High Responsiveness for Group Editing CRDTs

Loïck Briot
Université de Lorraine
Loria & Inria
Nancy, France
loick.briot@mines-nancy.org

Pascal Urso
Université de Lorraine
Loria & Inria
Nancy, France
pascal.urso@loria.fr

Marc Shapiro
Sorbonne Universités
UPMC-LIP6 & Inria
Paris, France
marc.shapiro@acm.org

ABSTRACT

Group editing is a crucial feature for many end-user applications. It requires high responsiveness, which can be provided only by optimistic replication algorithms, which come in two classes: classical Operational Transformation (OT), or more recent Conflict-Free Replicated Data Types (CRDTs).

Typically, CRDTs perform better on *downstream* operations, i.e., when merging concurrent operations than OT, because the former have logarithmic complexity and the latter quadratic. However, CRDTs are often less responsive, because their *upstream* complexity is linear. To improve this, this paper proposes to interpose an auxiliary data structure, called the *identifier data structure* in front of the base CRDT. The identifier structure ensures logarithmic complexity and does not require replication or synchronization. Combined with a block-wise storage approach, this approach improves upstream execution time by several orders of magnitude, with negligeable impact on memory occupation, network bandwidth, and downstream execution performance.

1. INTRODUCTION

Mass remote collaboration tools are increasingly used. Massive group editing appears in many end-user applications such as Wikipedia, Google Docs, Office, Evernote, Dropbox, or VCSs such as git or svn. Group editing aims to allow a large number of users to edit the same document at the same time and from different places. Centralized platforms such as Google Docs offer world wide group editing service to millions of users but limit the number of participants to a document [3]. An alternative to centralized platforms is to take advantage of availability and scalability of decentralized peer-to-peer networks.

In group editing, high responsiveness is crucial [22]. Every user device hosts a replica of the document that she/he modify at will. The user observes the effect of her/his modification immediately, without any locking mechanism – or even network latency – perturbing his interaction. This behavior is called *optimistic replication* [19]. Also, the modification must be propagated to other users' devices and applied there with the same effect. Group editing systems distinguish *upstream* and *downstream operation execution.* A user's modification is applied immediately to some first, so-called *upstream* replica, and then sent to other replicas *(downstream).* Delivering a downstream operation merges its effects with concurrent modifications. Merging arbitrary modifications is a difficult and error-prone task, and thus many mechanisms have been proposed.

Submitted for publication. Do not disseminate without the authors' permission. 2016-02-19

Operational Transformation (OT) is an established mechanism for group editing applications [14, 17, 23] because this class of algorithms promises very high responsiveness for upstream operations execution. However, in a decentralized context, downstream execution of OT algorithms is complex and expensive because each replica must reorder its history and transform concurrent operations, implying a $O(N^2k)$ complexity with N the number of operations and k the number of replica.

To avoid this issue, Conflict-Free Replicated Data Types (CRDTs) have been proposed [13, 15, 25]. This class of algorithms uses operations whose downstream execution is commutative; this enables to keep the consistency of the collaborative document easily, without using transformations. Ahmed-Nacer et al. [6] show in particular that downstream execution of CRDT algorithms is more efficient by a factor between 25 to 1000 compared to representative OT algorithms.

However, upstream execution – and thus responsiveness – of CRDT algorithms often performs poorly. These algorithms associate a unique identifier with each element in the document. Unique identifiers allow a quicker and easier downstream execution, but must be generated and retrieved during upstream execution. A first improvement is to coarsen the granularity to identify large blocks rather than single characters [26]. Indeed, collaborative documents are often written block by block – copy/paste or batch operations for instance. The fewer elements in the document, the less computation is needed to apply and merge modifications. Moreover, each element contains metadata (including unique identifiers), so the fewer elements there are, the less memory overhead. Nevertheless, an algorithm with, say, linear complexity, remains linear using blocks. Furthermore, blocks slow down upstream execution, since elements are of different sizes, and retrieving element identifiers affected by user modifications requires to traverse the document. Finally, block-based algorithms remain relatively inefficient in execution complexity.

In this paper we propose a solution, easily adaptable to most algorithms, and that improves responsiveness considerably. The idea is to have an efficient auxiliary data structure – called the *identifier structure* – dedicated to CRDT identifiers. This data structure has no consistency requirements between replicas; and each peer can modify its own structure independently and efficiently; thus, there are no issues due to concurrency. More specifically, we implement this insight for element-based and block-based CRDT algorithms. Our solution is easy to implement; it decreases upstream complexity, without significantly degrading downstream complexity. Its memory and bandwidth usage is similar to the original algorithm.

The remainder of this paper proceeds as follows. Section 2 introduces the state of the art of the current knowledge in group editing OT and CRDT algorithms, as well as some definitions useful in the following sections. Section 3 presents our identifier structure for improving upstream complexity of CRDT algorithms and proposes its application on two different CRDT. Section 4 demonstrates how to combine our approach with block management. Section 5 evaluates our implementations with several and various experiments. Section 6 concludes.

2. BACKGROUND, STATE OF THE ART AND RELATED WORK

2.1 Definitions

To clarify and unify the presentation of the existing approaches we introduce the following definitions [9].

- **replica**: Several peers can edit a document from different places. Each peer hosts its own *replica* of the collaborative document. The replica contains elements visible to the end user as well as metadata required to manage eventual consistency.
- **operation**: When a peer apply user modification, it modifies its replica and generates *operations*. These operations are sent by the peer to the other peers. These operations contain the necessary data and metadata to ensure consistency when executed on the other peers.
- **upstream**: The process of applying user modification and generating corresponding operation(s) is called *upstream update*.
- **downstream** The process of executing remote operation coming from another peer and modifying the receiving replica is called *downstream update*.

2.2 Operational Transformation

Operational Transformation (OT) [17] is an optimistic replication mechanism [19]. It was the first approach used in the real-time group editing field and has been studied in the literature [14, 22].

In the OT approach, downstream operations are transformed before their integration in order to handle concurrency. The transformations should preserve the "intentions" of the user [22]. To ensure consistency, a received downstream operation is transformed against every concurrent update operation in a specific order.

The two main issues of OT algorithms in a decentralized context are consistency and scalability. Consistency issues are due to difficulties to define correct transformation functions. For linear documents, the only known transformation functions that are correct in a peer-to-peer context are the Tombstones Transformation Functions (TTF) [14]. However, tombstones impact responsiveness negatively since the upstream procedure must count them for every modification. Scalability issues are due to history ordering and downstream transformations. History ordering has a $O(N^2k)$ complexity[1] and requires vector clocks [12] whose impact negatively memory and bandwidth usage and limit the scalability of the approach.

2.3 Conflict-free Replicated Data Types

An alternative approach, called Conflict-free Replicated Data Type (CRDT), has been proposed [11]. The insight behind CRDT is to design a set of commutative update operations. Thus, no transformation and no history ordering is necessary. Commutative operations can be applied in any order, and are guaranteed to converge. The literature proposes several CRDT algorithms, but in what follows we mention only the most representative ones in the context of group editing.

2.3.1 WOOT

WOOT [13] was the first CDRT algorithm proposed for group editing. It is based on the principle that in a text document, an element is inserted between two others. In the WOOT approach, each element is identified by a unique identifier and linked to the two elements between which its was inserted. WOOT identifiers are composed of the identifier of the upstream peer and the value of local clock of this peer time. A deterministic algorithm orders the elements inserted in the same region, which is quite complex and is more less efficient than more recent CRDTs. Moreover, since each element position is relative to its neighbors, elements cannot be completely removed from the document, but are turned into invisible "tombstones". As TTF, these tombstones must be counted and impact upstream execution complexity. Optimized versions of this algorithm have been proposed [24], but only improve downstream complexity [6].

2.3.2 TreeDoc

TreeDoc [15] is a CRDT based on a binary unbalanced tree of elements. An element's a unique identifier is its path in the tree. This approach supports dichotomic search, but is expensive when the tree is unbalanced, for instance when many inserts occurs at the end of the document. It is impossible to balance the tree without synchronizing all nodes, because the tree must be the same at all peers in order to generate consistent operations. TreeDoc uses tombstones but they can be purged in some conditions. For example once a deleted node does not have any visible children, it can be safely removed.

2.3.3 Logoot

Logoot [25] is a CRDT that uses a lexicographic order to totally order the document elements. A Logoot identifier is a list of triples of integers. The first integer of the triple is a priority, the second is the upstream peer ID and the third is the upstream logical clock. When an insertion happens, the new element's ID ensures that it will be placed at the wanted position. Element identifiers are stored into an array. Upstream insertion consists of directly accessing elements n and $n + 1$, generating an identifier between them, and

[1] N is the number of operations and k is the number of replicas.

shifting the table to place the new element. Shifting a whole document is costly.

Another issue of Logoot is that the size of identifiers grows and becomes costly in terms of memory and performance. Identifiers grow faster when there are several insertions in the same part of the document. Let's say a peer p' (with clock $'i'$) decides to insert an element between two successive identifiers $[\langle p, i, n\rangle]$ and $[\langle p, i, n+1\rangle]$. The new identifier will be $[\langle p, i, n\rangle\langle p', i', n'\rangle]$.

2.3.4 LogootSplit

LogootSplit [7] is a block version of Logoot. Blocks are of different sizes and created trough copy-paste or buffering as a whole single Logoot identifier, reducing memory usage. Block algorithms must be able to split blocks in order to remove a part or to insert an element within the block. An offset number is present in the LoogotSplit identifiers to identify block's parts in a unique manner.

However, when naively using blocks, the upstream algorithm becomes more complex since the it needs to count the block size to map the position of a user modification to the corresponding block identifier and offset. Three different versions of LogootSplit have been proposed, the most efficient being the one using an AVL tree to organize the nodes.

2.3.5 RGA

The Replicated Growable Array (RGA) algorithm [18] uses the fact that every element is included immediately after another one in a text document. RGA data structure is essentially a linked-list, in which a node has a content, an identifier, a flag to say if the node is a tombstone, and the identifier of the next node in the list. A RGA identifier is composed of the upstream peer ID (sid) and of the sum of the upstream vector clock value during insertion (sum). Identifier i_1 is said to "precede" i_2 ($i_1 \prec i_2$) if $sum_1 < sum_2$ or $sum_1 = sum_2 \wedge sid_1 < sid_2$.

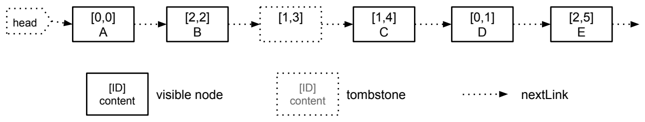

Figure 1: Example of a RGA list

An insertion operation contains the identifiers of both the new node and the node that it follows (called *reference node*). Insertion consists in placing the new node to the right of the reference node and at the left of the first node whose identifier **precedes** the new identifier. This simple and efficient procedure orders concurrent inserts. Moreover, to retrieve reference nodes, RGA uses a hash map, keyed by node identifier, mapped to the node itself.

In the common cases, the new node is inserted just after its reference node, but other computations are required if there are concurrent insert. Suppose three peers work on a collaborative document. We note o_x the operation that inserts the node N_x whose identifier is i_x. The initial state of the document is n_1n_5 with $i_1 = [0,2]$ and $i_5 = [2,1]$. The three peers concurrently insert a node just after N_1. Peer 0 inserts n_4, whose identifier is $i_4 = [0,3]$, Peer 1 inserts n_3 with $i_3 = [1,4]$, and Peer 2 n_2 with $i_2 = [2,5]$. Therefore, $i_5 \prec i_4 \prec i_3 \prec i_2$. The result below shows for each peer a different execution order, which nonetheless reach a convergent state.

$$Peer0 : n_1n_5 \rightarrow_{o_4} n_1n_4n_5 \rightarrow_{o_2} n_1n_2n_4n_5 \rightarrow_{o_3} n_1n_2n_3n_4n_5$$

$$Peer1 : n_1n_5 \rightarrow_{o_3} n_1n_3n_5 \rightarrow_{o_2} n_1n_2n_3n_5 \rightarrow_{o_4} n_1n_2n_3n_4n_5$$

$$Peer2 : n_1n_5 \rightarrow_{o_2} n_1n_2n_5 \rightarrow_{o_3} n_1n_2n_3n_5 \rightarrow_{o_4} n_1n_2n_3n_4n_5$$

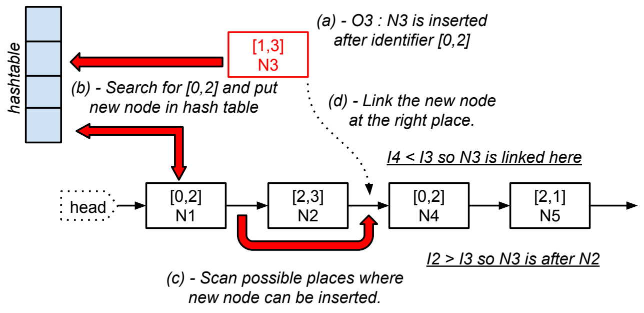
hashtable
[1,3]
N3
(a) - O3 : N3 is inserted after identifier [0,2]
(b) - Search for [0,2] and put new node in hash table
(d) - Link the new node at the right place.
I4 < I3 so N3 is linked here
head
[0,2]
N1
[2,3]
N2
[0,2]
N4
[2,1]
N5
I2 > I3 so N3 is after N2
(c) - Scan possible places where new node can be inserted.

Figure 2: Execution of operation o_3 at Peer 0

2.4 Summary

If we consider a high degree of responsiveness as a – at most – logarithmic complexity for upstream execution, then none of the existing algorithms present such performance. Indeed, each one has a flaw that affect upstream execution :

- OT needs to count TTF tombstones
- RGA and WOOT also needs to count tombstones.
- Logoot needs to shift its array and, as LogootSplit, needs to manage growing identifiers.
- Treedoc needs to traverse an unbalanced tree.
- Block approaches need to count block sizes.

We consider upstream efficiency as the most noticeable by the user [20]. However, downstream efficiency is also important, and RGA score best by this metric [6].

3. THE IDENTIFIER DATA STRUCTURE

The basic idea of our solution is to build an identifier data structure local to each peer. This is easily adaptable to the most CRDT algorithms, and improve upstream complexity. This structure must not augment remote complexity of algorithms, and must not use excessive memory compared to the base algorithm. Our approach really on the classic CRDT algorithm, and update the identifier structure on the side.

Figure 3 illustrate our solution. It shows the different steps in which our identifier structure is used. The step 1 is the user action. The steps 2 correspond to the steps where we use our structure to retrieve the node(s) corresponding to the user's action. The steps 3-5 are the base replication algorithm unmodified. The step 6-7 updates the identifier structure according to remote operations.

The critical steps are to retrieve efficiently identifiers from a user position (step 2) and conversely to update the identifier structure during downstream (step 6). To achieve this, we first observe that all the base CRDT algorithms manage a structure composed of nodes identified by a unique identifier. Our identifier structure will contain a set of organized nodes, called idNode. A idNode has a reference to an

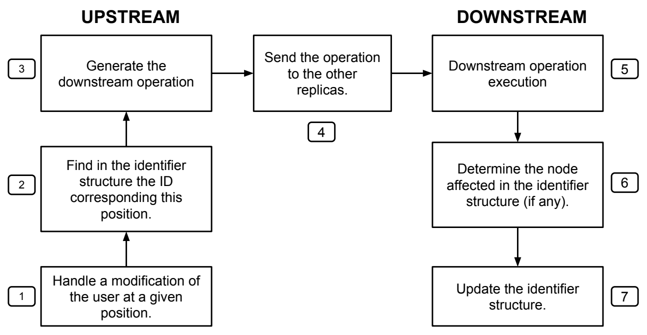
UPSTREAM
DOWNSTREAM
1 Handle a modification of the user at a given position.
2 Find in the identifier structure the ID corresponding this position.
3 Generate the downstream operation
4 Send the operation to the other replicas.
5 Downstream operation execution
6 Determine the node affected in the identifier structure (if any).
7 Update the identifier structure.

Figure 3: Steps to generate and execute an operation with our identifier structure

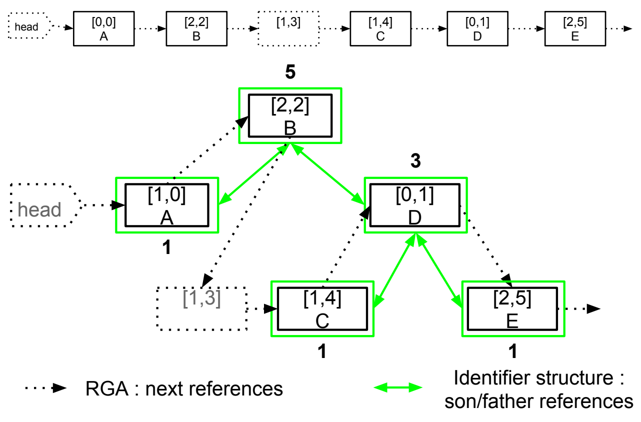

Figure 4: Example of a base and identifier structure for RGA

base CRDT node, as well as references to other idNodes to enable efficient traversal. Symmetrically, each node of the base data structure contains a reference to the corresponding idNode.

3.1 Retrieving CRDT identifiers

Retrieving a CRDT identifier is straightforward in algorithms such as Logoot that have no tombstones and single identifier per element. It is more complex for algorithms using tombstones and/or blocks. The naive solution used in RGA, WOOT, or LogootSplit consists in counting visible elements, starting from the head of the document, until the appropriate position is reached. Such procedures have a linear complexity, i.e. $O(N)$ with N the number of element ever inserted in the document, including tombstones.

The identifier data structure allows to map efficiently position of user modification to CRDT identifier. This data structure can be a skip list [16] when CRDT identifiers are totally ordered – e.g. Treedoc or Logoot identifiers – or a weighted binary tree for other CRDT.

A weighted tree contains a set of idNodes, one of them being the root of the tree. A idNode contains a *left child*, a *right child*, and a *parent* references to idNodes, a *weight* which is the total size of the sub-tree, and a reference to an base CRDT node. We add to the base CRDT node a reference to the corresponding idNode.

In the case of RGA, for instance, an base node contains a unique identifier, a content, a next reference, and a tombstone flag. We add to this RGA node, a reference to an idNode, whose reference is unset when the node becomes a tombstone.

Figure 4 illustrates the structure described above. The base RGA structure is drawn with dotted lines, whereas the new identifier structure is drawn with plain lines. Notice that identifier and base structures are decoupled – each structure has its own way to traverse the nodes – but both are linked. Only the non-tombstone nodes belongs to the identifier structure. Tombstones such as one identified by [1,3] are not present in the identifier structure, limiting memory overhead.

The weight of a node is the size of the subtree. The algorithm to find the node corresponding to a given position is the following:[2]

```
1: function FINDPOSINIDENTIFIERTREE(pos)
2:     if $pos == 0$ then
3:         return $head.identifier$
4:     else
5:         $node = root$                ▷ Begins from the tree's root
6:         while $pos \neq node.leftWeight() + 1$ do
7:             if $pos < node.leftWeight() + 1$ then
8:                 $node = node.leftChild()$        ▷ Goes left
9:             else
10:                $pos = pos - (node.leftWeight() + 1)$
11:                $node = node.rightChild()$      ▷ Goes right
12:        return $node.identifier$
```

The procedure traverses only the height of the tree. Let's suppose the user wants to insert element "X" after the third position in the RGA structure described in Figure 4. The FINDPOSINIDENTIFIERTREE procedure,

1. with $pos = 3 \geq 2$, goes to the right of $\{[2,2], B\}$
2. with $pos = 1 < 2$, goes to the left of $\{[0,1], D\}$
3. with $pos = 1$, returns $\{[1,4], C\}$.

Then, the new element "X" is inserted as the right-most child of the left sub-tree of $\{[1,4], C\}$ – here, directly to left of $\{[1,4], C\}$. The weights of the ancestors of the new node are updated as shown in red in Figure 5. The generated insertion operation (step 3) will contain the new node $\{[3,5], X\}$ and its *reference* [1,4].

Such an identifier structure can be easily adapted to similar tombstone-based algorithms such as WOOT, but also to block-based algorithm including the size of the nodes content in weights (see Section 3). It is also beneficial in a no-tombstone and no-block algorithm such as Logoot (see Section 3.3). With a balanced tree or a periodically balanced tree (see Section 5), the complexity of upstream execution is $O(log(n))$ with n the number of visible elements.

3.2 Updating identifier structure during downstream

As described above, we rely of the base CRDT algorithm to apply downstream operation. In RGA, for instance, an insertion consists of following the RGA list (presented with dotted lines in Figure 4), starting from the reference node until it finds a node with a lesser identifier (*target node*).[3]

[2]The method $leftWeight()$ returns the weight of the left child of a node (0 if their is no left child).

[3]RGA uses an hash table to start directly from the reference

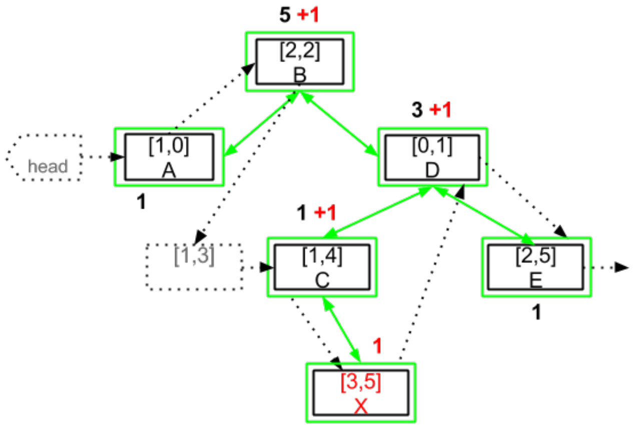

Figure 5: Upstream insertion

After the target node has been identified by the base algorithm we update the identifier structure. However, the target node might be a tombstone, and thus not belonging to the identifier structure. If the downstream operation is also a deletion we don't need to do anything. If the operation is an insertion, we continue to follow the base RGA linked-list chaining until the first non-tombstone node and we insert the new idNode at the left of this node.

```
1: function INSERTDOWNSTREAM($target, newNode$)
2:     while $target.nextLink \neq null \land \neg target.visible$ do
3:         $target = target.nextLink$
4:     if $\neg target.visible$ then          ▷ insertion at the end
5:         INSERTRIGHT($root, newNode$)
6:     else
7:         INSERTBEFORE($target, newNode$)
```

INSERTRIGHT inserts a node at the right-most position in the tree. INSERTBEFORE inserts a node at the right-most node of the left child of a substree. Both procedures update the corresponding weights.

Considering algorithms such as Logoot (or Treedoc), the problem is a bit different since there is no linked link. However, in these cases, since the identifier structure are totally ordered, the identifier structure is a binary search tree. So, we can simply traverse the tree to update it.

Last but not least, each peer can use an arbitrary identifier structure, including different shapes for it. This means that identifier structure can be different for each peer even if they base structure is the same. For instance, in Figure 6, we show two peers that have the same base RGA state but two different identifier data structures. This enable a peer to balance its identifier tree locally without synchronizing with the other peers.

3.3 Logoot with identifier structure

Our approach can be applied beneficially to the other single-element CRDT algorithms. For instance, we can apply it to Logoot [25]. The base Logoot algorithm uses an array to store identifiers and does not use tombstones. It retrieves identifiers for generating operations in constant time,

node.

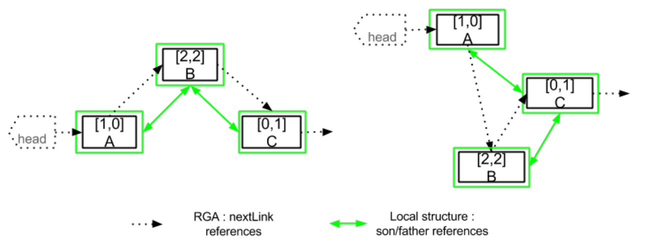
head
[1,0] A
[2,2] B
[0,1] C
head
[1,0] A
[0,1] C
[2,2] B
RGA : nextLink references
Local structure : son/father references

Figure 6: Two possible identifier tree states for the same RGA base state

but upstream complexity is affected by the shifting of the array to insert (or remove) an element. In essence, the complexity of the shift is $O(n)$.

We can replace this array by a skip list or the same weighted tree than above. The gain obtained by removing the shift exceeds the overhead due to the structure traversal which is only in $O(log(n))$.[4]

Considering downstream execution, the base Logoot uses a binary search to retrieve the targeted identifier. Since Logoot identifiers are totally ordered, the identifier tree structure is a binary search tree. Thus, the search of the target element for downstream execution will use the identifier structure with the same algorithmic cost. Furthermore, once the targeted element is found, updating the structure is done in constant time $O(1)$ in contrast to shifting the array in $O(n)$.

4. RGATREESPLIT: A BLOCK-WISE RGA WITH IDENTIFIER STRUCTURE

Since RGA is the most efficient algorithm for downstream execution, and since block-wise algorithm improve efficiency of both upstream and downstream execution [26], we propose a new algorithm called RGATreeSplit that combine these concepts.

To be able to handle blocks in a RGA algorithm, some changes are required on the base structure. These changes will be described in the next section. In the late section, we apply our identifier structure concept on this block-wise algorithm.

4.1 Block-wise RGA

A block-wise RGA node contains the information usual to RGA with some additional metadata. This algorithm uses the notions of split and offset introduced by Yu [26]. A block identifier is composed of a CRDT identifier plus an offset. When a node is first inserted, its offset is 0. Splitting a node whose offset is x at position pos produces to two nodes: a first one with offset x, and a second one with offset $x+pos$. In this way, each node is always uniquely identified. There is an additional reference to nodes, called splitLink. This reference enables quick access to all the nodes of the resulting from the split of a node by linking the two resulting nodes directly. This reference is set when a node is split in two parts.

In summary, a block-wise RGA node contains the following attributes:

[4]Assuming a balanced, or periodically re-balanced tree.

content : a block of elements. The content can be freed when the node becomes a tombstone in order to minimize memory consumption.
identifier : the RGA unique identifier of the node.
nextLink : the base RGA linked-list reference.
offset : the offset of the node within the node originally inserted by the user.
splitLink : next part of the block in case of a split.
length : the length of the content of the node. When a node becomes a tombstone, the content is freed but the length of the node is saved. This enables to split tombstones and to handle concurrent operations correctly.

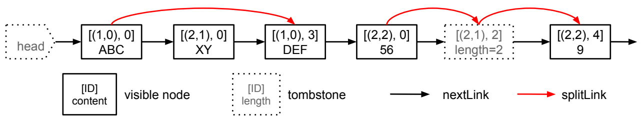

Figure 7: Example of a block-wise RGA structure

Figure 7 shows an example of a block-wise structure for RGA. The state of the structure results from the following modifications, starting from an empty document.

1. A user inserts "ABCDEF" in replica 1, the identifier is $[[1, 0], 0]$.
2. After receiving "ABCDEF", a user inserts "XY" at position 3 in replica 2. The identifier is $[[2, 1], 0]$. The block "ABCDEF" is split into "ABC" with identifier $[[1, 0], 0]$ and "DEF" with identifier $[[1, 0], 3]$.
3. A user inserts "56789" at the end in replica 2, the identifier is $[[2, 2], 0]$.
4. A user deletes "78". Block "56789" is split into "56" with identifier $[[2, 2], 0]$, a tombstone with identifier $[[2, 2], 2]$, and the "9" with identifier $[[2, 2], 4]$.

The function FINDOFFSET returns the node corresponding to the operation's offset. The inputs are the identifier of the node sought and the offset. We start from the base hash table reference and we follow the splitLink chaining. For instance, in the above example, an operation targeting identifier $[[1, 0], 4]$ has an effect on the block "DEF" with identifier $[[1, 0], 3]$ and length 3.

```
1: function FINDOFFSET(identifer, offset)
2:     $node = hashtable.get(identifier)$
3:     while $node.offset + node.length < offset$ do
4:         $node = node.splitLink$
5:     return $node$
```

The splitLink chaining allows good performance but is not mandatory, especially in tombstone-based approaches. Indeed, following the RGA nextLink chaining is still possible. Also, an alternative to a physical splitting of the block, is to keep them intact and to reference the parts of the blocks in the RGA structure. A precise performance analysis should be conduct to determine the optimal solution, especially in term of memory usage, but the whole complexity is equivalent for all alternatives.

4.2 Identifier data structure of RGATreeSplit

We now discuss the addition of the identifier structure to RGASplit. It is similar to the one described in Section 3. However, some generalization is need. First, we need to take block length into account in the FINDPOSINIDENTIFIERTREE function, generalized as follow.[5]

```
1: function FINDPOSINIDENTIFIERTREE(pos)
2:     if $pos == 0$ then
3:         return $head.identifier$
4:     $node = root$                                   ▷ Begins from the tree's root
5:     while $\neg(pos > node.leftWeight() + node.length \wedge pos \leq node.leftWeight() + node.length)$ do
6:         if $pos \leq node.leftWeight() + node.length$ then
7:             $node = node.leftChild()$               ▷ Goes left
8:         else
9:             $pos = pos - (node.leftWeight() + node.length)$
10:            $node = node.rightChild()$              ▷ Goes right
11:    return $[node.identifier, pos + node.offset - node.leftWeight]$
```

Second, when the target node is found, it can be necessary to split it. Indeed, a user insertion applied within a block, will create three new blocks (the two split parts and the new one). A simple solution is first to replace the original block by the new node, to INSERTBEFORE the left part of the split block and to INSERTAFTER the right part. Figure 8 illustrates this procedure. The block "XYZ" is inserted at position 5 in the string "ABCDEFGHIJ". The block "XYZ" replaces and splits the block "DEFG". The two parts "DE" and "FG" are respectively inserted before and after the new block "XYZ".

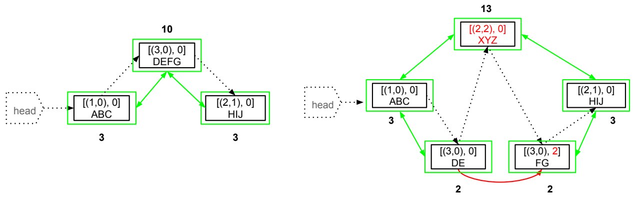

Figure 8: Before and after split

5. EVALUATION AND EXPERIMENTS

In this section, we evaluate our approach, by comparing implementations of the solutions presented above, to existing implementations of the best comparable replication algorithms.

To conduct this experimentation, we use a Java benchmarking framework [6, 7], in which the original authors of TreeDoc, RGA, and Logootsplit implemented their algorithms. Within this framework, we implement four new algorithms: RGA with an identifier structure, Logoot with an identifier structure, RGA with blocks, and RGATreeSplit.

5.1 Implementations

The binary weighted tree identifier structure as presented in Section 3 can be implemented in two ways: always balanced or unbalanced with periodic re-balancing. Furthermore, there exist several algorithms for each. For simplicity, we re-use existing implementations and/or naive algorithms.

[5]With blocks of length 1 and null offsets, this function correspond with its definition in the single element approach.

For the RGA and Logoot identifier structure, we use Apache's TreeList [1]. This Java collection implements a list as a balanced tree. Positions in the list are retrieved using weights similarly to what we have presented above.[6] We add references in both direction between visible CRDT nodes and TreeList nodes. To further simplify the downstream execution implementation, we traverse the height of the tree to find the position of a TreeList node, and we use the base add or remove methods of the TreeList to update the tree. All operations, whether upstream or downstream, executes in $O(log(n))$.

For the RGATreeSplit algorithm, we implement a simple weighted and unbalanced binary tree for our identifier structure. To keep good performance during long-term execution, we periodically re-balance the tree by brute-force copying the visible nodes into a new balanced tree. Each replica is re-balanced independently when the number of operations since the last re-balancing reaches $k\frac{n}{ln(n)}$, with n the number of nodes, and k and arbitrary constant factor.[7]

The re-balancing procedure execution time is taken into account in the experimental results presented below. For shake of implementation simplicity, we call it during downstream execution. However, to ensure real-time responsiveness, the system should call it when idle, and can interrupt it as soon as an upstream or downstream operation is received.

Our implementations, especially the RGATreeSplit could be improved by using a balanced (n-ary-)tree, or a smarter re-balancing algorithm [21], and finer studies to detect when to re-balance the tree. However, the goal of the experiments is only to demonstrate that our approach greatly improves upstream execution performance even using a naive implementation and preserves good downstream performance.

All the implementations and the benchmark platform are publicly available at
https://github.com/score-team/replication-benchmarker/.

5.2 Experiments

The following experiments were executed on the Grid5000 experimental facility [8]. The nodes are powered by Intel Xeon X3440 processors (2.53GHz) with 16GB of RAM and run a Wheezy-x64-big-1.0 operating systems which based on Debian. Moreover the benchmark is not multi-threaded, therefore each algorithm uses only one core.

We randomly generated different types of traces to evaluate the performance of our algorithms. A trace is a sequence of user modifications on a text document on which we apply on all studied algorithms. Thus, all algorithm execute the same set of modifications.

We conduct two series of experiments: the first concerns modifications that only affect one element, the second contains block modifications. In the block series, 20% of the modifications are block modifications and the average length of a block is 20 elements (Gaussian normal distribution with standard deviation $\sigma = 2$).

The other parameters the same for both series. Each series contains six experiments traces containing respectively 5000, 10000, 15000, 20000, 30000 and 40000 modifications. As results may sightly depend on the content of the traces, we generated, three different traces for each experiment. Each trace simulates 10 peers that work on the same text document. Each modification is concurrent to 2 other modifications ($\sigma = 1$). 80% of modifications are insertions, and 20% are deletions following to an existing study in group editing performance [6]. We keep the number of peers fixed since previous experiments [5] have demonstrated that this parameter does not influence the CRDTs performance, contrary to the number of operations.

For each experiment we measured the average upstream execution time, the average downstream execution time, the bandwidth consumed – i.e., the total size of data transmitted between peers – and memory occupation (as the average of memory occupation during the last 100 operations). Time is measured using the Java System.nanotime() method, and the bandwidth and memory occupation are measured by means of a library from the EMMA coverage tool [2]. The algorithms evaluated are the original RGA, Logoot, LogootSplitAVL, TreeDoc, and the four new CRDT described above: RGA with a TreeList identifier structure (called RGATreeList), Logoot with a TreeList identifier structure (called LogootTreeList), RGA with blocks (called RGASplit) and RGATreeSplit.

5.3 Single element series

In this series, each modification affects only one text element. The average execution time for upstream and downstream is presented in Figure 9, and memory occupation is presented in Figure 10.

We observe that RGA and RGASplit achieve roughly the same performance, and the same for RGATreeList and RGATreeSplit. This was expected. The main result here, is that identifier structure drastically improves the upstream performance. RGATreeList, RGATreeSplit and LogootTree are superior to RGA, RGASplit and Logoot respectively. These three algorithms have the best observed performance and are not affected by the length of the experiment. Downstream performance of RGA approaches is sightly degraded, due to identifier structure updating but remains better than non-RGA approaches.

As expected, LogootTree has better performance than Logoot in downstream, but more surprisingly, also has better performance than LogootSplitAVL, even though the latter also uses a balanced binary search tree.

Bandwidth is not affected by the use of the identifier structure, since only base RGA or Logoot operations are sent between peers.[8] Memory occupation is only slightly affected due to the identifier structure. TreeDoc has the lowest memory footprint thanks to the absence of auxiliary structure and fewer tombstones than RGA, but has high bandwidth usage, due to its identifier length.

5.4 Block modification series

In this series, 20% of user modifications are block modifications (average length 20, $\sigma = 2$). The average execution time for upstream and downstream is presented in Figure 11, and memory occupation and bandwidth are presented in Figure 12.

Observe first, in such case, all the block-wise algorithms improve both upstream and downstream performance compared to non-block versions: LogootSplitAVL, RGASplit and RGATreeSplit versus Logoot, RGA, and RGATreeList

[6]In Apache's TreeList, weights (called relative positions) are the size of the left subtree.

[7]We empirically identified $k = 5$ as a adequate factor.

[8]RGATreeList has slightly better bandwidth usage than RGATreeSplit since block operations contains offsets.

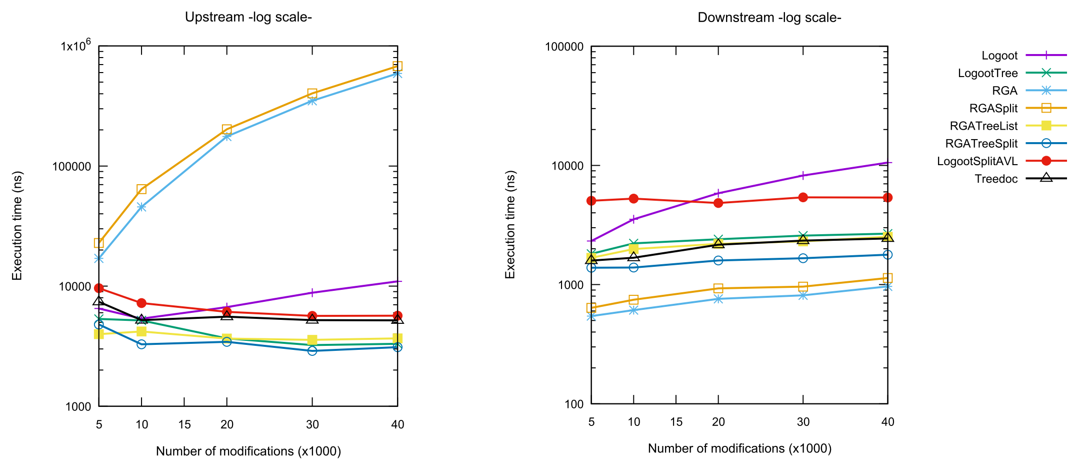

Figure 9: Execution time – single element series

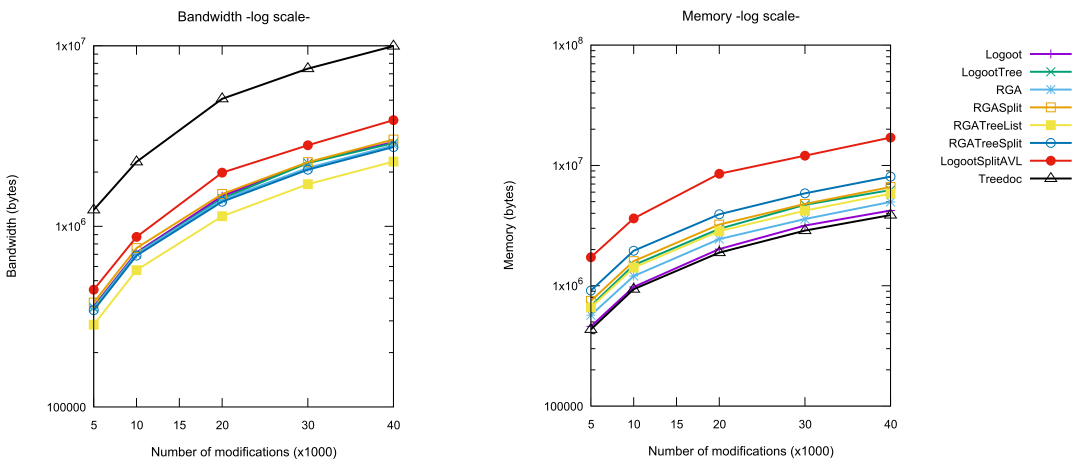

Figure 10: Memory occupation – single element series

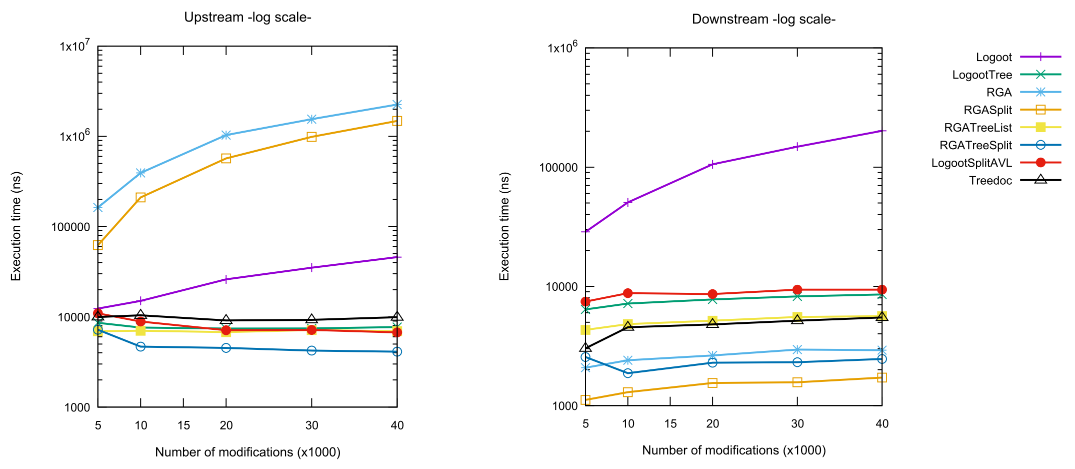

Figure 11: Execution time – block series

respectively. However, block management is not enough to guaranty high responsiveness. Indeed, RGASplit performs quite poorly and deteriorates over time. Furthermore, LogootSplitAVL and TreeDoc have worse downstream performance than the original RGA algorithm while RGATreeSplit has the best overall performance.

LogootTree performs worse than RGATreeSplit, but surprisingly again, it performs similarly to LogootSplitAVL, even without block management.

Concerning memory, block-based approaches improve both bandwidth and memory usage. RGATreeSplit and RGASplit have the best results. We note that TreeDoc obtains better bandwidth performance than in the single element series due to a kind of compression of operations engendered by block modifications.

Concerning ease of implementation, we note that RGATreeSplit and LogootTree obtain better performance than LogootSplitAVL while having a simpler code (respectively 747 and 391 versus 1,255 non-commenting source statement measured using JavaNCSS [4]). Indeed, our approach allows to use an existing or simple data structure contrary to LogootSplitAVL that requires its own and specific data structure.

6. CONCLUSION

In this paper, we present an approach to improve responsiveness of CRDT algorithms. This approach is based on an additional identifier structure that can be adapted to the CRDT algorithms, including block-wise algorithms, to achieve logarithmic complexity. Experiments demonstrate that we reach the best overall performance among existing CRDT algorithms. Even though our implementation is quite naive and can be improved.

More particularly, we significantly improve performance of incorporating users' modifications. This aspect is important since users can directly observe it, and may otherwise lead to user un-satisfaction [10].

We suggest that a similar approach may be applicable to other CRDT data types such as sets or structured documents and also to operational transformations, in order to improve the responsiveness of the TTF solution, and possibly to improve the downstream execution by identifying the operations to transform more efficiently.

Acknowledgment

This research is supported in part by European FP7 project 609 551 SyncFree (2013–2016).

http://syncfree.lip6.fr/ .

References

[1] Apache TreeList. https://commons.apache.org/proper/commons-collections/apidocs/org/apache/commons/collections4/list/TreeList.html.

[2] Emma code coverage tool. http://emma.sourceforge.net/.

[3] Google. drive help: Limits on sharing. retrieved 19 february 2016. URL https://support.google.com/drive/answer/2494827.

[4] JavaNCSS. http://www.kclee.de/clemens/java/javancss/.

[5] M. Ahmed-Nacer. *Evaluation methodology for replicated data types.* Theses, Université de Lorraine, May 2015. URL https://tel.archives-ouvertes.fr/tel-01252234.

[6] M. Ahmed-Nacer, C.-L. Ignat, G. Oster, H.-G. Roh, and P. Urso. Evaluating CRDTs for real-time document editing. In ACM, editor, *ACM Symposium on Document Engineering*, page 10 pages, San Francisco, CA, USA, september 2011.

[7] L. André, S. Martin, G. Oster, and C.-L. Ignat. Supporting adaptable granularity of changes for massive-scale collaborative editing. In *International Conference on Collaborative Computing: Networking, Applications and Worksharing (Collaboratecom)*, pages 50–59, Austin, TX, USA, Oct 2013. .

[8] D. Balouek, A. Carpen Amarie, G. Charrier, F. Desprez, E. Jeannot, E. Jeanvoine, A. Lèbre, D. Margery, N. Niclausse, L. Nussbaum, O. Richard, C. Pérez, F. Quesnel, C. Rohr, and L. Sarzyniec. Adding virtualization capabilities to the Grid'5000 testbed. In I. Ivanov, M. Sinderen, F. Leymann, and T. Shan, editors, *Cloud Computing and Services Science*, volume 367 of *Communications in Computer and Information Science*, pages 3–20. Springer International Publishing, 2013. ISBN 978-3-319-04518-4. .

[9] P. A. Bernstein and S. Das. Rethinking eventual consistency. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 923–928, New York, NY, USA, 2013. ACM.

[10] C. Jay, M. Glencross, and R. Hubbold. Modeling the effects of delayed haptic and visual feedback in a collaborative virtual environment. *ACM Transactions on Computer-Human Interaction*, 14(2), August 2007. ISSN 1073-0516. . URL http://doi.acm.org/10.1145/1275511.1275514.

[11] M. Leţia, N. Preguiça, and M. Shapiro. CRDTs: Consistency without concurrency control. In *SOSP W. on Large Scale Distributed Systems and Middleware (LADIS)*, pages 29–34, Big Sky, MT, USA, October 2009. ACM SIGOPS.

[12] F. Mattern. Virtual time and global states of distributed systems. In *International Workshop on Parallel and Distributed Algorithms*, pages 215–226, Château de Bonas, France, October 1989. Elsevier Science Publishers.

[13] G. Oster, P. Urso, P. Molli, and A. Imine. Data Consistency for P2P Collaborative Editing. In *Conference on Computer-Supported Cooperative Work (CSCW)*, pages 259–267, Banff, Alberta, Canada, nov 2006. ACM Press.

[14] G. Oster, P. Urso, P. Molli, and A. Imine. Tombstone transformation functions for ensuring consistency in collaborative editing systems. In *International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2006)*, Atlanta, Georgia, USA, November 2006. IEEE Press.

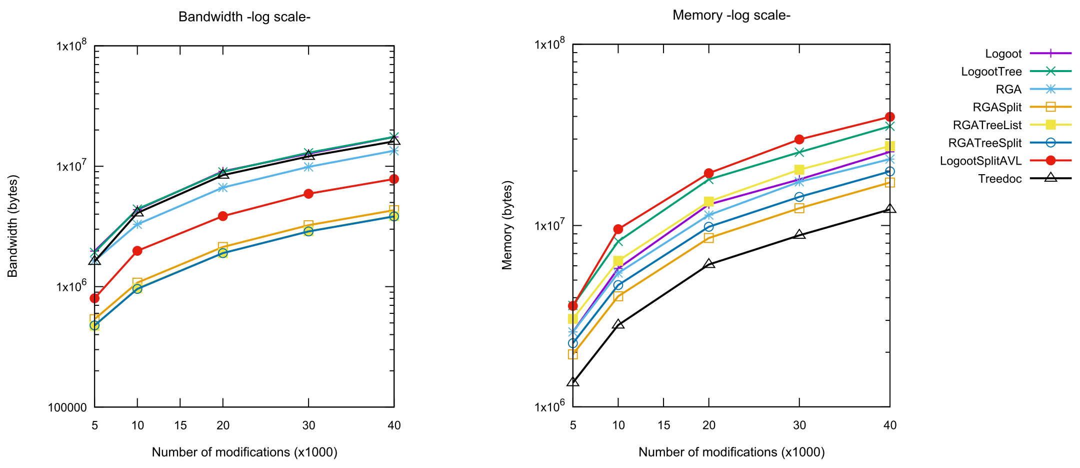

Figure 12: Memory occupation – block series

[15] N. M. Preguiça, J. M. Marquès, M. Shapiro, and M. Leţia. A commutative replicated data type for cooperative editing. In *International Conference on Distributed Computing Systems (ICDCS)*, pages 395–403, Montréal, QC, Canada, 2009. IEEE Computer Society.

[16] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM*, 33(6):668–676, 1990.

[17] M. Ressel, D. Nitsche-Ruhland, and R. Gunzenhäuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *Conference on Computer supported cooperative work (CSCW)*, pages 288–297, Boston, MA, USA, 1996.

[18] H.-G. Roh, M. Jeon, J.-S. Kim, and J. Lee. Replicated abstract data types: Building blocks for collaborative applications. *Journal of Parallel and Distributed Computing*, 71(3):354 – 368, 2011. ISSN 0743-7315.

[19] Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1):42–81, 2005. ISSN 0360-0300.

[20] B. Shneiderman. Response Time and Display Rate in Human Performance with Computers. *ACM Computing Surveys*, 16(3):265–285, September 1984. ISSN 0360-0300. . URL http://doi.acm.org/10.1145/2514.2517.

[21] Q. F. Stout and B. L. Warren. Tree rebalancing in optimal time and space. *Commun. ACM*, 29(9):902–908, September 1986. ISSN 0001-0782. . URL http://doi.acm.org/10.1145/6592.6599.

[22] C. Sun, X. Jia, Y. Zhang, Y. Yang, and D. Chen. Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 5(1):63–108, March 1998. ISSN 1073-0516.

[23] N. Vidot, M. Cart, J. Ferrié, and M. Suleiman. Copies convergence in a distributed real-time collaborative environment. In *Conference on Computer supported cooperative work (CSCW)*, pages 171–180, Philadelphia, PN, United States, 2000. ACM. ISBN 1-58113-222-0.

[24] S. Weiss, P. Urso, and P. Molli. Wooki: a P2P Wiki-based Collaborative Writing Tool. In *International Conference on Web Information Systems Engineering (WISE)*, pages 503–512, Nancy, France, December 2007. Springer.

[25] S. Weiss, P. Urso, and P. Molli. Logoot: A scalable optimistic replication algorithm for collaborative editing on P2P networks. In *29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009)*, pages 404 –412, Montréal, QC, Canada, jun. 2009. IEEE Computer Society.

[26] W. Yu. A string-wise CRDT for group editing. In *Proceedings of the 17th ACM International Conference on Supporting Group Work*, GROUP '12, pages 141–144, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1486-2. . URL http://doi.acm.org/10.1145/2389176.2389198.