Controlled conflict resolution for replicated document

Stéphane Martin
Université de Lorraine
CNRS – INRIA – LORIA
54500 Vandoeuvre-lès-Nancy, France
Email: stephane.martin@loria.fr

Mehdi Ahmed-Nacer
Université de Lorraine
CNRS – INRIA – LORIA
54500 Vandoeuvre-lès-Nancy, France
Email: mehdi.ahmed-nacer@loria.fr

Pascal Urso
Université de Lorraine
CNRS – INRIA – LORIA
54500 Vandoeuvre-lès-Nancy, France
Email: pascal.urso@loria.fr

***Abstract*—Collaborative working is increasingly popular, but it presents challenges due to the need for high responsiveness and disconnected work support. To address these challenges the data is optimistically replicated at the edges of the network, i.e. personal computers or mobile devices. This replication requires a merge mechanism that preserves the consistency and structure of the shared data subject to concurrent modifications.**

In this paper, we propose a generic design to ensure eventual consistency (every replica will eventually view the same data) and to maintain the specific constraints of the replicated data. Our layered design provides to the application engineer the complete control over system scalability and behavior of the replicated data in face of concurrent modifications. We show that our design allows replication of complex data types with acceptable performances.

***Index Terms*—optimistic replication, replicated document, collaborative editing**

I. Introduction

Replication allows accessibility of shared data in collaborative tools (such as Google Docs) and mobile applications (such as Evernote or Dropbox). Indeed, collaboration is achieved by different distinct sites that work independently on a replica, i.e. a copy of the document. Due to high responsiveness and disconnected work requirements, such applications cannot use lock or consensus mechanisms.

However, the CAP theorem [3] states that a replicated system cannot ensure strong Consistency together with Availability and Partition tolerance. In such applications, where availability is required by users and partition is unavoidable, a solution is temporal divergence of replicas, i.e. to use optimistic replication. Of course, at the end of the modification process, users aim to have the same document. This kind of consistency model is called "*eventual consistency*" which guarantees that if no new update is made to the object, eventually all accesses will return the same value. To obtain eventual consistency, a particular merge procedure that handles conflicting concurrent modifications, is required.

We consider that two concurrent modifications *conflict*, if, once both integrated, they violate the structural constraints of a data type. For instance, with a replicated structured document, adding concurrently two titles conflicts if the document type accepts only one title. To obtain a conflict-free replicated data type, the merge procedure must make an arbitrary choice (such as: appending the titles, "priority-replica-wins", "last-writer-wins", etc.). Moreover, every replica must make independently the same choice. Conflict resolution is also a question of scalability and performances since different choice procedures may have different computational complexities.

Unfortunately, eventual consistency is more difficult to achieve facing complex conflict resolution as demonstrated by the numerous proposed approaches that fail to ensure it for simple plain text document [7], [13]. Indeed, more the data type is complex, more conflicts appear. For instance, in a hierarchical document, modifications such as adding and removing an element, or adding a paragraph while removing the section to which it belongs, or setting concurrently two titles conflict.

We propose a framework that decouples eventual consistency management from data type constraints satisfaction. Our framework is made of layers. A layer can use the result of one or more independent layers. The lowest layer hosts the replicated data structure and are in charge to merge concurrent modifications. These lowest layers encapsulate an existing eventually consistent data type from the literature. Other layers are in charge to ensure a constraint on a data type. It does not modify the inner state of the replicated data but only computes a view that satisfies the constraint.

Our framework manages each conflict type independently while assuring eventual consistency. Thanks to layered design, any combination of conflict resolution is designable, giving to the application the entire control on the system scalability and behavior of the replicated data in face of concurrent mutations.

II. Motivation

Our approach is based on the observation that obtaining eventual consistency while ensuring complex constraints on a data type is difficult. Thus, we propose to decouple eventual consistency from data integrity insurance trough layers.

To illustrate the behavior of such a decoupling, let's imagine a replicated file system. Ensuring eventual consistency of a file system is complex [5], while ensuring eventual consistency of a set can be achieved in numerous ways with quite simple algorithms. For instance, [17] defines multiple replicated sets with different behaviors and performances.

So we can imagine a file system as the set of absolute paths present in the file system.

1) A first layer contains the set of independent couples $(path, type)$ which are elements present in the file system. Types can be directory or file. This layer communicates with the first layer of the other replicas. It transmits simple messages that correspond to an addition or a

COLLABORATECOM 2012, October 14-17, Pittsburgh, United States
Copyright © 2012 ICST
DOI 10.4108/icst.collaboratecom.2012.250461

suppression in the set. This layer ensures alone eventual consistency by merging these messages.

2) The second layer is in charge of producing a tree from the set of paths. To produce this tree, it must ensure the constraint that all nodes are accessible by the root. Indeed if a replica removes a directory, while another adds a file into this directory, the path to the file is present in the set while the path to the directory is not. Such a layer may drop this "orphan" file or place it under some special "lost-and-found" directory (see Section IV-B).
3) The third layer is in charge of producing a file system from the tree. It satisfies the unique name constraint on a directory. Indeed, a directory may contains two children (one directory and one file) added concurrently with the same name. Such a layer may rename elements, or enforce specific name when adding an element (files and only files must have an extension, such as `.java`).

Replicated file systems (and some other complex data types), already exist in the literature. The advantage of our model is twofold. The first advantage is that only the first layer is in charge of merging concurrent operations. For the other layers, the data is handled as local data, simplifying the eventual consistency issues. The second advantage is the modularity of the approach. A layer that provides a data type can be freely substituted by another implementation. Thus, our approach can provide many different behaviors, while each existing solution proposes only one or a small number of different behavior(s) with an associated performance level which could not be appropriate to every collaborative application context.

III. Layered data types

We define a data type as an object with a two methods interface: i) the "lookup" method returns the data type state; ii) the "modify" method performs modifications in the data type state.

A *replicated data type* is a data type with a communication interface to merge its state with other replicas. Concretely, on each update invocation from an application, the replicated data type sends to another replica a message that represents the local modification. A replicated data type which receives such a message, integrates it on its own state. We require that a replicated data type ensures eventual consistency. This means that, after all modifications were performed, the invocation of the lookup method eventually returns the same result.

First, we encapsulate an existing eventually consistent data type in a *replication layer*. This kind of layer is the bottom layer of our model. It ensures communication between replicas and manages concurrent modifications. The other kind of layer we define is the *adaptation layer* that uses the data provided by one or more layers and ensures a particular constraint on the data type. An adaptation layer can be placed on top of one or more layers that can be replication or adaptation layers.

As presented in Figure 1, the generic computational aspect of our model is quite simple. When an application modifies a data type, it calls the higher layer modify function. The higher layer adapts the given local operation into one or more local operation(s) applied on the layer just below. This layer will itself adapt these local operations for the third layer, and so on until the replication layer. Only the replication layer is in charge to communicate local updates to other replicas and to merge local and remote modifications. When the application asks for the value of the data type, it calls the higher layer lookup interface. The layer calls the lookup interface of the layer just below and computes a result corresponding to the application needs.

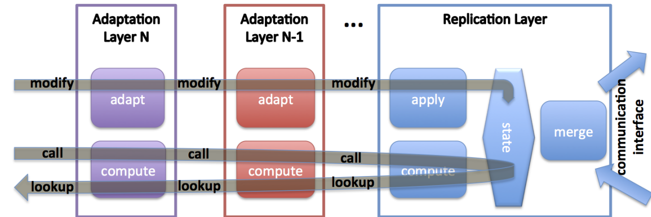


Fig. 1. Layers

The lookup method of an adaptation layer recomputes totally its result from the inner layer(s) lookup invocation(s) result(s). *This computation does not affect the inner-layer state, if any.* Assuming this computation is deterministic and that the below layer(s) ensure(s) eventual consistency, we can prove straight-forwardly that the adaptation layer provides an eventually consistent data type.

Such a computation must be done when a view is requested, but only if the inner data was modified since the last request. This is adapted to *state-based* replication mechanisms [16] (such as version control systems). State-based replication mechanisms transfer their whole state to other replicas, thus, fewer merge occurs but each merge may modify up to the whole state of the data.

However, for *operation-based* replication mechanisms [16], we should define incremental adaptation layers. Operation-based replication mechanisms sends update operations (or differences).

Incremental Layers

An incremental adaptation layer stores the state of the data type that will be returned to the application. It modifies this data type each time its inner layer state is modified, following an observer design pattern, see Figure 2. Therefore, it modifies only a part of the data type. Potentially, an incremental lookup has better performances. Eventual consistency can be ensured by an equivalence between the incremental lookup and some non-incremental lookup. Anyway, as non-incremental layers, incremental layers computations do not affect their inner-layer state.

Even if incremental layers seem more adapted to operation-based replication mechanisms, any combination of layers can be constructed. Indeed, a state-based replication layer that notifies changes to its observers can be used below an incremental layer. Also, an incremental layer can be used below a non-incremental one.[1]

[1]This last combination can be useful when no incremental solution is available for a given constraint (for XSD schema repairing for instance).

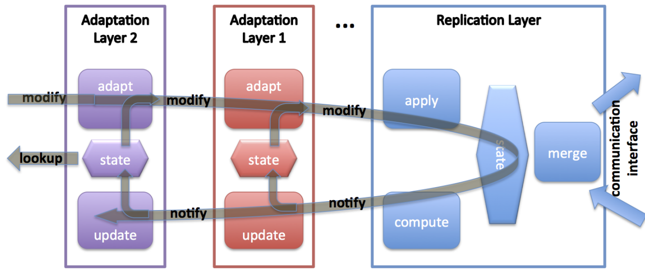


Fig. 2. Incremental layers

IV. Examples

This section presents several examples of data types that can be obtained using our framework. Due to space limitation, only some of them will be completely detailed.

A. Text data type

In this section, we show how to obtain a text data type, i.e. an ordered sequence of elements (lines, character, or paragraphs, etc.). Beside its apparent simplicity, this a non-trivial problem as evidenced by the huge literature on the subject: [13], [24], [14]. The challenge comes from puzzles such as TP2-puzzles [22], where two elements are inserted concurrently just before and after an element which is being deleted. Since deleted elements no longer separates the inserted ones, they may be swapped.

We present a composition of two layers to ensure the ordering constraint. We use a set element associated with an un-mutable ordering information called *position identifier (PI)*.

As presented in Figure 3, we define an adaptation ordering layer on top of a set replication layer. The set contains elements coupled with a position identifier (PI). For example, the sequence 'AC' corresponds to the set $\{('A', p_a), ('C', p_c)\}$. To add 'B' between 'A' and 'C', we must forge p_b such that $p_a \prec p_b \prec p_c$. The set becomes $\{('A', p_a), ('C', p_c), ('B', p_b)\}$. The "lookup" function uses the total order between PIs to compute the ordered sequence 'ABC'.

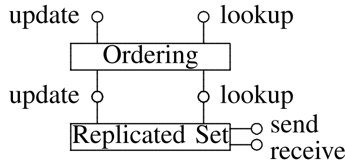


Fig. 3. Text data type using sets

Position identifiers are defined in a dense space equipped with a total ordering relation. The total order ensures that any pair of elements appear in the same order on each replica. The space is dense to allow insertion of an element between any two others.

In the literature, such spaces already exist. Logoot [26] and FCEdit [10] use integer or strings concatenated with unique identifiers; the ordering relation is a lexical ordering. The Treedoc [14] algorithm uses depth-first search on a binary tree as ordering. The position identifier of Treedoc is a path in this tree with unique identifiers to distinguish two similar paths.

The algorithms cited above generate unique identifier (unique for all replicas). These identifiers are unique to ensure eventual consistency. So, when a same element is added concurrently at the same place, it is inserted twice with two different identifiers. For instance, if two users aim to correct the word 'ct' into 'cat', these algorithms add two 'a' and word becomes the 'caat'.

In our framework, the set ensures the eventual consistency. So, we can relax the uniqueness of the position identifier. For instance, in Logoot positions, the operation timestamp could be replaced by the element it-self. Thus, we will obtain a different behavior than the above algorithms since the concurrent insertion of two same element at the same position will lead to a unique appearance.[2] This behavior may seem more natural to users and is the behavior (called "accidental clean merge") of most of the control version system software (Git, SVN, etc.). Obviously, all editing conflicts cannot be resolved using such approaches. However, thank to our layered framework, one can add a semantic correction layer such as [4] above our own layers.

We define a couple object which contains a position identifier and a label. We assume that each ordering algorithm implements the interface described in Figure 4.

```
1  interface Ordering<L>{
2    /*gets the position where the pi will be inserted in pis list .*/
3    int getPos(PI pi, L label, List <Couple> pis);

5    /* returns an ordered list built from set of couple.*/
6    List <Couple> order(Set <Couple> cs);

8    /* generate position identifier with c1 < returned pi < c2 */
9    PI generatePI(Couple c1, Couple c2);
10 }
```

Fig. 4. Interface of ordering algorithm.

We define the Ordering layer in two versions : the non-incremental version in figure 5 and the incremental version in figure 6.

The difference between two versions is the presence of the inner state. The non-incremental layer must order the set to have a lookup or to modify the sequence, while the incremental version uses its inner state to avoid re-computation.

The application or upper layer invokes the modify function of ordering layer with operation as argument. This operation can be an add or delete operation.

For both layer versions, the "add" operation parameters are an element (line, characters, ...) and an integer position. In this case, the layer gets the previous and next element PI from the lookup list . It generates a position identifier help with ordering algorithm between two PIs ($generatePI$) (l.9 fig. 5 and fig. 6) and store the couple with added element and generated position identifier in the inner set (l.15). In case of delete, the operation contains only the element position to remove. The modify function gets the element from lookup

[2]Two 'a' added sequentially, for instance, in the word 'aardvark', will have different PIs.

list (l.12) and forges the operation for deletion from the inner set (l.13).

The difference between incremental and non incremental version is: for non-incremental version, the lookup list is built from the inner set (using of the ordering algorithm) for each call (l.6 fig. 5); while the lookup of the incremental version returns its own up-to-date list (l.3 fig. 6). In incremental case, when the inner set is modified by local or remote operation the layer is notified and update function is called. The update function places the new element in the layer state in position given by ordering algorithm (l.22 fig 6) or deletes from layer state the element which, contains the position (l.24).

```
1  class OrderingLayer{
2    Ordering algo;
3
4    void modify(SequenceOperation change){
5      SetOperation op;
6      List <Couple> list = lookup(); //Reordering
7      if (change.type == add){
8        int pos=change.position;
9        PI pi = algo.generatePI( list .get(pos), list .get(pos+1));
10       op = new SetOperation(add, new Couple(change.label, pi));
11     }else{     //del operation
12       Couple c = list .get(change.position);
13       op = new SetOperation(del, c);
14     }
15     innerSet.modify(op);
16   }
17
18   list lookup(){
19     return algo.order(innerSet.lookup);
20   }
21 }
```

Fig. 5. Non-Incremental Sequence layer

B. Unordered tree

In this section, we design replicated unordered trees. The unordered tree node contains a $Label \in \Sigma$, a father and a set of children. The root is a special node without father and label.

As presented in Figure 7, to provide this tree, the layer uses a set of paths. More formally, we define a path as a sequence of label: $p \in Path, p = l_1 l_2 \cdots l_n, l_i \in \Sigma, \forall i \in [1..n]$. Each path in this set represents a node. For example, the tree drew in figure 8 is represented by $\{a, ab, ac\}$. In this example, when the replica 2 adds c under b the word abc is added in inner set. When the replica 1 removes b, the word ab is deleted in inner set. In second time, both replica exchange these operations and those states become $\{a, ac, abc\}$. This set does not represent directly a tree because the node b is not present and has one child. We call the path abc, respectively the node represented by this path, an orphan path respectively an orphan node. In this case, there are different ways to adapt the tree from the path set. Each way makes a different behavior.

In Figure 9, we present four different behaviours: i) Skip behaviour does not return orphan nodes; ii) Reappear behaviour returns the orphan node at their original path; if the node abc is finally deleted, ab disappears; iii) Root behaviour places orphans under a specific directory (root or lost-and-found); iv) Compact behaviour moves c node under node a, both ac are merged.

```
1  class OrderingLayer{
2    Ordering algo;
3    List <Couple> list;
4
5    void modify(SequenceOperation change){
6      SetOperation op;
7      if (change.type == add){
8        int pos=change.position;
9        PI pi = algo.generatePI( list .get(pos), list .get(pos+1));
10       op = new SetOperation(add, new Couple(change.label, pi));
11     }else{       // del operation
12       Couple c = list .get(change.position);
13       op = new SetOperation(del, c);
14     }
15     innerSet.modify(op);
16   }
17
18   void update(SetOperation change){
19     Couple couple = change.label
20     if (change.type == add){
21       int pos = getPos(couple.pi, list );
22       list .add(pos, couple);
23     }else{   // delete
24       list .remove(couple);
25     }
26   }
27
28   list lookup(){
29     return list ;
30   }
31 }
```

Fig. 6. Incremental Sequence layer

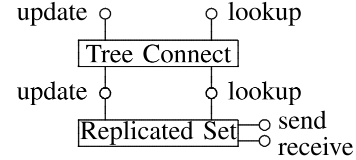


Fig. 7. Layered tree

More formally, we call an orphan path, a path in the inner set lookup (LS) that has a prefix which is not in LS. We start by adding all non-orphan paths of LS to lookup of the tree (LT). Then, we treat the orphan paths in LS in length order (shortest first, then Σ order). Considering each orphan path $a_1 a_2 \dots a_n \in LS$ with $\forall i \in [1, n]$. $a_i \in \Sigma$, we can apply the following *connection policies* :

skip: *drops the orphan path.*

reappear: *recreates the path leading to the orphan path.* We add all $a_1 \dots a_j$ with $j \in [1, n]$.

root: *places the orphan subtree under the root.* We add $a_j \dots a_n$ to LT with j such that $a_1 \dots a_{j-1} \notin LS$ and $\forall k \in [j, n]$, $a_1 \dots a_k \in LS$.

compact: *places the orphan subtree under its longest non-orphan prefix.* We add $a_1 \dots a_m a_j \dots a_n$ to LT with j and m such that $m < j$ and $a_1 \dots a_m \in LT$ and $a_1 \dots a_{m+1} \notin LS$ and $a_1 \dots a_{j-1} \notin LS$ and $\forall k \in [j, n]$, $a_1 \dots a_k \in LS$.

Using any of the above policies ensures that the lookup trees presented to the client by any layered tree are eventually consistent. Indeed, we assume that the inner set is eventually consistent. Since the tree lookup is deterministically computed

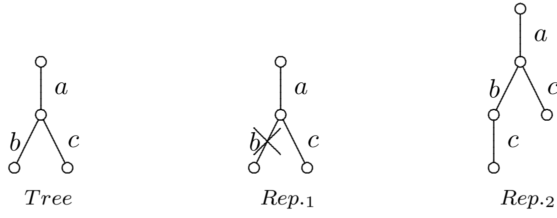


Fig. 8. Concurrent operations in replicated trees

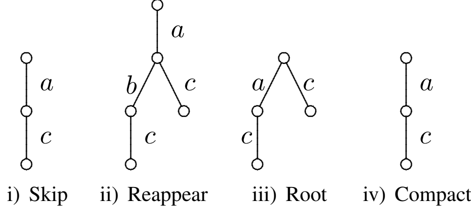


Fig. 9. Different behavior for resolving conflict in trees

each time the set is modified, this tree lookup is eventually consistent. Of course, re-computing the whole tree lookup is not efficient, and we can define incremental version of the four policies. We present here the reappear and root incremental policies[3].

1) Reappear Policy: The reappear algorithm presented in Figure 10, uses a set of "ghosts". When an orphan node is added in the inner set, the policy recreates its ancestors as ghosts by browsing through the path. When a node with children is removed in the inner set, this node is not removed in the tree. But it is just marked as a ghost. Ghosts are unmarked when the node path is re-added in the set. All leaf nodes marked as "ghost" are recursively removed until there was nothing left. In our example b is a ghost (see Fig. 9ii)).

The update function for the reappear algorithm is written in figure 10. The modify function converts a path of lookup to a path for inner set. By chance, in this policy the path is not modified. Thus, add operation is not modified. However, the delete operation must delete the subtree. In this case, the algorithm looking for all children to remove from the inner set.

The update function accepts an operation which contains type of operation (add or delete) and a path. The path designates the new label or the label to remove; and where to add the new node or the node to remove. The constructor prototype of this operation is `Operation(Optype optype, Path path)`.

2) Root policy: The root algorithm moves all orphan nodes to the root or some special "lost-and-found" directory. The update function of this algorithms is presented in figure 11. When two nodes with same label are orphans, the orphans are merged and the view presents only one node under the root. The internal state of the connecting layer is a decorated tree. Nodes are decorated with $Paths$, the set of original paths leading to the node. The connecting layer also uses $path2node$, a map to link original paths to the node objects.

When a node is added, if this path is prefix of orphans

[3]Due to space limitation, skip and compact policies are not presented but are implemented in our open-source framework.

```
1  void Update(SetOperation change) {
2    Path path = change.content;
3    if (change.type == add) {          // Adds Operation.
4      Label last = path.removeLast();   // Computes the father path
5      Node father = tree.getNode(Path); // Get father from path
6      if (father == null) {             // If node is Orphan node
7        Node node = tree.root;
8        Path nPath = new Path();
9        for (Label l: path) {
10         Node c = node.getChild(l);
11         if (c == null) {
12           c = tree.add(node, l);        // reappear as ghost
13           ghosts.add(c);
14         }
15         node = c;
16       }
17       tree.add(node, last);
18     } else {                           // Not Orphan Node
19       Node node = tree.add(father, last, path);
20       ghosts.remove(node);
21     }
22   } else {                             // Del Operation
23     Node node = tree.getNode(path);
24     if (node.children.isEmpty()) {
25       do {                             // Purge ghosts
26         Node father = node.getFather();
27         ghosts.remove(node);
28         tree.del(node);
29         node = father;
30       } while (ghosts.contains(node) && node.children.isEmpty());
31     } else {                           // Node has children
32       ghosts.add(node);                // Become a ghost
33     }
34   }
35 }
```

Fig. 10. Update function for incremental reappear policy

paths, then all corresponding nodes are reattached by move function. The move function looks for all prefixes in $Paths$ of all children of the root node and removes them. It adds the node to reattach and adds this prefix. All nodes with empty $Paths$ are deleted.

The modify function browses the tree through a path, takes the last node and forges the operation with the $Paths$. For example, in case of add operation, the modify function adds each element of $Paths$ concatenated by new label and in case of delete operation it deletes every path present is $Paths$.

In our example9iii), when b is deleted and c is added under b, the c is moved under the root. However, a node c is already under the root. Two nodes c fusion and c contains the path c and path abc.

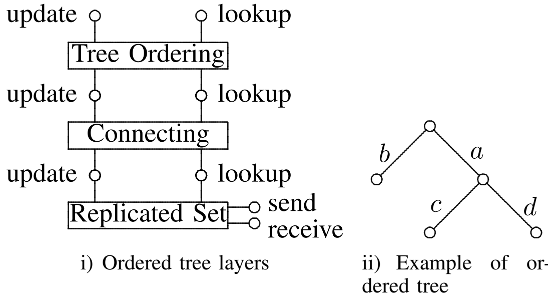
C. Ordered Tree Data Type

In this section, we design ordered tree. As presented in Figure 12i), we directly use the unordered tree data structure and we add an ordering layer. To order the children of a node we use *Position Identifier* (introduced in Section IV-A). We mark all labels with a position identifier. Therefore, the nodes become totally ordered. The set of paths, managed by the replication layer, is represented by $p = (l_1, p_1) \cdots (l_n, p_n)$ with $l_i \in \Sigma$ a label and p_i a position identifier. However, the modify interface of the tree ordering layer must be independent of the chosen ordering algorithm. The ordering layer interface receives operation based on a path defined on integer position without label (ex : 2.4.5.1). Each integer position corresponds

```
1  //move node identified by path from srcFather to dest
2  void move(Node srcFather, Node dest, List path) {
3    for (Node child: srcFather.getChildren()) {
4      /* Make path with prefix and label */
5      List childPath = new Path(path, child.getValue());
6      /* node contains good prefix*/
7      if (child.Paths.contains(childPath)) {
8        child.del(childPath);
9        Node node = dest.add(child.label, childPath);
10       move(child, node, childPath);
11       path2node.put(childPath, node);
12     }
13   }
14 }

16 void Update(SetOperation change) {
17   Path path = change.getContent();
18   if (change.getType() == add) { // Add
19     Path fatherPath = path.clone();
20     Label last = fatherPath.removeLast();
21     Node father = tree.path2node.get(fatherPath);
22     if (father == null){                    // Orphan node
23       father = tree.root;
24     }
25     Node node = father.add(last, path);
26     tree.path2node.put(path, node);
27     move(tree.root, node, path); // Reattach adopted
28   } else {                                 // Remove
29     Node node = tree.path2node.get(path);
30     tree.path2node.remove(path);
31     move(node, root, path);
32     tree.del(node, path); //remove if paths is empty
33   }
34 }
```

Fig. 11. Update function for Incremental root policy



i) Ordered tree layers

ii) Example of ordered tree

Fig. 12. Ordered tree

to a children number in the ordered tree. For example, consider the tree on the Figure 12ii). The inner replicated set contains $\{a_{p_a}, b_{p_b}, a_{p_a}c_{p_c}, a_{p_a}d_{p_d}\}$ with $p_b \prec p_a$ and $p_c \prec p_d$. The ordered path leading to c is 2.1.

In fact, in a similar way as an unordered tree, the layer state contains nodes, but, each node, contains additionally the position identifier and each child is ordered by chosen ordering algorithm.

The modify function converts an integer position path $j_1 ... j_n$, $j_i \in \mathbb{N}$ into a path containing couples of label and position identifier. It browses through the tree and pushes the couple of label and position identifier for each node, until the last but one. If the operation is an add, the last position identifier pi_n is generated by ordering algorithm. The generated position identified by pi_n where $pi_{j_n} \prec pi_n \prec pi_{j_n+1}$ if j_n is the last position of path and p_{j_n} is position identifier in position j_n. This holds as the last position of the path is the new node. In case of delete operation, the modify function converts all of path.

The update function receives a path with label and positions identifier from the inner set. It browses through the tree until the last node but one of the path. The algorithm can use a Hashmap or dichotomy algorithm to find a node in the children ordered list. In case of add operation, the update function adds the new node in good place defined by ordering relation. In case of delete, the update function deletes the node.

D. Extension to schema

In this section, we consider ordered trees with schema (such as XSD or DTD for XML documents). Concurrent modifications can produce a tree which does not respect the schema. For example, consider a schema which accepts zero to one title element. If two users add concurrently a title, they will create two title nodes in the internal tree data type. To fix it, we add a new layer called schema repair. In this layer (see Fig. 13), lookup interface calls a repair algorithm (such as [19]) to return a valid tree. The "modify" must ensure that each operation generated on lookup view is valid on internal data structure [11].

For example, in an agenda, we assume that under a participants node, there is one or more person. If there are two persons and two replica delete one distinct, then each replica has generated an operation compatible with the schema. However, at the end, no person is present. The repairing algorithm has two choices: add a person or delete participants markup. However, if the schema needs participants under event node, then the algorithm chooses to add a person. In this case, each replica will repair by adding a person node. This addition will not be passed to the inner data type. In our model the lookup or update does not modify the inner state. When a node is added under the virtual person node like a name, the modify function creates the missing node before to add name, because the participant is not present in the inner state. An addition under non-present node implies a fix in tree layer. If the chosen policy is different from reappear the result is not compliant with the schema and the tree will be fixed again.

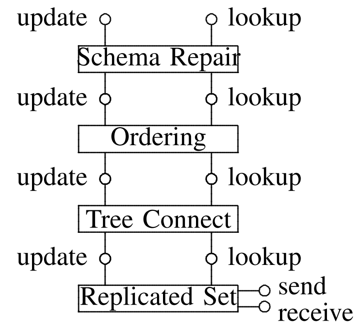


Fig. 13. Tree with schema

Optimization with DTD schema: The particularity of DTD schema is a poor language. An add or remove of a node can invalidate only a part of the tree. It's possible to use a subquadratic algorithm [27] to approximate regular expression

matching on children to fix the tree. All added edges by this algorithm could be added with a template of recursive valid children.

E. Directed acyclic graph

This kind of data type can be used for task dependence representation, such as Gantt or Pert diagram. In this example, we use two replicated sets: a set of nodes and a set of edges. The nodes represent the tasks, and the edges represent the dependency between the tasks. Two concurrent dependency additions conflict when they introduce a cycle in the graph. An un-cycling layer resolves such conflict by traversing the graph using a breath-first search (see Fig. 14).

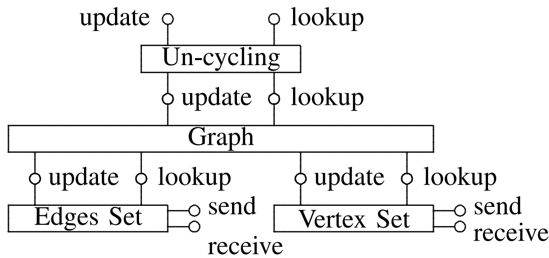


Fig. 14. Directed acyclic graph

V. Experimental Evaluation

To evaluate the performances of our approach, we have implemented it in the framework ReplicationBenchmark developed in Java, available on the GitHub platform [4] under the terms of the GPL license. In this framework, we have implemented different set layers, different ordering algorithms, the connecting layer with the four policies described Section IV-B and the tree ordering layer described Section IV-C.

The framework follow our layer structure. For instance, creating a ordered tree based on a reappear policy and a counter replicated set is done by the following Java expression: new PositionIdentifierTree(new WordTree(new ReappearPolicy(), new CounterSet())). The framework provides base classes for common elements, such as a version vector, set, tree and ordered tree operations.

The framework provides a simulator that generates a trace of operations randomly, according to provided parameters such as trace length, percentage of adding, removing, number of replica, communication delay, etc. It also provides a controlled simulation environment that replays a trace of operations and measures the performance of the replicated algorithms. The simulation ensures that each replica receives operations in the order as defined in the logs. The framework lets replicas of every algorithm generate operations in its own formats for the given trace operations provided from the simulated logs. The trace obtained to run our experiment has 30000 operations with 88% of insertions and four replicas. The trace is available on the web [5].

We denote a *local operation* an operation appearing in the trace. Such operation will be given to the modify interface. For ordered tree, operations are insertion of an element or deletion of a sub-tree. A local operation is divided into one to several *remote operation* that the simulation sends to remote replicas. A replica, therefore, executes remote operation. We measure the net execution time of local and remote operations for each algorithm. The framework uses java.lang.System.nanoTime() for the measurement of execution time of each local operation and each remote operation.

To obtain a correct result, we ran each algorithm on traces three times on the same JVM execution. We also measure the size memory occupied by each algorithm. We serialize each document replica by using Java serialization after each hundred operations generated, and measure the size of the serialized object.

All executions are run on the same JVM, on a dual-processor machine with Intel(R) Xeon(R) 5160 dual-core processor (4Mb Cache, 3.00 GHz, 1333 MHz FSB), that has installed GNU/Linux 2.6.9-5. During the experiment, only one core was used for measurement. All graphics are smoothed by bezier curves.

Before the representation out result of the experiment, we briefly describe some representative algorithms that exist and which we will compare our approach.

A. TreeOpt and OTTree

TreeOPT (tree OPerational Transformation) [6] is a general algorithm designed for hierarchical documents and semi-structured documents. Each node contains an instance of an operation transformation algorithm [2], [15], [21]. The algorithm applies the operational transformation mechanism recursively over the different document levels. In our experimentation, we have used this algorithm with SOCT2 [20] algorithm and TTF (Tombstone Transformation Functions) approach [13]. For little optimization, we save only insertion operation in log of SOCT2.

The OTTree, an unpublished algorithm, uses only one instance of SOCT2 for entire the tree (not on each node) and TTF on each children list. The operation of TTF and its integration function were modified to include the path information.

B. FCEdit

FCEdit [10] is a CRDT designed for collaborative editing of semi-structured documents. It associates to each element a unique identifier. FCEdit maps $identifier \rightarrow node$. So it uses just an hash table to find an element in the tree. Each child is ordered by a position identifier. Unlike OTtree, FCEdit does not need to store an element in tombstone. The elements are really deleted from tree making it more efficient in memory.

In the following, we present behaviors of each ordered tree algorithms executed on simulated traces with the different policies described in Section IV.

C. Execution times

In [8], studies have shown that users can comfortably observe modifications on their application if the local and remote response time do not exceed $50\,ms$. In this section,

[4]http://github.com/score-team/replication-benchmarker

[5]http://www.loria.fr/~mahmedna/trace

we address an experimental evaluation of algorithms based on our layer structure, compared to existing ones to verify if this design is suitable for real time collaborative applications.

1) Skip policy:

a) Local operations: The average execution time of Local operations are presented in figure 15.

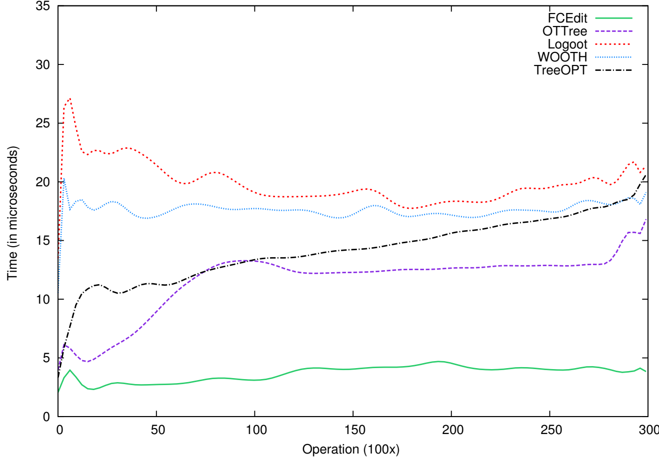


Fig. 15. Execution time for algorithms with Skip -local-

The performances of the algorithms based on the layer structure (Logoot and WOOTH) are the less efficient compare to the algorithms that exist (OTTree and FCEdit), but it remains stable throughout the experiment. They do not exceed $30\mu s$, and thus $50\,ms$, what makes them acceptable for the users. The performances of OTTree and TreeOPT based on SOCT2 algorithm degrade in the beginning of experiment, since the rate of insertion is greater than the deletion, the tree becomes quickly large. TreeOPT makes an operation by each element of the path contrary to OTTree. This explains that the difference of both algorithms depends of tree depth. After the 100 000 operations, the majority of algorithms become stable. FCEdit is the best algorithm since each node is identified by an unique identifier, using a hash table to link identifiers and node, they obtain a result with a complexity around $O(1+n/k)$ in the average case. Such a "trick" is only possible since FCEdit uses a unique identifiers.

The global performance behaviors of Logoot and WOOTH are quite similar, even if they are very different algorithms. This proves that the layer structure cost in performance, but this remains stable and does not exceed $50\,ms$.

b) Remote operations: In Figure 16 we present an execution time behaviours of algorithms using a skip policy for the remote operations on logarithmic scale.

To simulate a real experiment, the garbage collection mechanism of SOCT2 is disabled. Indeed, when users may disconnect, a garbage collection mechanism of SOCT2 cannot purge the history. The performances of OTTree and TreeOPT degrade over time since SOCT2 algorithm can not purge the history. Thus, the whole of operations received are stored in the history and it takes time to separate concurrent operations and transforms them that makes the algorithm the least efficient.

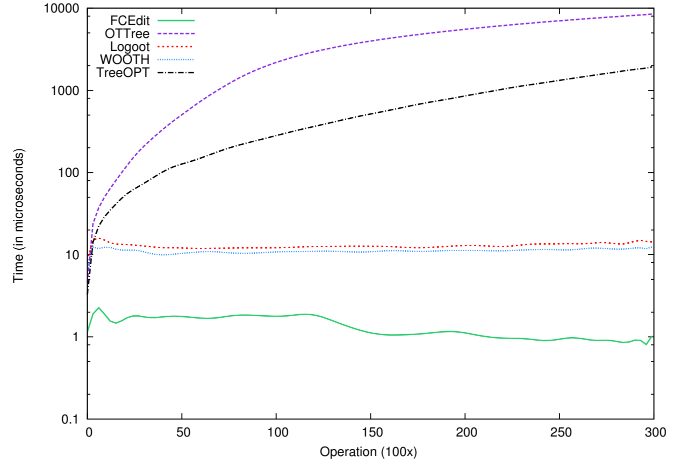


Fig. 16. Execution time for algorithms with Skip -remote-

Indeed, even if some garbage collection mechanisms exist, we consider that they can not be used in a general context where the number of replicas is unknown and fluctuating. As locally, the behaviors of Logoot and WOOTH algorithms remains stable, although these algorithms are based on layer structure, they outperform OTTree and TreeOPT with $10\mu s$ compare to $10\,ms$. The performance of FCEdit remains good and stable during all experiments, with just $3\mu s$ it represents the best algorithm in our experiment.

2) Compare policies: In what follows, we will present the behaviors of Logoot algorithm with different policies that exist and also WOOTH with reappear policy. For ordered tree based on WOOTH algorithm, a root and compact policies are not permitted. Because, we cannot merge different nodes that depends by their previous and next element with another located in different origin.

a) Local operations: In Figure 17 the global performance behaviors are the same excepted for root policy. In both policies, the algorithm must move all subtree deleted. In case of root policy, it moves under the root while for compact policy it moves under the last father on the tree. In the case where the node located in the origin path has a same label as the node in the new path, the two nodes are merged. Since, number of nodes located under the root in root policy are greater than the number of children under a node in compact policy, the time lost to find the nodes with the same label in root policy takes more time than for compact policy. Indeed, all nodes deleted in the tree are located under the root whereas in compact policy, a node contains his children and the nodes removed from their child.

b) Remote operations: The behavior of the different algorithm for remote operation presented in 18 is a slightly different compared to figure 17 since the behaviors are more chaotic for the root policy.

The behavior of Logoot with skip policy is the most stable. The average time of execution remains around $10\mu s$. As previously, the root behavior is the least efficient and the most chaotic. It improves when a replica deletes a path from the tree, as in operation number 6000 or 23000. In both algorithms Logoot and WOOTH with Reappear policy and also Logoot

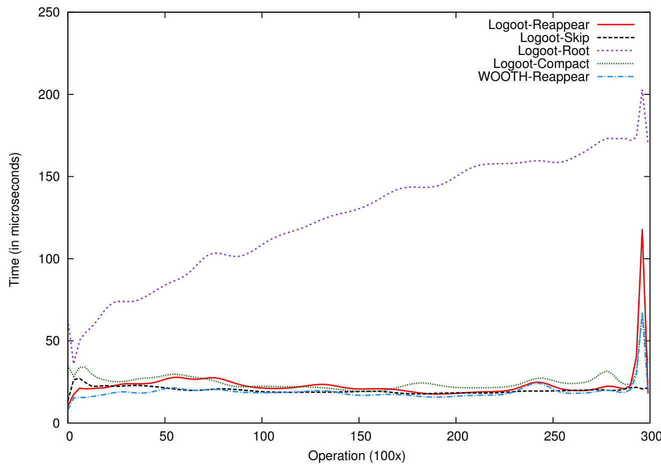


Fig. 17. Execution time for algorithms with policies -local-

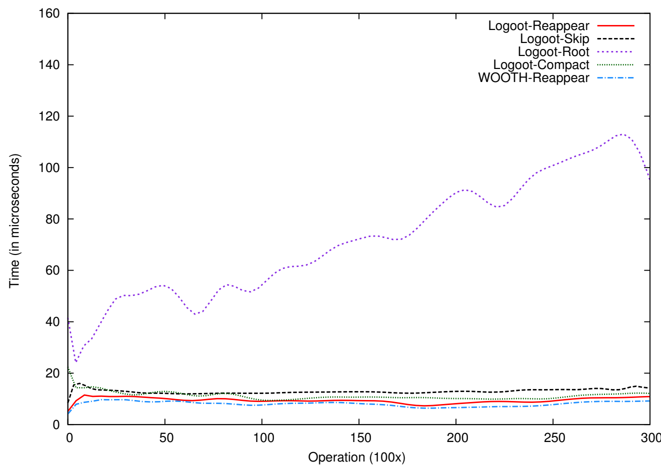


Fig. 18. Execution time for algorithms with policies -remote-

with compact policy have a chaotic behaviors although it remains stable globally.

Finally, although some algorithms are less efficient than other, the execution time never exceeds $1ms$ (far below 50ms). And almost every algorithm has a very stable behaviour below $30\mu s$. The Algorithms based on layer structure are acceptable and suitable for real-time collaboration. Moreover, they outperform some representative operational transformation as OTTree.

D. Memory occupation

Size of memory occupied by each studied algorithm may increase over time due to history, tombstones or growing identifiers. We present in the following, the algorithms behavior regarding memory usage in case of skip policy on logarithmic scale illustrated in figure 19.

A tree based on WOOTH algorithm occupies more memory compared to other tree algorithms, since in WOOTH an identifier is never deleted but just stored in tombstone and marked as invisible to users. OTTree and tree based on Logoot algorithm have almost the same behavior. The memory size occupied by Logoot depends of the size of identifiers Logoot,

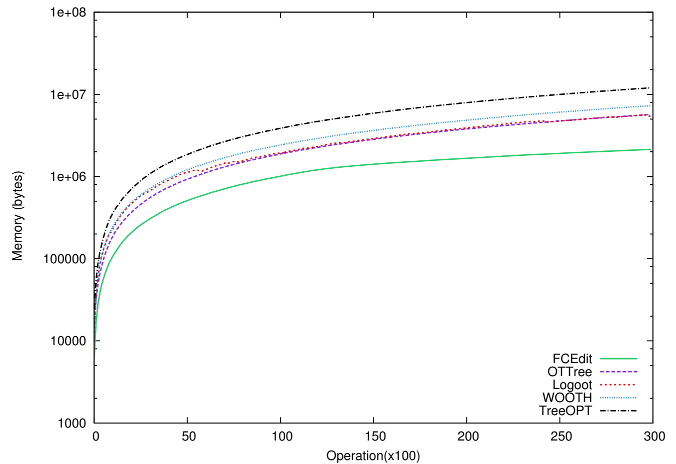


Fig. 19. Memory occupation for algorithms with Skip

whereas OTTree depends of number operation generated. Indeed, SOCT2 used in OTTree stores all operations in history, in addition, the garbage collector was quenched, moreover a deleted node is never removed. TreeOPT consumes more memory than OTTree because each node has a SOCT2 instance with a log. FCEdit remains the best algorithm regarding the memory space requirement since the identifiers are less cost than Logoot and the nodes removed are really deleted contrary to WOOTH and OTTree.

VI. Related work

Some collaborative system, such as version control system (Git, SVN, etc.), or distributed file systems [5] relies on human merging phases for some conflict cases, while some conflicts are resolved automatically. For instance, SVN creates a ”tree conflict” when a file is created in a concurrently deleted directory. On the other hand, Git behavior is similar to “reappear policy” (see Section IV-B) since it recreates silently the directory. However, human conflict resolving does not scale to massive collaboration use cases, and complex data types conflicts may be difficult to represent and resolve. For instance, Git is unable to merge correctly XML files. Our approach computes automatically a best effort merge, and can be combined to awareness mechanisms [1] to allow users to be conscious of concurrent modifications.

There exists many systems which satisfy the *eventual consistency* properties. Industrial systems, such as No-SQL data-stores (Amazon S3, CouchDB, Cassandra, etc.), relies on eventual consistency, but only manage key-value data types. Bayou [23] and Icecube [9] systems use constraints resolution mechanisms to resolve the conflicts. So, they can ensure generic data types constraints. But, these approaches do not scale well since they require a central or primary server and, as in version control systems, the system is not stable as soon as the update are delivered, since their merge procedures produce new operations.

Replicated data types are well-known in the literacy. For instance, there exists sets [18], sequences [13], [25], trees [12], file systems [5], etc. In Operational Transformation (OT) [2], replicas transform received operations against concurrent ones.

The OT approach has been successfully applied on several general public collaborative editing software, including Google Docs. Conflict-free Replicated Data Types (CRDT) [18] aims to design replicated data-types that integrate remote modifications without transformation. The goal of our approach is encapsulate any eventually consistent approach (OT or CRDT) in a replication layer and to design adaptation layer provide to satisfy non-trivial constraints. For instance, in our implementation (see Section V), we have implemented and tested trees layers on top of both different CRDT sets and OT sets.

VII. Conclusion

In this paper, we have presented a layered approach to design eventually consistent data types. Our approach composes one or several existing replicated data types which ensure eventual consistency, and adaptation layers to obtain a new eventually consistent data type. Each layer or replicated data type can be freely substituted by one providing the same interface.

We have demonstrated that our approach is implementable and obtains acceptable performances, even if these performance are sometimes slightly worse than some specific algorithms. Our experiments and implementation are public available and re-playable. Compared to existing solutions, the composition design can fit precisely the distributed application engineer wishes in terms of behavior and scalability.

In the future works, we will run experiments on a real data like git software histories and we will formally establish the equivalence proof between incremental and non-incremental algorithms.

Acknowledgement

This work is partially supported by the ANR national research grants STREAMS (ANR-10-SEGI-010) and ConcoRDanT (ANR-10-BLAN 0208).

References

[1] P. Dourish and V. Bellotti. Awareness and coordination in shared workspaces. In *Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, CSCW '92, pages 107–114, New York, NY, USA, 1992. ACM.

[2] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In J. Clifford, B. G. Lindsay, and D. Maier, editors, *SIGMOD Conference*, pages 399–407. ACM Press, 1989.

[3] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33:51–59, June 2002.

[4] N. Gu, J. Xu, X. Wu, J. Yang, and W. Ye. Ontology based semantic conflicts resolution in collaborative editing of design documents. *Advanced Engineering Informatics*, 19(2):103 – 111, 2005.

[5] R. G. Guy, J. S. Heidemann, and T. W. Page, Jr. The ficus replicated file system. *SIGOPS Oper. Syst. Rev.*, 26(2):26–, April 1992.

[6] C.-L. Ignat and M. C. Norrie. Customizable collaborative editor relying on treeopt algorithm. In *Proceedings of the eighth conference on European Conference on Computer Supported Cooperative Work*, ECSCW'03, pages 315–334, Norwell, MA, USA, 2003. Kluwer Academic Publishers.

[7] A. Imine, P. Molli, G. Oster, and M. Rusinowitch. Proving correctness of transformation functions in real-time groupware. In *Proceedings of the eighth conference on European Conference on Computer Supported Cooperative Work*, ECSCW'03, pages 277–293, Norwell, MA, USA, 2003. Kluwer Academic Publishers.

[8] 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.

[9] A.-M. Kermarrec, A. I. T. Rowstron, M. Shapiro, and P. Druschel. The IceCube approach to the reconciliation of divergent replicas. In *Proceedings of the twentieth annual ACM symposium on Principles of distributed computing - PODC'01*, pages 210–218. ACM Press, 2001.

[10] S. Martin and D. Lugiez. Collaborative peer to peer edition: Avoiding conflicts is better than solving conflicts. In H. Weghorn and P. T. Isaías, editors, *IADIS AC (2)*, pages 124–128. IADIS Press, 2009.

[11] S. Martin and D. Lugiez. Fixing collaborative edition on typed documents. In *CDVE*, 2010.

[12] S. Martin, P. Urso, and S. Weiss. Scalable xml collaborative editing with undo. In R. Meersman, T. Dillon, and P. Herrero, editors, *On the Move to Meaningful Internet Systems: OTM 2010*, volume 6426 of *Lecture Notes in Computer Science*, pages 507–514. Springer, 2010.

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

[14] N. M. Preguiça, J. M. Marquès, M. Shapiro, and M. Letia. A commutative replicated data type for cooperative editing. In *ICDCS*, pages 395–403. IEEE Computer Society, 2009.

[15] M. Ressel, D. Nitsche-Ruhland, and R. Gunzenhäuser. An Integrating, Transformation-Oriented Approach to Concurrency Control and Undo in Group Editors. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work - CSCW '96*, pages 288–297, Boston, MA, USA, November 1996. ACM Press.

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

[17] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. A comprehensive study of Convergent and Commutative Replicated Data Types. Research Report RR-7506, INRIA, January 2011.

[18] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski. Conflict-free replicated data types. In X. Défago, F. Petit, and V. Villain, editors, *Stabilization, Safety, and Security of Distributed Systems (SSS)*, volume 6976, pages 386–400, Grenoble, France, October 2011.

[19] S. Staworko and J. Chomicki. Validity-sensitive querying of XML databases. In *EDBT Workshops (dataX)*, pages 164–177. Springer LNCS 4254, 2006.

[20] M. Suleiman, M. Cart, and J. Ferrié. Serialization of Concurrent Operations in a Distributed Collaborative Environment. In *Proceedings of the ACM SIGGROUP Conference on Supporting Group Work - GROUP '97*, pages 435–445, Phoenix, AZ, USA, November 1997. ACM Press.

[21] 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*, pages 59–68, Seattle, WA, USA, November 1998. ACM Press.

[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.

[23] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the fifteenth ACM symposium on Operating systems principles - SOSP'95*, pages 172–182. ACM Press, 1995.

[24] S. Weiss, P. Urso, and P. Molli. Wooki: a P2P Wiki-based Collaborative Writing Tool. In *Web Information Systems Engineering*, 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, Québec, Canada, jun. 2009. IEEE Computer Society.

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

[27] S. Wu, U. Manber, and E. Myers. A subquadratic algorithm for approximate regular expression matching. *Journal of Algorithms*, 19(3):346 – 360, 1995.