Scalable XML Collaborative Editing with Undo

Stéphane Martin, Pascal Urso, Stéphane Weiss

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Scalable XML Collaborative Editing with Undo

Stéphane Martin — Pascal Urso — Stéphane Weiss

N° 7362

August 2010

Thème COG

Rapport
de recherche

ISRN INRIA/RR--7362--FR+ENG

SN 0249-6399

Scalable XML Collaborative Editing with Undo

Stéphane Martin * , Pascal Urso** , Stéphane Weiss**

Thème COG — Systèmes cognitifs
Équipes-Projets SCORE

Rapport de recherche n° 7362 — August 2010 — 20 pages

Abstract: Commutative Replicated Data-Type (CRDT) is a new class of algorithms that ensure scalable consistency of replicated data. It has been successfully applied to collaborative editing of texts without complex concurrency control.

In this paper, we present a CRDT to edit XML data. Compared to existing approaches for XML collaborative editing, our approach is more scalable and handles all the XML editing aspects : elements, contents, attributes and undo. Indeed, undo is recognized as an important feature for collaborative editing that allows to overcome system complexity through error recovery or collaborative conflict resolution.

Key-words: XML, Collaborative Editing, P2P, Group Undo, Scalability, Optimistic Replication, CRDT

* stephane.martin@lif.univ-mrs.fr, Laboratoire d'Informatique Fondamentale, MoVe
** {pascal.urso,stephane.weiss}@loria.fr, Université de Lorraine - LORIA - SCORE

Centre de recherche INRIA Nancy – Grand Est
LORIA, Technopôle de Nancy-Brabois, Campus scientifique,
615, rue du Jardin Botanique, BP 101, 54602 Villers-Lès-Nancy (France)
Téléphone : +33 3 83 59 30 00 — Télécopie : +33 3 83 27 83 19

Edition Collaborative passant à l'échelle pour les documents XML avec Annulation

Résumé : Le type de données répliqué commutatives (CRDT) est une nouvelle classe d'algorithmes qui assurent la cohérence des données répliquées tout en passant à l'échelle. Il a été appliqué avec succès à l'édition collaborative de textes sans mécanisme de contrôle de la concurrence complexe.

Dans cet article, nous présentons un CRDT pour éditer des données XML. Par rapport aux approches existantes pour l'édition collaborative d'XML, notre approche offre un meilleur passage à l'échelle et gère tous les aspects de l'édition de document XML: éléments, le contenu, les attributs et l'annulation. En effet, l'annulation est reconnue comme un élément important pour l'édition collaborative qui permet de surmonter la complexité du système de collaboration grâce à la récupération d'erreur ou de résolution des conflits.

Mots-clés : XML, Edition Collaborative, P2P, Annulation de groupe, Passage à l'échell, Réplication Optimiste, CRDT

Scalable XML Collaborative Editing with Undo

Stéphane Martin[1], Pascal Urso[2], and Stéphane Weiss[2]

[1] stephane.martin@lif.univ-mrs.fr
Laboratoire d'Informatique Fondamentale
39 rue F. Jolio-Curie,
13013 Marseille, France
[2] (pascal.urso,stephane.weiss)@loria.fr
Université de Lorraine
LORIA, Campus Scientifique,
54506 Vandoeuvre-lès-Nancy, France

Commutative Replicated Data-Type (CRDT) is a new class of algorithms that ensure scalable consistency of replicated data. It has been successfully applied to collaborative editing of texts without complex concurrency control.

In this paper, we present a CRDT to edit XML data. Compared to existing approaches for XML collaborative editing, our approach is more scalable and handles all the XML editing aspects : elements, contents, attributes and undo. Indeed, undo is recognized as an important feature for collaborative editing that allows to overcome system complexity through error recovery or collaborative conflict resolution.

Keywords: XML, Collaborative Editing, P2P, Group Undo, Scalability, Optimistic Replication, CRDT.

1 Introduction

In large-scale infrastructures such as clouds or peer-to-peer networks, data are replicated to ensure availability, efficiency and fault-tolerance. Since data are the heart of the information systems, the consistency of the replicas is a key feature. Mechanisms to ensure strong consistency levels – such as linear or atomic – do not scale, thus modern large-scale infrastructures now rely on eventual consistency [29].

Commutative Replicated Data Types [21] (CRDT) is a promising new class of algorithms used to build operation-based optimistic replication [24] mechanisms. It ensures eventual consistency of replicated data without complex concurrency control. It has been successfully applied to scalable collaborative editing of textual document [31,19] but not yet on semi-structured data type.

EXtensible Markup Language (XML) is used in a wide range of information systems from semi-structured data storing to querying. Moreover, XML is the standard format for exchanging data, allowing interoperability and openness.

XML data editing is mainly done through domain specific applications or general purpose editors. Collaborative editing (CE) provides several advantages such as obtaining different viewpoints or reducing task completion time to obtain a more accurate

⋆ stephane.martin@lif.univ-mrs.fr, Laboratoire d'Informatique Fondamentale, MoVe
⋆⋆ {pascal.urso,stephane.weiss}@loria.fr, Université de Lorraine - LORIA - SCORE

final result. Undo has been recognized as an important feature of single and collaborative editors [2,6]. The undo feature provides a powerful way to recover from errors, edit conflicts and vandalism acts. So, it helps the user to face the complexity of the system [16].

Nowadays, collaborative editing becomes massive and part of our every day life. The online encyclopedia Wikipedia users have produced 15 millions of articles in a few years. Another example is Google Wave, the new Google service based on XML documents that mixes real-time collaborative editing and communication. Despite its success lower than excepted [3], it already has one million users. These examples stress the scalability requirement that will eventually face an XML collaborative editing system that should be deployed on clouds or peer-to-peer networks.

In the research field of collaborative editing, some approaches are generic enough to deal with XML documents [27,11,23]. However they suffers for their lack of scalability that makes them unsuitable for clouds or peer-to-peer networks. There exists other approaches that are specifically designed for peer-to-peer XML collaborative editing. [8] uses tombstones that makes the document growing without limits; while [17] does not treat XML attributes. Moreover, none of these specific approaches propose an undo feature.

We propose to design an XML CRDT. This CRDT handles both aspects of XML trees : elements' children and attributes. The order in the list of the elements' children are treated as in linear structure CRDT. Elements' attributes are treated using a last-writer-wins rule.

Designing an undo feature is a non-trivial task. First, in collaborative editing this feature must allow to undo any operation – and not only the last one – from any user [2,3]. This is called global selective undo (or anyundo). Second, this undo must be correct from the user point of view. The system must return in a state such as the undone operation was never been performed [25]. Our undo is obtained by keeping the previous value given to attributes and delete operations on elements, and then counting concurrent undo and redo operations. A garbage collection mechanism is presented to garbage old undo values.

This paper is structured as follows. Section 2 presents a brief overview of comparable approaches. Section 3 introduces the notion of a distributed XML collaborative editor. Section 4 describes an XML CRDT without undo. Section 5 describes an XML CRDT with undo. Section 6 formally establishes the correctness of our approach according to eventual consistency. Section 7 discusses about the theoretical scalability of the approach and describes a garbage collection mechanism. And finally, Section 8 briefly concludes the paper.

2 State of the art

There exists several approaches that can be used for ensuring eventual consistency of XML data. Most of them issue from the field of collaborative editing.

[3] Maybe due to some missing features including undo

The Operational Transformation (OT) [23] approach is an operation-based replication mechanism. OT relies on a generic integration algorithm and a set of transformation functions specific to the type of replicated data. Some integration mechanism use states vectors – or context vectors [27] in presence of undo – to detect concurrency between operations; such mechanisms are not adapted to large-scale infrastructures. Ignat et al. [8] couples an integration mechanism [5] that uses anti-entropy, with some specific transformation functions [20] to obtain P2P XML collaboration. However, this proposition replaces deleted elements by tombstones in the edited document to ensure consistency, making the document eventually growing without limits and proposes no undo.

Other generic approaches can be adapted to edit XML document. Some reconciliation mechanisms [11,28] allow to define the specific constraints that must satisfy the editing and undo operations [18]. However, the complexity of the reconciliation mechanism is exponential in term of number of editing operations or replicas.

Martin et al. [17] proposes an XML-tree reconciliation mechanism very similar to a CRDT since concurrent operations commute without transformation. However, this approach does not treat XML element's attributes which require a specific treatment since they are unique and unordered. Also, it uses state vector that limits its scalability and proposes no undo feature.

In the field of Data Management, some works give attention to XML replication. Some of them [12,1] suppose the existence of some protocol to ensure consistency of replicated content without defining it. Finally, [15] proposes a merging algorithm for concurrent modifications that can only be used in a centralized context.

In the field of distributed systems, several well-known methods exist to obtain replica consistency. Consensus [4] or quorum [9] algorithms can be used to obtain transaction atomicity on data updates. Their limited scalability makes them less suitable for large-scale applications where only eventual consistency is required. Lastly, the Thomas-Write-Rule (aka Last-Writer-Wins) [10], allows to obtain an agreement on a value, but is much more scalable. We uses a variation of it for updating attribute values.

3 XML Collaborative Editing

An XML document is an ordered tree of elements. Element's content, including text, forms the ordered list of children of the element. Elements have a map that associates name to value. This map represents the unordered attributes of the element.

```
<article xmlns="http://docbook.org/ns/docbook">
  <title>Extensible Markup Language</title>
  <para>
    <acronym>XML</acronym>
  </para>
</article>
```

More formally XML documents are defined by the grammar :

$$
\begin{aligned}
T &::= < tag\ Attributes > T' < /tag > T \\
&\ \mid\ \varepsilon \\
T' &::= text \\
&\ \mid\ T \\
Attributes &::= attribute = \text{"}value\text{"}\ Attributes \\
&\ \mid\ \varepsilon
\end{aligned}
$$

Where $attribute, value, text$ and tag are non-null strings. All $attribute$ in same element are unique. Even if they appear in an XML file in a certain order, attributes are unordered, i.e., their appearance order has no meaning.

In a collaborative editor, to ensure scalability and high-responsiveness of local modifications, data must be replicated [7]. This replication is optimistic [24] since local modifications are immediately executed. The replicas are allowed to diverge in the short time, but the system must ensure eventual consistency. When the system is idle (i.e., all modifications are received), the replicas must have the same content.

Thus, we see an XML collaborative editor as a set of network nodes that host a set of replicas (up to one per node) of the shared XML document. Local modifications are immediately executed and disseminated to all other replicas. We assume that every replica will eventually receive every modification.

A Commutative Replicated Data Type [21] is a data type where all operations commute. I.e., whatever the delivery order of operations, the resulting document is identical. The basic operations that affect an XML tree are :

- $Add(e_p, e)$: Adds a new edge e under the edge e_p
- $Del(e)$: Deletes the edge e
- $SetAttr(e, attr, val)$: Sets the value val to the attribute $attr$ of the edge e. The deletion of an attribute is done by setting is value to nil.

4 XML CRDT

In this section, we define an XML CRDT without undo. We define the set of operations that modify the XML tree and their effect. This CRDT is a generalized version of [17] extended with attributes management.

4.1 Add and delete edges

To allow *Add* and *Del* operations to commute, we use a unique timestamp identifier. Timestamp identifiers are defined as follows: each replica is identified by a unique *SiteNb* and each operation generated by this site is identified by a numbering *NbOp*. An identifier *id* is a pair $(NbOp : SiteNb)$. For instance $(3:2)$ identifies the operation 3 of the site number 2. The set of the identifiers is denoted by *ID*. Thus, two edges added concurrently at the same place in the tree have different identifiers.

Add and delete operations becomes :

- $Add(id_p, id)$: Adds a edge with identifier id under the edge id_p. This edge is empty, it has no tag-name, child or attribute.
- $Del(id)$: Deletes the edge identified by id.

The tag-name and the position of an edge regarding to sibling edges, are treated as attributes. Thus they can be modified without deleting and creating a new edge.

The position of an edge is not a standard number. Indeed, to ensure that the order among edges is the same on all replicas, this position must be *unique, totally ordered and dense*. Positions are dense if a replica can always generate a position between two arbitrary positions. This position can be a priority string concatenated with an identifier [17], a sequence of integers [31], or a bitstring concatenated with an identifier [21] all with a lexicographic ordering.

4.2 Update attributes

To allow *SetAttr* operations to commute, we use a classical last-writer-wins technique. We associate to each attribute a timestamp ts. A remote *SetAttr* is applied if and only if its timestamp is higher than the timestamp associated to the attribute. This timestamp is formed by a clock h (logical clock or wall clock) and a replica number *SiteNb*. Timestamps are totally ordered. Let $ts_1 = (h_1 : s_1)$ and $ts_2 = (h_2 : s_2)$, we have $ts_1 > ts_2$ if and only if $h_1 > h_2$, or $h_1 = h_2$ and $s_1 > s_2$. Timestamps are loosely synchronized, i.e., when a replica receives an operation with a timestamp $(h_2 : s_2)$, it sets its own clock h_1 to $max(h_1, h_2)$.

The *SetAttr* operation becomes :

- $SetAttr(id, attr, val, ts)$: Sets the value val with the timestamp ts to the attribute $attr$ of the edge identified by id. The deletion of an attribute is done by setting is value to nil.

The special attributes $@tag$ and $@position$ that contains the tag-name and the position of an edge cannot be nil. Without loss of generality, the add operation can be $Add(id_p, id, tag, pos)$ that adds the edge and sets the tag-name and position. To modelize the textual edges we use another special attribute $@text$. If this attribute has a value v, whatever the value of other attributes, the edge is considered as a textual edge with content v.

4.3 Algorithms

We consider an XML tree as an *edge e* with three elements :

- $e.identifier$: the unique identifier of the edge (a timestamp)
- $e.children$: the children of the edge (a set of edge)
- $e.attributes$: the attributes of the edge (a map string to value). The key of the map are the attribute's name (a string), and a value *value* whose has two elements
  - *av.value* : the current value of the attribute (a string)
  - *av.timestamp* : the current timestamp of the attribute.

The function deliver(op, t) applies an operation op on an XML tree t. The function find(t, id) returns the edge identified by id. The function findFather(t, id) returns the father of the edge identified by id.

```
deliver (Add(id_p, id), t) :
  edge p = find(id_p, t), e = new edge(id);
  if p ≠ nil then p.children = p.children ∪ {e};
end

deliver (Del(id), t) :
  edge p = findFather(id, t), e = find(id, p);
  if p ≠ nil then p.children = p.children \ {e};
end

deliver (SetAttr(id, attr, val, ts), t) :
  edge e = find(id, t);
  if e ≠ nil and (e.attribute[attr] = nil or e.attribute[attr].timestamp < ts) then
      e.attribute[attr].value = val;
      e.attribute[attr].timestamp = ts;
  endif
end
```

Example 1. This example is the begin of XML example of Section 3. We start with edge "article". First, each user adds an edge using the $Add(id_p, id)$ operation, where id_p is the identifier of the edge "article". Second, each user renames concurrently the tag of one of the previously inserted edge. Using the $SetAttr(id, @tag, v, ts)$ operation. User1 renames the tag to "title" while User2 changes the tag of the same edge to "para". The operation with the higher timestamp sets the tag of the element[4]. The result of these concurrent operation is presented Figure 1.

4.4 Semantic dependency

The semantic dependency is a relation which relies the operation with those necessary to its executions.

- $Add(id_p, id) \succ_s Del(id)$: an edge can be deleted only if it has been created.
- $Add(id', id_p) \succ_s Add(id_p, id)$: adding edge id under edge id_p requires that edge id_p has been created.
- $Add(id_p, id) \succ_s SetAttr(id, Attr, Value, ts, id_{op})$: Creating or modifying edge attribute requires edge id has been created.

To respect these semantic dependencies, we can use a scalable causal broadcast [13]. Moreover, causality preservation is often cited as a user requirement for collaborative editing [26].

[4] For simplicity reason, positions are omitted.

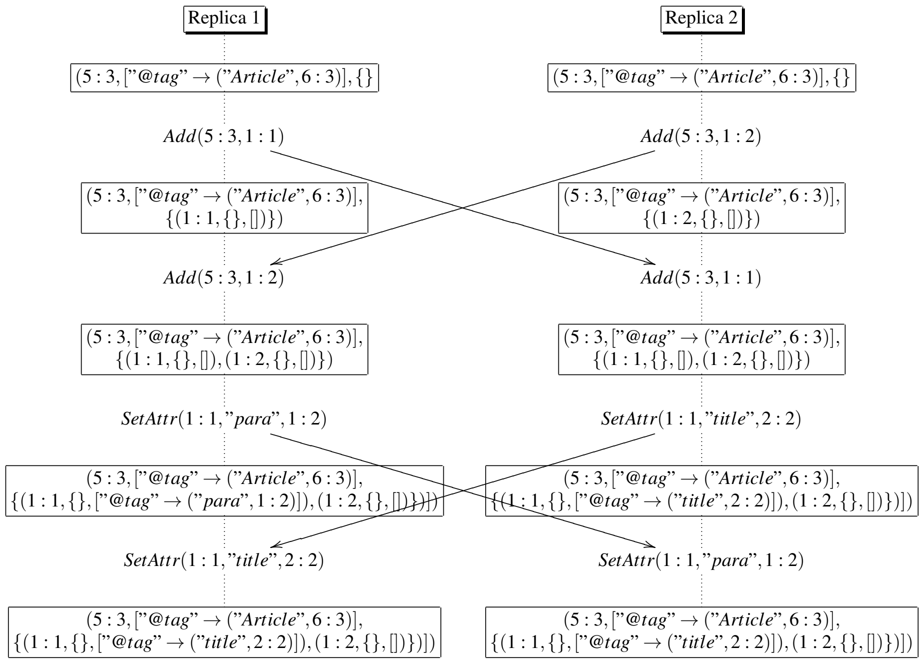

Fig. 1. Concurrent attribute update

5 XML CRDT with undo

Obtaining a correct undo from the user's point of view is a non-trivial task [22,6,30]. In this section, we informally describe how to deal with undo operations, and then we formally describe our mechanism. This mechanism allows to undo and redo any operation that affects the XML tree.

5.1 Undoing add and delete

The operation that undoes an *Add* is not strictly a *Del*. Let's have the following scenario (see Figure 2).

1. A user adds an element
2. A user deletes this element
3. The add is undone
4. The delete is undone concurrently by 2 different users.

Using *Del* to undo *Add* leads to different result according to the reception order of the operations. The element is visible if an un-delete is received in last or not if it is a un-add. This behavior violates eventual consistency.

So, we must keep the information about deleted elements as tombstones. Moreover, simply counting the number of "appearing" operations (add and undelete) minus the number of "disappearing" operations (delete and unadd) is not sufficient. In the above

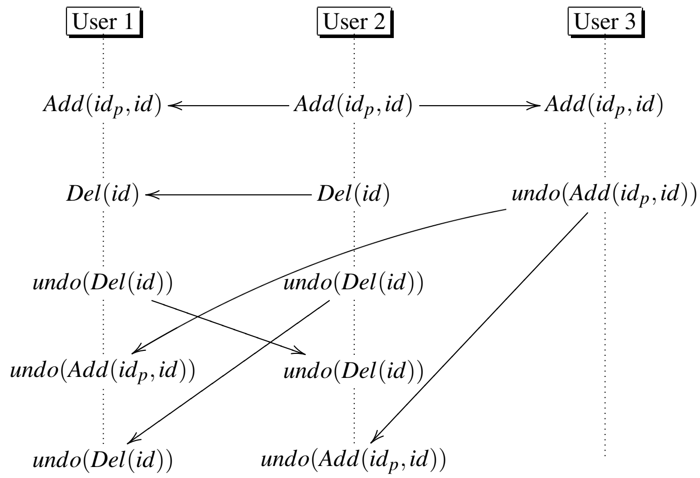

Fig. 2. Concurrent undos.

example, we have 3 appearing operations minus 2 disappearing ones, thus, the element may be visible. However, there are two operations (one add and one delete) and both of them are undone. According to undo definition, none of their effect must be performed and the element must not be visible.

To obtain a satisfying undo, we keep the information about the add and every delete operations associated to each edge. Then we count the *effect counter* of an operation : one minus the number of undo plus the number of redo. If this effect counter is greater than 0, the operation has an effect. Finally, an element is visible if the add has an effect counter greater than 0, and no delete with an effect counter greater than 0[5].

5.2 Undoing attribute updates

Similarly to undo of add and delete operation, we need to keep the operations affecting an attribute (i.e., previous update values). We have for each value an effect counter. The value of an attribute is determined by the more recent value with an effect counter greater than 0. Thus we need to keep into the map of attributes, the list of values – including nil value – associated to an effect counter. The list is ordered by the decreasing timestamp.

[5] One may argue that all the delete operations are identical, and thus keep only one effect counter for all of them. On the other hand, one can also argue that undo operations are normal operations and require their own counter in order to obtain a real undo of undo whose is sightly different than redo. All these alternatives are possible in our framework. For shake of efficiency and clarity we present the above one that has a limited overhead with a good respect of user's intentions.

5.3 Algorithms

With undo, the attributes of an edge becomes an ordered list of *value*, each value contains 3 elements :

- $v.value$: a value of the attribute (a string)
- $v.timestamp$: the timestamp associated to this value
- $v.effect$: the effect counter of this value (a integer)

The list is ordered by the timestamp. The function add(l, v) adds a value v in the list l at its place according to $v.timestamp$. The function get(l, ts) returns the value associated to ts in the list l. The special $@add$ attribute has only one value associated to the timestamp equal to the edge identifier. The special $@del$ attribute store the list of timestamp of delete operation that affect the edge.

Thus, the original edit operation delivery becomes :

deliver $(Add(id_p, id), t)$:
  edge p = find(t, id_p), e = **new** edge(id);
  $p.children = p.children \cup \{e\}$
  add($e.attributes[@add]$, **new** value ($nil, id, 1$));
end

deliver $(Del(id, ts), t)$:
  edge e = find(t, id);
  add($e.attributes[@del]$, **new** value ($nil, ts, 1$));
end

deliver $(SetAttr(id, attr, val, ts), t)$:
  edge e = find(t, id);
  add($e.attributes[attr]$, **new** value ($val, ts, 1$));
end

Undo of an operation is simply achieved by decrementing the corresponding effect counter. When a *Redo* is delivered, the increment function is called with a delta of $+1$.

deliver $(Undo(Add(id_p, id)), t)$:
  increment($t, id, @add, id, -1$);
end

deliver $(Undo(Del(id, ts)), t)$:
  increment($t, id, @del, ts, -1$);
end

deliver $(Undo(SetAttr(id, attr, val, ts)), t)$:
  increment($t, id, attr, ts, -1$);
end

function increment($t, id, attr, ts, delta$)

```
  edge e = find(t, id);
  value v = get(e.attributes[attr], ts);
  v.effect += delta;
end
```

Figure 3 presents the application of our function on the introducing scenario with concurrent undo and redo. At the end, on every replica, the add and del operations have an effect counter lesser or equal to 0, thus the node is invisible.

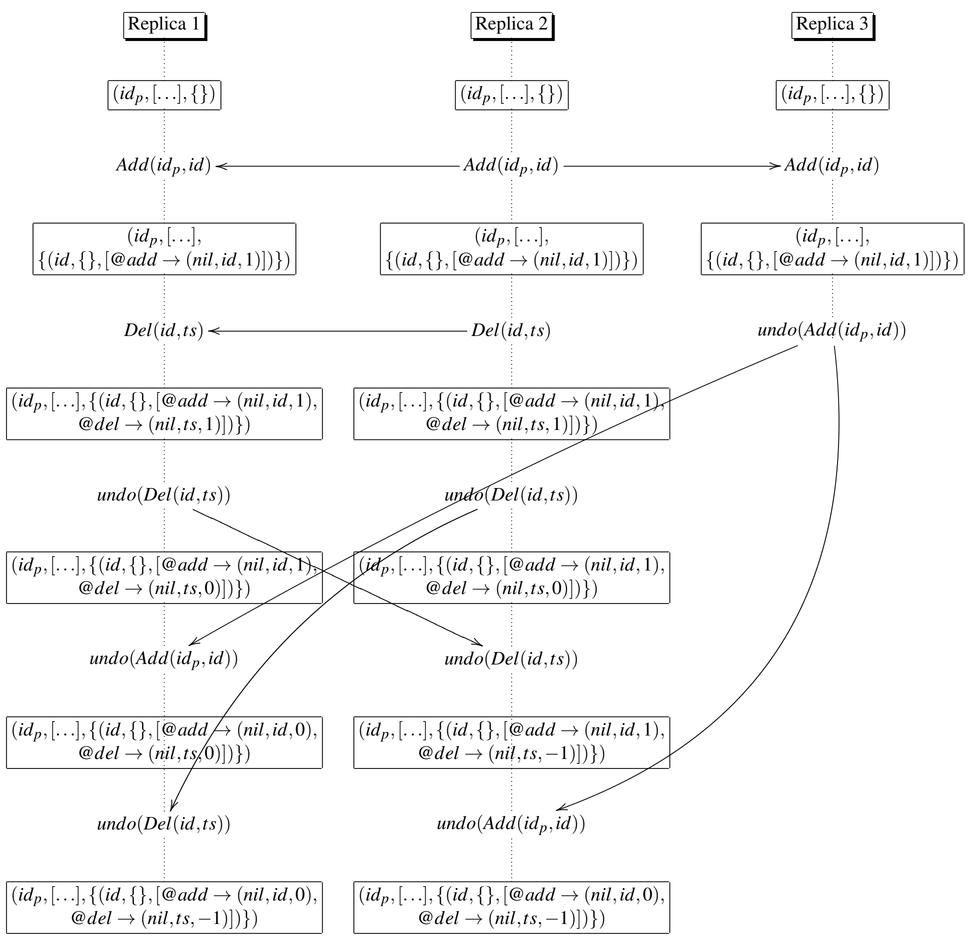

Fig. 3. Concurrent undos with effect counters.

5.4 Model to XML

As the model described above includes tombstones and operation information, it cannot be used directly by applications. Indeed, applications must not see tombstones. First, we need to define which element belongs to the view, i.e., which element is visible. A node is visible if the effect counter of the attribute "@add" is at least one, and if all values of the attribute "@del" have an effect counter of at most 0. As a result, we must know if the attribute "@del" has an active value. For that purpose, we define the *visible(e)* function that returns true if the edge *e* is visible.

```
function visible (e):
  if get(e.attribute[@add], e.identifier).effect < 1 then return false;
  for d in e.attribute[@del] do
    if d.effect > 0 then return false;
  done
  return true;
end
```

We can now determine whether a node is visible or not. However, each attribute still contains several "values", indeed, the value list associated to each attribute contains old and undone values. Therefore, we need to compute the current value of the attributes. The function *getValue(vlist)* is designed to find the newest non-undone value of a value list. Since elements the list *vlist* is ordered according to their timestamp, we assume that the loop "for" walks the list from the newest to the oldest. Therefore, the first element with an effect counter greater than 1 is the current value of this element.

```
function getValue(vlist):
  for v in vlist do
    if v.effect ≥ 1 then return v.value;
  done
  return nil ;
end
```

Finally, we can write the function *model2XML(e)* that exports the model in an XML format. First, the function *model2XML(e)* checks whether the edge *e* is visible or not. If not, the function does nothing. Otherwise, we need to write this element in the XML Document. For that purpose, we assume the existence of a function *out(str)* that writes the XML document. If the edge is a text, i.e., the attribute $@text$ has a value, we write this value in the XML Document. If the edge is visible and is not a text, we write the tag and the attributes corresponding to that edge. We assume that *child2List(s)* is a function that returns a list of the edge of the set *s* sorted according to their attribute $@position$. Finally, the function *model2XML* calls itself to treat the children of the edge *e*.

```
function model2XML(e):
  if visible (e) then
    t = getValue(e.attributes[@text]);
    if t ≠ nil then out(t);
    else
```

```
      out("< " + e.tag);
      for (attr, vlist) in e.attributes do out(" "+attr+"="+getValue(vlist));
      out(" > ");
      list l = child2List(e.children);
      for n in l do model2XML(n);
      out("</"+e.tag+">");
    endif
  endif
end
```

5.5 Semantic dependency

- $Add(id', id_p) \succ'_s Add(id_p, id)$: adding edge *id* under edge id_p requires that edge id_p has been created.
- $Add(id_p, id) \succ'_s SetAttr(id, Attr, Value, ts)(n(t))$: Creating or modifying edge attribute requires edge *id* has been created.
- $Add(id_p, id) \succ'_s Undo/Redo(Add(id_p, id))$: Undoing or Redoing of edge creating require edge *id* has been created.
- $SetAttr(id, Attr, Value, ts, id_{op}) \succ'_s Undo/Redo(SetAttr(id, Attr, Value, ts, id_{op}))$ Undo or Redoing attribute setting, require creation of this *attribute*.
- by definition we have :
  - $Add(id_p, id) \succ'_s Del(id, id_{op})$
  - $Del(id, id_{op}) \succ'_s Redo/Undo(Del(id, id_{op}))$

With undo, semantic dependency can be achieve by a very simple mechanism. An $Del/SetAttr/Add$ operation affecting an edge can only be delivered if an edge is already present (visible or not). A $Undo/Redo$ operation can only be delivered if the edge and the corresponding timestamped value is present.

6 Correctness

In this section we show that our operations commutes, and thus, that our data type is a CRDT and eventual consistency is ensured[6].

Theorem 1. *Let $Op_1 = \{Add, Del, SetAttr\}$ without undo. The set $(Op_1, \succ_s)$ is an independent set of operations.*

We define $\succ_s^*$ by: $op_1 \succ_s op_2 \wedge op_2 \succ_s op_3 \Rightarrow op_1 \succ_s^* op_3$ and $\|_s^*$ by $op_1 \not\succ_s^* op_2 \wedge op_2 \not\succ_s^* op_1 \Leftrightarrow op_1 \|_s^* op_2$

We define a sequence of operation: $Do(op_n, Do(op_{n-1}, ...Do(op_1, t)...)) = [op_1, ..., op_n](t)$.

Proof. We prove that if $op_1 \|_s^* op_2$ then $[op_1, op_2](t) = [op_2, op_1](t)$ by a case analysis on all possible pairs op_1, op_2.

[6] The $Do(op, t)$ function used in the proof is the state of t after applying deliver(op, t).

1. $op_1 = Add(id_1, id_{p_1})$
   (a) $op_2 = Add(id_2, id_{p_2})$
   - if $id_{p_1} = id_{p_2}$ in same set we add n_1 followed by n_2 or vice versa. the only one set which be modified is identified by $id = id_{p_1} = id_{p_2}$ and $t_i'd = t_i d \cup \{n_1\} \cup \{n_2\}$
   - else the two effect are in two independent subtrees or $op_1 \not\Vert_s^* op_2$

   (b) $op_2 = Del(id_2)$
   - $id_2 = id_{p_1}$ or id_{p_1} is in subtree id_2:
     let t a tree. $t_1 = Do(Del(id_2), t)$ by definition id_p is deleted.
     $Do(Add(id_1, id_{p_1}), t) = t_1$. $t_2 = Do(Add(id_2, id_{p_1}), t)$ and $Do(Del(id_2), t_2) = t_1$ because a subtree is erased.
   - $id_2 = id_1$: because $Add(id_1, id_{p_1}) \succ_s Del(id_1)$.
   - other : the edge id_1 has been created and id_2 has been deleted whatever order.

   (c) $op_2 = SetAttr(id_2, attr_2, val_2, ts_2)$
   - $id_2 = id_1$: the edge be created before attribute settings because $Add(id_1, id_{p_1}) \succ_s SetAttr(id_1, attr_2, val_2, ts_2)$.
   - other, the add has no effect on SetAttr and vice versa. ◇

2. $op_1 = Del(id_1)$
   (a) $op_2 = Add(id_2, id_{p_2})$: It's 1b case.
   (b) $op_2 = Del(id_2)$ If id_1 is a subtree id_2 then $[Del(id_1), Del(id_2)](t)$ there are no edge to delete with $Del(id_1)$ because it was deleted with $Del(id_2)$. And $[Del(id_2), Del(id_1)](t)$ the the edge and subedge of id_1 were deleted at first time and id_2 with id_1 was deleted too. else two subtree are distinct .
   (c) $op_2 = setAttr(id_2, attr_2, value_2, ts_2)$
   - $id_1 = id_2$
     Let $t' = Do(Del(id_1), t)$. $Do(SetAttr(id_1, attr_2, value_2, ts_2)(t') = t'$ because id_1 is not present in t'.
     $Do(Del(id_1), Do(SetAttr(id_1, attr_2, value_2, ts_2), t)) = t'$ because id_1 and its subtree was deleted. Whatever its attribute.
   - Other : there are no problems.

   ◇

3. $op_1 = SetAttr(id_1, attr_1, value_1, ts_1)$
   (a) $op_2 = Add(id_2, id_{p_2})$: It's 1c case.
   (b) $op_2 = Del(id_2)$: It's 2c case.
   (c) $op_2 = SetAttr(id_2, attr_2, value_2, ts_2)$:
   - $id_1 \neq id_2$: The edge is different.
   - $attr_1 \neq attr_2$ by definition the list is same, because it is ordered by the lexicographic order.
   - $id_1 = id_2 \wedge attr_1 = attr_2$
     - $ts_1 < ts_2$ let $t_1 = op_1(op_2(t))^{(1)}$
       let $t_2 = op_2(op_1(t))^{(2)}$
       In $^{(1)}$ the attribute of id_1 is $value_2$ and not changed by op_1 (definition). in $^{(2)}$ the attribute of id_1 is $value_1$ and changed by op_2 to $value_2$ (definition). therefore $t_1 = t_2$.

- $ts_2 < ts_1$: idem with values number inverted.
- $ts_1 = ts_2$ By definition $ts_1 \neq ts_2$ $\diamond$

In undo case the $Del(id,ts)$ operation becomes equivalent to a $SetAttr(id,@del,nil,ts)$ operation.

Theorem 2. *Let $Op_2 = \{Add, SetAttr, Undo/Redo(SetAttr)\}$ with undo.The set $(Op_2, \succ'_s$ $)$ is an independent set of operations.*

Proof. We prove that if $op_1 \parallel_s^* op_2$ then $[op_1, op_2](t) = [op_2, op_1](t)$ by a case analysis on all possible pairs op_1, op_2.

1. $op_1 = Add(id_1, id_{p_1})$
   (a) $op_2 = Add(id_2, id_{p_2})$ the operation just add the special attribute $@add$, the previous proof is still valid.
   (b) $op_2 = SetAttr(id_2, attr_2, val_2, ts_2)$
      – $id_2 = id_1$: the edge is created before attribute settings because $Add(id_1, id_{p_1}) \succ_s SetAttr(id_1, attr_2, val_2, ts_2)$.
      – other, the add has no effect on SetAttr and vice versa. $\diamond$
   (c) $op_2 = Redo/Undo(SetAttr(id_2, attr_2, val_2, ts_2))$
      – if $id_1 = id_2$ by definition, $op_1 \not\parallel_s^* op_2$
      – else: the creation of edge is independent of another edge modification.

2. $op_1 = SetAttr(id_1, attr_1, value_1, ts_1)$
   (a) $op_2 = Add(id_2, id_{p_2})$: It's 1b case.
   (b) $op_2 = SetAttr(id_2, attr_2, value_2, ts_2)$:
      – $id_1 \neq id_2$: The edge is different.
      – $attr_1 \neq attr_2$ by definition the list is same, because it is ordered by the lexicographic order.
      – $id_1 = id_2 \wedge attr_1 = attr_2$
         - $ts_1 \neq ts_2$: by definition we add in list of values ordered by timestamp. The add in ordered list is independent of adding order. The two values are present.
         - $ts_1 = ts_2$ By definition $ts_1 \neq ts_2$ $\diamond$
   (c) $op_2 = undo/redo(SetAttr(id_2, attr_2, value_2, ts_2))$
      – if $id_1 \neq id_2$ or $attr_1 \neq attr_2$: by definition, $op_1 \not\parallel_s^* op_2$
      – else op_1 create a value item and op_2 increase or decrease effect on another value item. It is independent.

3. $op_1 = undo/redo(SetAttr(id_1, attr_1, value_1, ts_1))$
   (a) $op_2 = Add(id_p, id)$ is same of case 1c.
   (b) $op_2 = SetAttr(id_2, attr_2, value_2, ts_2)$ is same of case 2c
   (c) $op_2 = undo/redo(SetAttr(id_2, attr_2, value_2, ts_2))$
      – if $id_1 = id_2 \wedge attr_1 = attr2 \wedge ts_1 = ts_2$: by definition each operation increases or decreases the same effect field, it is commutative operation.
      – else each operation decreases or increases two different counters.

The other operations – Del, $Undo/Redo(Add)$, $Undo/Redo(Del)$ – can be defined using operations in op_2 set – $SetAttr$ and $Undo/Redo(SetAttr)$.

7 Scalability discussion

In this section we discuss about the scalability of the approach. The XML CRDT with and without undo scales in term of replicas number. The replicas number in not a factor in every elements of the CRDT. There is no consensus, central point or state vector embedded on messages.

The only requirement to ensure consistency of the XML CRDT without undo is to receive delete operation after insert of a node. With undo, this constraint is not required to ensure consistency since a delete can be received before an insert. The delete produces directly a tombstone.

7.1 Complexity

Here is the time complexity of the functions used in our approach.

- find, findFather : the worst time complexity is $O(n)$ with n the number of edge in the tree (including invisible ones in case of undo). Using path to an edge – as list of identifier – instead of identifier in operation, the average time complexity becomes $O(hc)$ with h the average height of the tree and c the average number of children per edge. If we use hash table that associate identifier to edge, the average complexity becomes $O(1)$ for find. If we store the father *id* in the edge, the average complexity using hash table becomes also $O(1)$ for findFather.
- add, del : since the list are ordered the average time complexity is $O(log(s))$ with s the average number of $SetAttr$ operation applied to an attribute. Using hash tables whose key are timestamp, the average complexity becomes $O(1)$.
- model2XML : in case of undo, the theoretical time complexity is $O(o)$ with o the number of operations – except undos/redos – applied to the whole tree. However all children and attributes of invisible edges will not be visited. Also, this function can be called incrementally, i.e. only on a node that becomes visible. A node that becomes invisible is simply removed from the view[7], and an operation on a non-special attributes has only a local affect.

Finally, concerning the scalability in term of operations number, the XML CRDT without undo requires tombstones for attributes as the Thomas Write Rule [10]. Also, there is an overhead effect observed experimentally by [31] : the CRDT position identifiers like $@position$ may grows if there is a big number of insert operations at a particular position, thus affecting the complexity of the function $child2List$ that sort children of an edge. This less likely to happen in an XML CRDT since insertion positions might be distributed among the whole tree.

7.2 Garbage collection

Undo requires to keep deleted elements (identifier and content) and the list of previous update values for attributes. Not surprisingly, the undo feature is provided as the cost

[7] For that, we need a function that computes the path in the view from the edge identifier with a complexity of $O(hc)$.

of keeping old operations. However, a garbage collecting mechanism can be designed. Such a garbage collection is similar to the one already present in the RFC 667 [10]. Each replica i maintains a vector v_i of the last clock timestamp received by all other replicas (including its own clock). From this vector i, computes m_i the minimum of these clocks. This minimum is sent regularly to the other replicas. It can be piggybacked to operation's message or sent regularly in a specific message. From the minimum received (including m_i), each replica maintains an other vector V_i. The minimum of V_i is M_i. The point is that, if communication are FIFO, a replica know that every replica have received all potential message with a timestamp less or equal to M_i. Thus any tombstone with a timestamp less or equal to M_i can be safely remove. This mechanism can be directly used in the XML CRDT without undo to remove old deleted attributes.

In the XML CRDT with undo, we only authorize to produce an undo of an operation whose timestamp is greater than m_i. Thus operations with a timestamp lesser than M_i will never see their effect modified. So, elements such as follow can be safely and definitively purged :

- attribute – including deletes – value v with $v.timestamp < M_i$ and $v.effect < 1$
- attribute – including deletes – value v with $v.timestamp < M_i$ and there exists v' with $v.timestamp < v'.timestamp < M_i$ and $v'.effect > 0$
- edge with any delete value d with $d.timestamp < M_i$ and $d.effect > 0$ or with the add value a with $a.timestamp < M_i$ and $a.effect < 1$.

Thus, the time and space complexity of the approach is greatly reduced to be proportional to the size of the view. Moreover, differently to the RFC 677, replicas send $m_i - k$ with k a global constant instead of m_i. Thus, even if replica are tightly synchronized – having m_i very close to their own clock –, the replicas can always undo the last operations.

For a replicas number of s, the garbage collection mechanism only requires additional storage space of size $O(s)$ and send additional message of size $O(1)$. However, it requires to knows the number of replica in the networks. This makes the mechanism unsuitable for P2P networks with a high degree of churn but still suitable for cloud infrastructures.

8 Conclusion

We have presented a commutative replicated data type that supports XML collaborative editing including a global selective undo mechanism. Our commutative replicated data type is designed to scale since the replicas number never impacts the operations execution complexity. Obviously, the undo mechanism requires to keep information about the operations we allow to undo. We presented a garbage collection mechanism that allows to purge old operation information.

Other tombstone-based approaches requires tombstones even for deleting element without undo. Also, the garbage collecting mechanism that can be adapted to them is much less scalable since based on a consensus-like method [14].

We still have much work to achieve on this topic. Firstly, we need to make experiments to establish the actual scalability and efficiency of the approach in presence of huge data. Secondly, we plan to study replication of XML data typed with DTD or XSD. This is difficult task, never achieved in a scalable way.

References

1. S. Abiteboul, A. Bonifati, G. Cobéna, I. Manolescu, and T. Milo. Dynamic xml documents with distribution and replication. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 527–538, New York, NY, USA, 2003. ACM.
2. G. D. Abowd and A. J. Dix. Giving undo attention. *Interacting with Computers*, 4(3):317–342, 1992.
3. T. Berlage. A selective undo mechanism for graphical user interfaces based on command objects. *ACM Trans. Comput.-Hum. Interact.*, 1(3):269–294, 1994.
4. M. Burrows. The chubby lock service for loosely-coupled distributed systems. In *OSDI*, pages 335–350. USENIX Association, 2006.
5. M. Cart and J. Ferrie. Asynchronous reconciliation based on operational transformation for P2P collaborative environments. In *Proceedings of the 2007 International Conference on Collaborative Computing: Networking, Applications and Worksharing*, pages 127–138. IEEE Computer Society, 2007.
6. R. Choudhary and P. Dewan. A general multi-user undo/redo model. In *ECSCW'95: Proceedings of the fourth conference on European Conference on Computer-Supported Cooperative Work*, pages 231–246, Norwell, MA, USA, 1995. Kluwer Academic Publishers.
7. 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.
8. C.-L. Ignat and G. Oster. Peer-to-peer collaboration over xml documents. In Y. Luo, editor, *CDVE*, volume 5220 of *Lecture Notes in Computer Science*, pages 66–73. Springer, 2008.
9. R. Jiménez-Peris, M. Patiño-Martínez, B. Kemme, and G. Alonso. How to select a replication protocol according to scalability, availability, and communication overhead. In *SRDS*, pages 24–33. IEEE Computer Society, 2001.
10. P. R. Johnson and R. H. Thomas. RFC 677: Maintenance of duplicate databases, January 1975, (Septembre 2005). `http://www.ietf.org/rfc/rfc677.txt`.
11. 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.
12. G. Koloniari and E. Pitoura. Peer-to-peer management of xml data: issues and research challenges. *SIGMOD Rec.*, 34(2):6–17, 2005.
13. A. D. Kshemkalyani and M. Singhal. Necessary and sufficient conditions on information for causal message ordering and their optimal implementation. *Distributed Computing*, 11(2):91–111, 1998.
14. M. Letia, 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. sigops, acm.
15. T. Lindholm. Xml three-way merge as a reconciliation engine for mobile data. In *MobiDe '03: Proceedings of the 3rd ACM international workshop on Data engineering for wireless and mobile access*, pages 93–97, New York, NY, USA, 2003. ACM.
16. J. Maeda. *The laws of simplicity*. MIT Press, 2006.
17. 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.

18. J. O'Brien and M. Shapiro. Undo for anyone, anywhere, anytime. In *EW 11: Proceedings of the 11th workshop on ACM SIGOPS European workshop*, page 31, New York, NY, USA, 2004. ACM.
19. G. Oster, P. Urso, P. Molli, and A. Imine. Data Consistency for P2P Collaborative Editing. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work - CSCW 2006*, pages 259–267, Banff, Alberta, Canada, nov 2006. ACM Press.
20. 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.
21. 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.
22. M. Ressel and R. Gunzenhäuser. Reducing the problems of group undo. In *GROUP '99: Proceedings of the international ACM SIGGROUP conference on Supporting group work*, pages 131–139, New York, NY, USA, 1999. ACM.
23. M. Ressel, D. Nitsche-Ruhland, and R. Gunzenhäuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *CSCW*, pages 288–297, 1996.
24. Y. Saito and M. Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1):42–81, 2005.
25. C. Sun. Undo as concurrent inverse in group editors. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 9(4):309–361, December 2002.
26. 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.
27. D. Sun and C. Sun. Operation Context and Context-based Operational Transformation. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work - CSCW 2006*, pages 279–288, Banff, Alberta, Canada, November 2006. ACM Press.
28. 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.
29. W. Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.
30. S. Weiss, P. Urso, and P. Molli. An Undo Framework for P2P Collaborative Editing . In *CollaborateCom*, Orlando, USA, November 2008.
31. S. Weiss, P. Urso, and P. Molli. Logoot-Undo: Distributed Collaborative Editing System on P2P Networks. *IEEE Trans. Parallel Distrib. Syst.*, 21(8):1162–1174, 2010.

Centre de recherche INRIA Nancy – Grand Est
LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex (France)

Centre de recherche INRIA Futurs : Parc Orsay Université - ZAC des Vignes
4, rue Jacques Monod - 91893 ORSAY Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399