Optimistic Replication for Massive Collaborative Editing

Gérald Oster, Pascal Urso, Pascal Molli, Hala Skaf-Molli, Abdessamad Imine

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Optimistic Replication for Massive Collaborative Editing

Gérald Oster — Pascal Urso — Pascal Molli — Hala Molli — Abdessamad Imine

N° 5719

Octobre 2005

Thèmes COG et SYM

ISSN 0249-6399

Optimistic Replication for Massive Collaborative Editing

Gérald Oster [*] , Pascal Urso [†] , Pascal Molli [‡] , Hala Molli [§] , Abdessamad Imine [¶]

Thèmes COG et SYM — Systèmes cognitifs et Systèmes symboliques
Projets ECOO et CASSIS

Rapport de recherche n° 5719 — Octobre 2005 — 18 pages

Abstract: In recent times, Wikipedia has opened the way to massive collaborative editing. More specially, it has demonstrated what can be achieved with a massive collaborative effort. Massive collaborative editing implies scalability however, pessimistic replication scales poorly in the wide area. Optimistic replication offers better performance but has severe drawbacks for maintaining consistency. In this paper, we propose a new optimistic replication algorithm for massive collaborative editing called WOOT. It is designed to scale, as well as to ensure eventual consistency and intention preservation. It is important to point out that WOOT efficiency does not depend on a number of sites and can be deployed on a very large pure peer-to-peer network.

Key-words: Real time groupware, optimistic replication, operational transformation, replicated data consistency

[*] ECOO
[†] ECOO
[‡] ECOO
[§] ECOO
[¶] CASSIS

Unité de recherche INRIA Lorraine
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

Réplication optimiste pour l'édition collaborative massive

Résumé : La Wikipedia a ouvert la voie à l'édition collaborative massive. Elle a montrée qu'un effort collaboratif massif peut produire une encyclopédie qui est aujourd'hui sans équivalent. Le passage à l'échelle du système d'édition collaboratif est primordial pour l'édition collaborative massive. Les systèmes basés sur la réplication pessimiste des données ne sont pas adaptés à notre contexte. Si La réplication optimiste offre de meilleures performances, elle souffre d'un problème récurrent de maintient de la cohérence des données répliquées. Nous proposons un algorithme de réplication optimiste pour l'édition collaborative massive. Il est conçu pour assurer la convergence des données, la préservation des intentions et le passage à l'échelle. Cet algorithme ne dépend pas du nombre de sites et peut être déployé sur un réseau pair à pair à large échelle.

Mots-clés : Editeurs temps rééls, réplication optimiste, transformées opérationnelles, cohérence des données réparties

1 Introduction

Collaborative editing allows users to edit the same text from multiple sites across Internet. For example, recently collaborative editing allowed Wikipedia [14] to collect more than 1,600,000 articles in more than 100 languages. Moreover, than 13 million of page requests per day, more than 4,000 changes are made every day by more than 12,000 active writers [9]. Thus, Wikipedia is an example of massive collaborative editing.

Scalability is one of the key issues for massive collaborative editing. For example, Wikipedia uses a database pessimistic replication approach [1]. Thus all changes are routed on a single master database that propagates changes to slaves within distributed atomic transactions. Consequently, the master database is a congestion point that limits scalability of this approach [15]. Wikipedia relies on "brutal force" to handle this load.

On the one hand, the optimistic replication approach [7] greatly improves performances. It can be deployed on peer-to-peer network and scales with cheap resources. On the other hand, it is more difficult to ensure consistency. Traditionally, the optimistic replication systems ensure eventual consistency i.e. replicas can diverge but must converge when system is idle.

Optimistic replication is suited for collaborative editing. CVS [2], real-time group editors [3, 11] are good examples of optimistic replication algorithms applied to collaborative editing. However, these systems have not been designed for massive collaborative editing. Consequently, they often require a central site, a total ordering, a consensus or vector clocks. These well known mechanisms or algorithms are not designed for a large number of sites.

The main issue for massive collaborative editing systems, based on optimistic replication, is to ensure eventual consistency and scalability. So far, only Usenet and DNS ensure both eventual consistency and scalability. However, they are not suited for collaborative editing.

WOOT is an optimistic replication algorithm designed for massive collaborative editing. It does not require the central site, total ordering, consensus or vector clocks. It does not use the number of sites as its parameter. It is designed to be deployed on a very large peer-to-peer network. In theory, WOOT can support Wikipedia editing with an optimistic replication approach. Futhermore, WOOT could allow massive collaborative application such as Wikipedia to be deployed on a cheap peer-to-peer network with better performances.

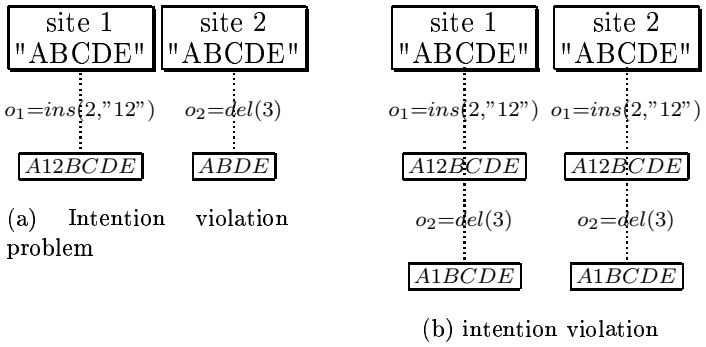


Figure 1: Intention violation problem

This paper focuses on sequence of characters but it is clear that the WOOT framework can support any linear structure. Thus a linear structure can be more complex e.g. in a form of an ordered tree that can be mapped into a linear structure.

Section 2 gives an overview of the WOOT approach. Section 3 describes formally the WOOT framework including the algorithms and the examples. Section 4 presents a formal evaluation of correctness of WOOT algorithms. Section 5 presents the related works. Section 6 summarises contributions of the paper and gives an overview of our future work.

2 Woot Approach

The traditional optimistic replication approaches do ensure eventual consistency [7]. Thus, in collaborative editing, real time group editors define the notion of intention preservation [10]. It means that when an operation is re-executed on remote sites, its effects on the generation state are preserved. Typically, rules as "the last writer wins" do not preserve intention. For example, in Wikipedia when two users edit concurrently the same article, the final article will just contain changes from the last writer. Consequently, this rule violates the intentions of the first writer. Manual merging is done a posteriori using versionning management.

The intention-violation problem has been introduced in the REDUCE approach [10]. Suppose that two sites share a string containing "ABCDE" (cf. figure 1(a)). Site 1 inserts "12" at position 2 and obtains "A12BCDE". Thus, site 1 has executed

operation $o_1 = ins(2, "12")$ with the intention to insert "12" between 'A' and 'B'. Suppose that site 2 deletes one character at position 3 and obtains "ABDE". Site 2 has executed operation $o_2 = del(3)$ with the intention to delete the character 'C'. If we execute both operations and preserve intentions, we must obtain "A12BDE". But if we use the serialization protocol, it may serialize o_1 before o_2. The final result in this case is "A1BCDE" (cf. figure 1(b)). Convergence is achieved but the intentions i.e. operation effects, are not preserved.

In this example, intention preservation means that if "12" has been inserted between 'A' and 'B', then this ordering $'A' \prec "12" \prec' B'$ must be preserved for any further states.

In order to ensure intention preservation, WOOT translates primitive text editing operations into orderings. Therefore instead of executing and broadcasting $insert(2, "12")$, it executes $insert(2, "12")$ and broadcasts $insert('A' \prec "12" \prec' B')$. When these operations are integrated, WOOT inserts "12" between 'A' and 'B'. Hence, the first correctness criterion of the WOOT approach is *intention preservation.*

A problem may occur if we receive $insert('A' \prec "12" \prec' B')$ and 'B' has been locally deleted. In the WOOT approach, 'B' will exist because a character is not deleted. Rather, it is marked as invisible.

Each insert operation generates two new order relationships, however, it does not generate a total order, just a partial one. For example, consider three sites where each site generates one operation as presented in figure 2(a). All relationships between characters are represented in the Hasse diagram (as depicted by figure 2(b)). From the diagram, we can notice that the states respecting intentions are the linear extensions of the partial order i.e. "a312b", "a321b", "a231b". In order to achieve convergence, all sites must have the same linear extension.

Each time an operation is received, a new linear extension is computed. This computation must be *monotonic* i.e. the new linear extension must be compatible with all previous ones. For example, if 'a' was placed before 'b', then a new linear extension cannot place 'b' before 'a'. Traditionally, a topological sort is used to find a linear extension of a partial ordered set. However, classical topological sort is not monotonic and consequently, not suitable in this context.

The challenge of the WOOT framework is to ensure convergence by using a monotonic linear extension function. Thus, when WOOT receives new character c, it inserts it between x and y i.e. $x \prec c \prec y$. However, there may be some other characters between x and y, concurrently inserted or previously deleted. Thus, c must be placed somewhere among these characters. To break the tie between

two unordered characters WOOT uses a total order based on the unique character identifier. However, to achieve convergence, *the method applied to place c must be independent from the order of reception of the characters.*

3 WOOT Framework

In this section, we present the WOOT model and its algorithms. We formally define the data structure used by WOOT and the order relations used to linearize characters. Finally, we describe the algorithms used by the WOOT framework.

A group editor is consistent if and only if, it satisfies the following two properties:

Convergence When the same set of operations has been executed at all sites, all copies of the shared document are identical.

Intention Preservation For any operation O, the effects of executing O at all sites are the same as the effects of executing O in its generation state.

3.1 Data model

WOOT manages W-characters by encapsulating the additional information about characters i.e. identifier, visibility and order relation.

Definition 1 *A* W-character c *is a five-tuple* $< id, v, \alpha, id_{cp}, id_{cn} >$ *where*

- id *is the identifier of the character.*
- $v \in \{True, False\}$ *indicates if the character is visible,*
- α *is the alphabetical value of the effect character,*
- id_{cp} *is the identifier of the previous W-character of* c*.*
- id_{cn} *is the identifier of the next W-character of* c*.*

The previous and the next W-characters of c are the W-characters between which c has been inserted.

Definition 2 *The previous W-character of* c *is denoted* $C_P(c)$*. The next W-character of* c *is denoted* $C_N(c)$*.*

Each site s has unique identifier $numSite_s$, logical clock H_s, sequence $string_s$ of W-characters and pool of pending operations $pool_s$. The site identifier and the local clock are used to identify characters in a unique way.

Definition 3 *A* character identifier *is pair* (ns, ng) *where* ns *is the identifier of a site and* ng *is a natural number. When a W-character is generated on site* s*, its identifier is* $(numSite_s, H_s)$*.*

Each time a W-character is generated on site s, the local clock H_s is incremented. Since $numSite$ is unique, the pair forms a unique identifier for a character.

Definition 4 *A* W-string *is an ordered sequence of W-characters* $C_b c_1 c_2 \dots c_n C_e$ *where* C_b *and* C_e *are the special W-characters (i.e. with special identifiers) that mark the beginning and the ending of the sequence.*

We define the following functions for sequence S:

- $|S|$ denotes the length of sequence S.
- $S[p]$ denotes the element at the position p in S. We state that the first element of sequence S is at position 0 and the last element is at position $|S| - 1$.
- $pos(S, c)$ returns the position of element c in S as a natural number.
- $insert(S, c, p)$ inserts element c in S at position p.
- $subseq(S, c, d)$ returns the part of sequence S between the elements c and d (excluding c and d).
- $contains(S, c)$ returns true if c can be found in S

The following functions are used to link the W-string with the string that user will eventualy see.

- $value(S)$ is the representation of S (i.e. the sequence of visible alphabetical values).
- $ithVisible(S, i)$ is i^{th} visible character of S.

The following two operations update a W-string:

ins(c) inserts W-character c between its previous and next characters under the condition that the previous and next characters exist.

del(c) deletes W-character c providing that c exists.

3.2 Orders

Definition 5 *Suppose that a and b are two W-characters.* $a \prec b$ *if and only if, there exists a set of characters* $c_0, c_1, ...c_i$ *such that* $a = c_0, b = c_i$ *and* $C_N(c_j) = c_{j+1}$ *or* $c_j = C_P(C_{j+1})$ *for all* $0 \leq j \leq i$.

where $\prec$ is a binary relation over the set of W-characters. $\prec$ is irreflexive, transitive and asymmetric. Thus, $\prec$ is a strict partial order.

To obtain a string from this partial order, we have to find a linear extension (i.e. a total order).

Definition 6 *Let* S *be a sequence, the relation* $\leq_S$ *is defined as* $a \leq_S b$ *if and only if* $pos(S, a) \leq pos(S, b)$

When no precedence relation can be established between two characters, it is necessary to order them. Furthermore, to ensure convergence, this order must be independent from the state of the sites. For this purpose, we use the characters identifier.

Definition 7 *Let a and b be two W-characters with their respective identifiers (ns_a, ng_a) and (ns_b, ng_b). $a <_{id} b$ if and only if (1) $ns_a < ns_b$ or (2) $ns_a = ns_b$ and $ng_a < ng_b$.*

3.3 Algorithms

When a site generates an operation, this operation is integrated locally, broadcasted and then integrated by all other sites. The reason for the local integration is to take into account the invisible characters that are previously deleted.

Generation. For an operation op, $type(op)$ denotes its type: del or ins. Also, $char(op)$ denotes the character manipulated by the operation.

When an user interacts with the framework, s/he only sees $value(S)$. So, when an insert operation is generated the user-interface only shows the visible position and the alphabetical value of the character to be inserted. For instance, $ins(2, a)$ in "xyz" is translated into $ins(x \prec a \prec y)$.

GenerateIns(pos, α)
  $H_s := H_s + 1$
  let $c_p := ithVisible(string_S, pos)$,
    $c_n := ithVisible(string_S, pos + 1)$,
    $wchar :=< (numSite_s, H_s), True, \alpha, c_p.id, c_n.id >$
  IntegrateIns$(wchar, c_p, c_n)$
  broadcast $ins(wchar)$

Similarly, when a delete operation is generated, it is necessary to retrieve the W-character from this position.

GenerateDel(pos)
  let $wchar := ithVisible(string_s, pos)$
  IntegrateDel$(wchar)$
  broadcast $del(wchar)$

Reception Sites may receive operations with unverified preconditions. In that case, the *isExecutable* function checks preconditions of an operation.

```
isExecutable(op)
  let c := char(op)
  if type(op) = del then
     return contains(string_s, c)
  else
     return contains(string_s, C_P(c)) and contains(string_S, C_N(c))
  endif
```

To deal with pending operations each site maintains a pool of operations.

```
Reception(op)
 add op to pool_s
```

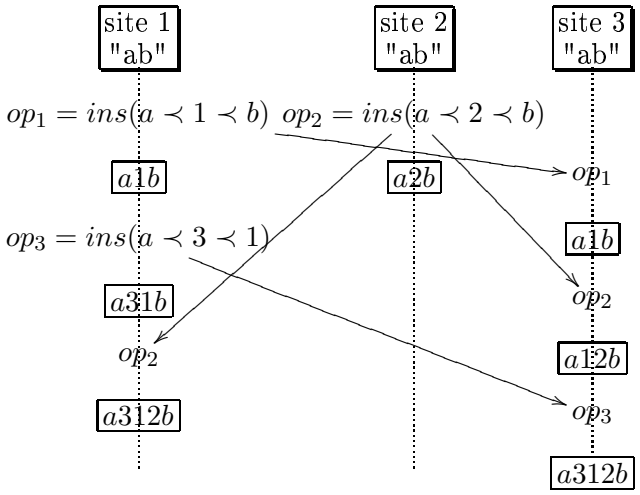
For instance, a site executes $del(c)$ only if c is present. If c is not present, the integration of the operation is delayed until c is present.

```
Main()
  loop
    find op in pool_s such that isExecutable(op)
    let c := char(op)
    if type(op) = del then
       IntegrateDel(c)
    else
       IntegrateIns(c, C_P(c), C_N(c))
    endif
  endloop
```

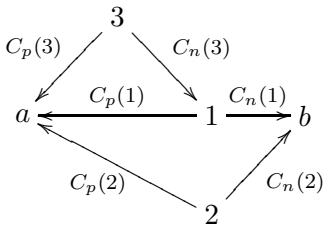
Integration To integrate an operation $del(c)$, we set the visible flag of character c to $False$, irrespectively of the previous value.

```
IntegrateDel(c)
  c.v := False
```

To integrate an operation $ins(c)$ in $string_s$, we need to place c among all the characters between c_p and c_n. These characters can be previously deleted characters or the ones inserted by concurrent operations. When operation $ins(c)$ is executed on a site, procedure $IntegrateIns(c, c_p, c_n)$ is executed.



(a) Generating partial orders



(b) C_p and C_n relationships

Figure 2: Partial orderings

IntegrateIns(c, c_p, c_n)
  let $S := string_s$
  let $S' := subseq(S, c_p, c_n)$
  if $S' = \varnothing$ **then**
    $insert(S, c, pos(S, c_n))$
  else
    let $L := c_p d_0 d_1 \ldots d_m c_n$ **where** $d_0 \ldots d_m$ **are the**
      W−char **in** S' **such that** $C_P(d_i) \leq_S c_p$ **and** $c_n \leq_S C_N(d_i)$
    let $i := 1$
    while $(i < |L| - 1)$ **and** $(L[i] <_{id} c)$ **do**
      $i := i + 1$
    endwhile
    IntegrateIns$(c,\ L[i-1],\ L[i])$
  endif

The algorithm orders characters with $<_{id}$ when no precedence relation $\prec$ is available. S' is the sequence of characters between c_p and c_n. if S' empty, c is inserted between c_p and c_n.

Otherwise, WOOT makes a copy of S' in L and removes from L all characters c_i with $C_p(c_i)$ or $C_n(c_i)$ between c_p and c_n. We explain this choice with figure 2(a). When op_2 is integrated on site 3, op_2 see only the string "a1b". Character '2' is compared to character '1' and then inserted after '1'. When op_2 is integrated on site 1, op_2 see the string "a31b". To be sure to make the same choice than on site 2, '2' must compared first with character '1' and maybe next with character '3'. WOOT detects that '3' must not be compared to '2' because $C_n(3)$ is between $C_p(2)$ and $C_n(2)$ (cf figure 2(b)). Thus, it can exist another site where character '2' is integrated and character '3' is not yet arrived as in figure 2(a) on site 3. By applying this strategy, we are sure that all characters in L are sorted by the $<_{id}$ relation. Next, WOOT has just to insert c in this sorted list. If i is the insert position of c in L, WOOT makes a recursive call to intergatedIns(c, $L[i-1]$,$L[i]$) where the subsequence bounded by $[L[i-1],L[i]]$ is strictly shorter than the sequence bounded by $[C_p, C_n]$.

3.4 Example

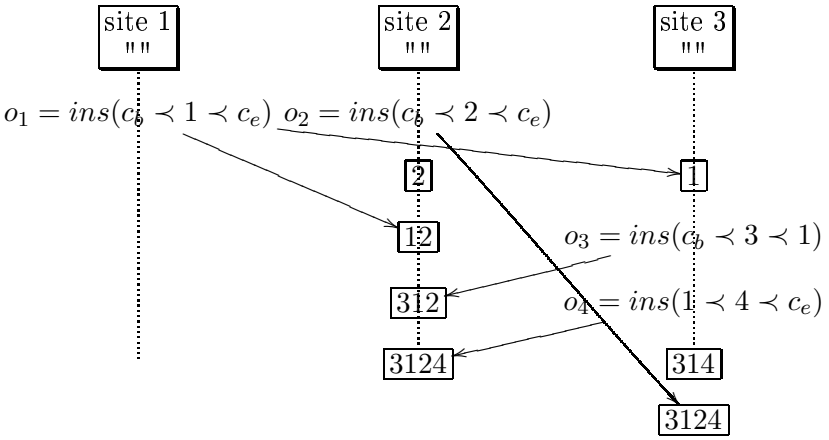
Suppose that three sites are in the initial state "$c_b c_e$". We consider the scenario shown by figure 3(a). It generates the orderings depicted by the Hasse diagram (figure 3(b)). We also assume that $<_{id}$ relation is defined as follows : '1' $<_{id}$ '2' $<_{id}$ '3' $<_{id}$ '4'.

Integration on Site 3. Site 3 receives o_1 and then generates o_3 and o_4. Thus, site 3 gets in the state "$c_b 314 c_e$".

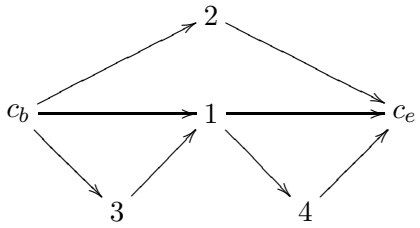
It integrates $o_2 = ins(c_b \prec 2 \prec c_e)$. '2' must be arranged among characters between c_b and c_e. Thus among "314" which are put in S'. $C_n(3)$ and $C_p(4)$ are between c_b and c_e so only '1' remains in L. WOOT compares '2' to '1' according to relation $<_{id}$. '1' $<_{id}$ '2'; therefore '2' must be inserted after '1'. Integration procedure is called recursively between '1' and c_e. Only '4' remains in L. '2' must be inserted before '4' as '2' $<_{id}$ '4'. Thus '2' must be inserted between '1' and '4'.

Finally, state of Site 3 becomes "$c_b 3124 c_e$".

Integration on Site 2. Site 2 generates o_2 to get state "$c_b 2 c_e$".



(a) Example scenario



(b) Hasse diagram

It integrates $o_1 = ins(c_b \prec 1 \prec c_e)$. '1' and '2' have the same previous and next characters. '1' must ordered with '2' according to relation $<_{id}$. As '1' $<_{id}$ '2', '1' is inserted before '2'. Site 2 has the state "$c_b 12 c_e$".

It integrates $o_3 = ins(c_b \prec 3 \prec 1)$. As there is no character between its branches, '3' is simply inserted. Site 2 obtains state "c_b312c_e".

It integrates $o_4 = ins(1 \prec 4 \prec c_e)$. '4' must be compared with '2' according to $<_{id}$. As '2' $<_{id}$ '4', '4' must be inserted after '2'. Thus, it must be inserted between '2' and c_e.

Finally, site 2 gets in state "c_b3124c_e".

Integration on Site 1. Site 1 generates o_1 to obtain state "c_b1c_e". Irrespectively of the arrival order of o_2, o_3 and o_4, WOOT computes c_b3124c_e. In all cases, the final string is "3124", thus convergence and intention preservation are ensured.

4 Correctness

Theorem 1 *The algorithm of integration terminates.*

Proof 1 *Proof by contradiction.*

The algorithm does not terminate, if and only if, the recursive call is not done on a strict subsequence. This can happen only if we get, non-empty S' and $L = c_pc_n$. If $L = c_pc_n$, every character in this S' has its predecessor or successor in S'. At least, the first integrated character between c_p and c_n has its previous and next characters outside of S'. This is because the characters have been generated in a strict order.

Thus, there are at least 3 characters in L. So the recursive call is done on a strictly smaller subsequence and the algorithm terminates.

Intention preservation Our linearization order must respect the precedence order defined when operations are generated.

Theorem 2 *Relation $\leq_s$ built by WOOT on each site, is a linear extension of the relation $\prec$.*

Proof 2 *Generation of an operation does not modify relation $\prec$. This relation is only modified through $IntegrateIns$. The integration of character c is always done by its insertion between $C_P(c)$ and $C_N(c)$. Thus $\leq_s$ is a linear extension of $\prec$.*

However, ensuring intention preservation is not sufficient. For instance, if two sites insert concurrently 'x' and 'y' between the same characters 'a' and 'b', the resulting strings can be "axyb" and "ayxb". These two linear extensions satisfy intention preservation but do not converge.

Convergence We do not have a manual proof of convergence. To verify the correctness of our algorithm, we have used the TLC model-checker on a specification modeled on the TLA+ specification language [16]. Note that Model-checking techniques are particularly suited to verify concurrent systems. With the TLC model checker we have verified a bounded model of WOOT. In practice, it is impossible to test a system with a large number of sites and characters due to the famous state explosion problem. **We verified convergence up to four sites and five characters.** It took about two weeks with a Pentium(R) 4 CPU $2.80GHz$. The TLA specification has been described in more details in the technical report [6] .

Convergence requires that two characters on two sites are linearized in the same order. The convergence criteria is ensured by the fact that every generated character will be inserted in every site.

Conjecture 1 *Let S_1 and S_2 be two W-strings maintained by two different sites. For every pair of characters $\{c, d\}$ that appear on both sites, we get*

$$c \leq_{S_1} d \Leftrightarrow c \leq_{S_2} d$$

To reduce design complexity of the specification and to accelerate verification of the proposed model, we have made two slight generalizations. Operation *del* and its integration do not appear in the model since this operation does not affect the linearization order. However, to simulate deletion of characters, we allow generation of $ins(c_p \prec c \prec c_n)$ in S that requires only $c_p \leq_S c_n$ i.e. as if the characters between c_p and c_n were deleted.

The second generalization assumes that characters are represented simply by an identifier. Thus according to this model generalization any site can generate any character identifier.

The TLC model-checker has found an error in a previous naive version of the integration algorithm. In this version, we did not filter S' to obtain L. We thought that since all the characters between c_p and c_n were concurrent to c we simply have to order them according to $<_{id}$. The model checker has found a counter example in the scenario presented in figure 2(b). This counter-example was helped us to design the current version of the integration algorithm.

5 Related works

Massive collaborative editing requires scalability, convergence and intention preservation.

In the operational transformation approach, only the SDT algorithm [5] ensures convergence and intention preservation. However, it uses vector clocks. This is not appropriate because in this context the actual number of sites is quite large and they come and go dynamically. On the other hand, WOOT does not require vector clocks because it is not use the number of sites as a parameter.

Another example, IceCube [4] also ensures convergence and intention preservation. But IceCube merges concurrent operations on one site. It means that all concurrent operations must be sent on one site for merging. Then, the merged log must be dispatched to all sites. Consequently, all sites must be connected during reconciliation and frozen until reconciliation is done. These constraints are not compatible with peer-to-peer networks. Compared to IceCube, WOOT proposes a distributed merge. It relies on a *monotonic* linearization function. If IceCube linearizes concurrent operations, its linearization function is not monotonic. Consequentley, its merging operation must be computed on one site.

Furthermore, Bayou [12] relies on the primary site to enforce the global continuous order on a prefix of history. This prefix is shared by all sites. Bayou maintains this global continuous order using a primary site. However, this site represents also a congestion point.

Usenet [8] ensures convergence, intentions and scalability. But although it is a groupware tool, it cannot be considered as a collaborative editing tool. This is because a Usenet message cannot be edited.

The Thomas's write rule [13] is heavily used in epidemic algorithms to achieve convergence. To ensure scalability Thomas's write rule needs to implement the strategy of "Last Writer Wins". However, this strategy does not ensure intention preservation. Consequently, it is not possible to build a massive collaborative editing system with Thomas write rule.

CVS [2] is a popular configuration management tool. It relies on an optimistic replication algorithm to ensure convergence. It also uses a central server that enforces a global continuous order. For example, if n sites produce a change, during round 1 only one site will be able to publish its changes. During round 2 all other sites have to update their replica. During round 3, only one site will be able to commit and so on. Thus, convergence will be achieved in $2n-1$ rounds. Compared to CVS, WOOT converges in one round.

In summary, WOOT is the only optimistic replication algorithm that ensures convergence, intention preservation and scalability. It requires no vector clocks as SDT. Unlike IceCube, it relies on a distributed merge. Moreover, WOOT does not require a primary site as Bayou. Unlike systems based on Thomas's write rule, WOOT

ensures intentions preservation. Finally, compared to CVS, WOOT converges in 1 round.

All these characteristics make WOOT the only algorithm suitable for massive collaborative editing.

6 Conclusion

WOOT is designed for massive collaborative editing. Based on a monotonic linearization function, it ensures convergence and intention preservation. It does not depend on a number of sites and can handle very large peer-to-peer networks. The proposed WOOT framework has been formally verified with model checking tools.

As a proof of concept, we have implemented a simple WOOT editor in Java deployed on a multicast network. It can be downloaded from `http://www.loria.fr/~molli/woot`.

Our current work includes further verification of WOOT correctness, group undo features, garbage collector as well as support for the XML tree.

7 Acknowledgments

Special thanks to Florent Jouille for his implementation of WOOT in few days. Thanks to Olivera Marjanovic for reviewing.

References

[1] Wikipedia Architecture. *Online http://meta.wikimedia.org/wiki/Wikimedia_servers*, (2005).

[2] Brian Berliner. CVS II : Parallelizing Software Development. In *Proceedings of USENIX*, Washigton D. C., 1990.

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

[4] Anne-Marie Kermarrec, Anthony Rowstron, Marc Shapiro, and Peter Druschel. The IceCube approach to the reconciliation of divergent replicas. In *Proceedings of the Twentieth ACM Symposium on Principles of Distributed Computing (PODC)*, Newport RI, USA, August 2001.

[5] Du Li and Rui Li. Preserving operation effects relation in group editors. In *Proceedings of the 2004 ACM conference on Computer supported cooperative work (CSCW'04)*, pages 457–466, New York, NY, USA, 2004. ACM Press.

[6] GÃ©rald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Real time group editors without operational transformation. Rapport de recherche, LORIA–INRIA Lorraine, May 2005.

[7] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Computing Surveys*, 37(1):42–81, 2005.

[8] Rich Salz. InterNetNews: Usenet transport for Internet sites. In *USENIX conference proceedings*, pages 93–98, San Antonio, Texas, USA, Summer 1992. USENIX.

[9] Wikipedia Statistics. *Online http://meta.wikimedia.org/wiki/Statistics*, (2005).

[10] Chengzheng Sun and Clarence A. Ellis. Operational transformation in real-time group editors: Issues, algorithms and achievements. In *Proceedings of the 1998 ACM conference on Computer Supported Cooperative Work (CSCW'98)*, pages 59–68, New York, NY, USA, November 1998. ACM Press.

[11] Chengzheng Sun, Xiaohua Jia, Yanchun Zhang, Yun Yang, and David Chen. Achieving convergence, causality-preservation and intention-preservation in real-time cooperative editing systems. *ACM Transactions on Computer-Human Interaction (TOCHI)*, 5(1):63–108, March 1998.

[12] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl 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.

[13] Robert H. Thomas. A solution to the concurrency control problem for multiple copy databases. In *Proceedings of the sixteenth IEEE Computer Society International Conference (COMPCON'78)*, pages 56–62, New York, NY, USA, Spring 1978. IEEE Computer Society.

[14] Wikipedia. the free encyclopedia that anyone can edit. *Online http://www.wikipedia.org/*, (2005).

[15] Haifeng Yu and Amin Vahdat. The costs and limits of availability for replicated services. In *SOSP*, pages 29–42, 2001.

[16] Panagiotis Manolios Yuan Yu and Leslie Lamport. Model checking TLA+ specifications. In *Proceedings of Correct Hardware Design and Verification Methods (CHARME'99)*, pages 54–66, 1999.

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

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

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