of 20 000 operations and split in two parts: the 10 000 first operations are made of 80% insertions and 20% deletions and these ratios are inverted in the second part of the trace. So basically, the first half of each trace builds the document with lots of insertions, while the second half destroys the document with lots of deletions.

In our experiments we compared the performances of our approach with the ones of main existing CRDT approaches. The comparison between the performances of CRDT and operational transformation approaches can be found in [14].

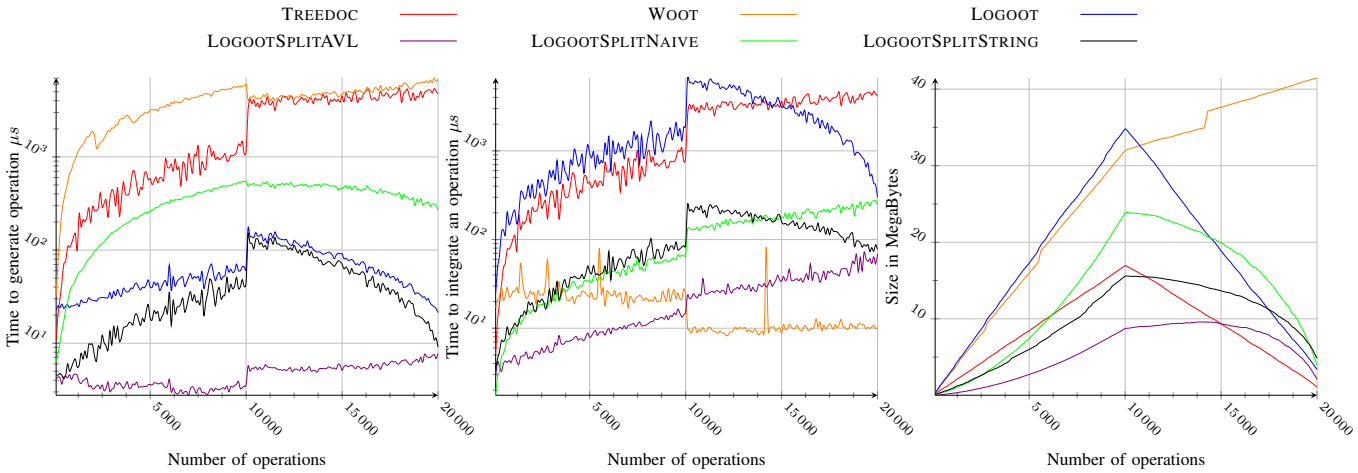
2) Results for traces with random insertion: We first analyse the generation time (Fig. 4a) and we can see the benefit of balanced trees. Indeed TREEDOC[5] and LOGOOTSPLITAVL feature the best performances. However, the benefits of using blocks are not noticeable as LOGOOT access is constant (implementation based on an array) and it is optimized to generate at once identifiers for all characters in the string to be inserted, and their insertion in the data structure is made at once. However, it generates many operations and features a high integration time as described later on in this section. As expected, the cost to find an element in naive data structure is high. We can see that the tombstone strategy is costly as elements are never deleted from the document model: during the destruction part, the size of the model does not change, and so does not the time to convert a position into an identifier.

Fig. 4b illustrates the remote integration time of the studied algorithms. The graphic shows the cost of integration of one character operation. We can notice the advantage of string management with LOGOOTSPLITSTRING. LOGOOTSPLITSTRING takes more time to integrate an operation than to generate it : integration needs a dichotomic search while generation requires a direct access. The naive implementation uses dichotomic search on blocks and not on characters as in LOGOOTSPLITSTRING or LOGOOT. TREEDOC model in this case is a balanced tree as the trace has random insertion positions and hence characters are equally distributed in the tree. Hence the tree traversal to find the right place to insert a character is quite fast. LOGOOTSPLITAVL uses a self-balanced tree on blocks, contrary to TREEDOC which uses characters. WOOT is fast because it uses a hash table to find an element or its neighbors. However we can notice some spikes in the WOOT representation as the integration mechanism is quite complex in the case of

[3]https://github.com/score-team/replication-benchmarker

[4]https://github.com/dweiss/java-sizeof – Memory consumption estimator

[5]In this experiment, the internal tree structure of TREEDOC is balanced since insertions are performed at random positions.



(a) Generation time (b) Remote integration time (c) Memory occupation

Figure 5: Experimentation results for trace with right-hand insertions

concurrent insertions at the same position.

Concerning memory occupation (Fig. 4c), the least performant algorithms are LOGOOT as it uses one identifier per character, and WOOT as it uses tombstones that grow constantly. In TREEDOC adding a character is equivalent to adding a node to the tree model which is less costly than generating large LOGOOT identifiers. For all LOGOOTSPLIT algorithms, the deletions in the second part of the trace fragment the blocks. In this case the ratio $\frac{character}{block}$ decreases and the required memory increases.

3) Results for traces with right-hand insertions: This experiment shows the drawbacks of unbalanced tree in TREEDOC. If every character is inserted at the right-hand of the document, the tree behaves as a unoptimized list, and the searching algorithm becomes linear. It also shows that LOGOOT identifiers grow continuously.

VI. CONCLUSIONS

In this paper we presented a novel commutative replicated data type for sequences of text. Our proposed CRDT has the particularity of assigning unique identifiers to substrings of variable length contrary to existing CRDTs that assign unique identifiers to fixed size elements of the text (i.e. characters or lines). This offers the possibility to define coarse-grained elements when they are created and refine them when needed. This greatly reduces the memory consumption since a smaller memory overhead is needed to store metadata (identifiers). Moreover, we show that overall performances of our algorithms are above the others in average using simulations. On frequent collaborative editing scenarios, the performance improvement is even better.

From a user point of view, our algorithm ensures that contiguous edits of the same user are not interleaved with any other concurrent edits. Other algorithms in the literature do not have this property, and the ordering of concurrent insertions is decided by means yet deterministic but still uncertain. From a user point of view, this allows users to contiguously type sentences without being disturbed by overlapping changes of any other users, and reduces the number of interventions needed to manually re-order the characters afterwards. Furthermore, large blocks can also be handled in one operation (during a copy-paste operation for instance). Thus, the integration of such edit is nearly as fast as a single character operation, and the user is not blocked a long time until every character of the string is inserted.

While we focused on collaborative text editing in this paper, the proposed CRDT deals with a linear sequence of elements. Indeed, this CRDT is above all an algorithm for optimistic replication on any dataset which can be linearizable and therefore could be applied in many other application contexts.

ACKNOWLEDGMENTS

This work is partially funded by the french national research program STREAMS (ANR-10-SEGI-010).

REFERENCES

[1] Y. Saito and M. Shapiro, "Optimistic Replication," *ACM Computing Surveys*, vol. 37, no. 1, pp. 42–81, 2005.

[2] C. A. Ellis and S. J. Gibbs, "Concurrency Control in Groupware Systems," *SIGMOD Record : Proceedings of the ACM SIGMOD Conference on the Management of Data - SIGMOD '89*, vol. 18, no. 2, pp. 399–407, 1989.

[3] D. A. Nichols, P. Curtis, M. Dixon, and J. Lamping, "High-latency, Low-bandwidth Windowing in the Jupiter Collaboration System," in *Proceedings of the 8th Annual ACM Symposium on User interface and Software Technology - UIST '95*, Pittsburgh, PA, USA, 1995, pp. 111–120.

[4] C. Sun and C. Ellis, "Operational Transformation in Real-Time Group Editors: Issues, Algorithms and Achievements," in *Proceedings of the ACM Conference on Computer-Supported Cooperative Work - CSCW '98*, Seattle, WA, USA, 1998, pp. 59–68.

[5] M. Suleiman, M. Cart, and J. Ferrié, "Concurrent Operations in a Distributed and Mobile Collaborative Environment," in *Proceedings of the International Conference on Data Engineering - ICDE'98*, Orlando, FL, USA, 1998, pp. 36–45.

[6] L. Lamport, "Times, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[7] A. Imine, P. Molli, G. Oster, and M. Rusinowitch, "Proving Correctness of Transformation Functions in Real-Time Groupware," in *Proceedings of the European Conference on Computer-Supported Cooperative Work - ECSCW 2003*, Helsinki, Finland, 2003, pp. 277–293.

[8] G. Oster, P. Molli, P. Urso, and A. Imine, "Tombstone Transformation Functions for Ensuring Consistency in Collaborative Editing Systems," in *Proceedings of the International Conference on Collaborative Computing: Networking, Applications and Worksharing - CollaborateCom 2006*. Atlanta, GA, USA: IEEE Computer Society, 2006, pp. 1–10.

[9] F. Mattern, "Virtual Time and Global States of Distributed Systems," in *Proceedings of the International Workshop on Parallel and Distributed Algorithms*. Château de Bonas, France: Elsevier B.V., 1989, pp. 215–226.

[10] G. Oster, P. Urso, P. Molli, and A. Imine, "Data Consistency for P2P Collaborative Editing," in *Proceedings of the ACM Conference on Computer-Supported Cooperative Work - CSCW 2006*, Banff, AB, Canada, 2006, pp. 259–267.

[11] N. Preguiça, J. M. Marquès, M. Shapiro, and M. Letia, "A Commutative Replicated Data Type for Cooperative Editing," in *Proceedings of the 29th International Conference on Distributed Computing Systems - ICDCS 2009*, Montreal, QC, Canada, 2009, pp. 395–403.

[12] S. Weiss, P. Urso, and P. Molli, "Logoot : A Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks," in *Proceedings of the 29th International Conference on Distributed Computing Systems - ICDCS 2009*, Montreal, QC, Canada, 2009, pp. 404–412.

[13] ——, "Wooki: a P2P Wiki-based Collaborative Writing Tool," in *Proceedings of the International Conference on Web Information Systems Engineering - WISE 2007*, Nancy, France, 2007, pp. 503–512.

[14] M. Ahmed-Nacer, C.-L. Ignat, G. Oster, H.-G. Roh, and P. Urso, "Evaluating CRDTs for Real-time Document Editing," in *Proceedings of the 11th ACM Symposium on Document engineering - DocEng 2011*, Mountain View, CA, USA, 2011, pp. 103–112.

[15] W. Yu, "A String-wise CRDT for Group Editing," in *Proceedings of the 17th ACM International Conference on Supporting Group Work - GROUP 2012*, Sanibel Island, FL, USA, 2012, pp. 141–144.

[16] R. Prakash, M. Raynal, and M. Singhal, "An adaptive causal ordering algorithm suited to mobile computing environments," *Journal of Parallel and Distributed Computing*, vol. 41, pp. 190–204, 1997.

[17] H.-J. Boehm, R. Atkinson, and M. Plass, "Ropes: An Alternative to Strings," *Software: Practice and Experience*, vol. 25, no. 12, pp. 1315–1330, 1995.

[18] G. M. Adelson-Velskii and E. M. Landis, "An Algorithm for the Organization of Information," *Doklady Akademii Nauk SSSR*, vol. 146, pp. 263–266, 1962, (English translation in Soviet Mathematics Doklady, vol 3, pp. 1259–1263).