Data Consistency for P2P Collaborative Editing

Gérald Oster
Institute for Information Systems
ETH Zurich
osterg@inf.ethz.ch

Pascal Urso, Pascal Molli, Abdessamad Imine
Université Henri Poincaré, Nancy 1
LORIA
{urso,molli,imine}@loria.fr

ABSTRACT

Peer-to-peer (P2P) networks are very efficient for distributing content. We want to use this potential to allow not only distribution but collaborative editing of this content. Existing collaborative editing systems are centralised or depend on the number of sites. Such systems cannot scale when deployed on P2P networks. In this paper, we propose a new model for building a collaborative editing system. This model is fully decentralised and does not depend on the number of sites.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications*; H.5.3 [**Information Interfaces and Presentation**]: Group and Organization Interfaces—*Collaborative computing, theory and models*

General Terms

Algorithms, Design, Human Factors

Keywords

CSCW, Collaborative editing, Optimistic replication, Concurrency control

1. INTRODUCTION

Currently, peer-to-peer systems demonstrated how they can ensure scalable content distribution. In their survey [4], Androutsellis-Theotokis et al. wrote:

> "Peer-to-peer content distribution systems rely on the replication of content on more than one node for improving the availability of content, enhancing performance and resisting censorship attempts."

We want to reuse these characteristics not only for content distribution but also for content editing. Currently, P2P networks mainly distribute immutable contents, we want to distribute updates on this content and manage collaborative editing on it. We are convinced that if we can deploy a group editor framework on a P2P network, we open the way for P2P content editing. It means that all existing collaborative editing applications such as CVS and Wiki can be redeployed on P2P networks and take advantage of availability improvements, performance enhancements and censorship resistance of P2P networks. For instance, Wikipedia [2] is currently a collaborative encyclopædia that has collected more than 4,700,000 articles in more than 200 languages. Wikipedia has more than 50 million of page requests per day. 200,000 changes are made every day [1]. However, Wikipedia needs a costly infrastructure to handle the load. Hundreds of thousands of dollars are spent every year to fund the infrastructure. A P2P massive collaborative editing system would allow to distribute the service, tolerate failures, improve performances, resist to censorship and share the cost of the underlying infrastructure.

Collaborative editing systems such as CVS or Wikis are currently centralised and cannot be adapted to peer-to-peer networks. Collaborative systems based on the operational transformation approach [7, 23] can be decentralised.

However, existing algorithms such as GOTO [23], ABT[16] and SOCT2 [21] rely on vector clocks to detect concurrent operations. OT approach supposes that each operation is immediately executed locally, stored in a local log and then broadcast to other sites in order to be re-executed and stored in their logs. A vector clocks is associated to each operation. A vector clocks [17] is an array of logical clocks, one clock per site. It is used to detect the happened-before relationship and therefore the concurrency between operation. It causes no problem if the number of sites is fixed and low but if the number of sites grows, the size of the vector clocks is unbounded. Thus, messages exchanged between sites will grow as well as the size of local operation. Also, the time efficiency of operation on vectors clocks will decline as vectors clocks grow. Clearly, vectors clocks prevent these algorithms to scale and represent a serious bottleneck for their deployment on P2P networks.

In this paper we propose a new model called WOOT for building group editors that is suitable for dynamic P2P systems. Compared to existing decentralised group editor models, the number of sites involved in group editing is not a variable.

The remainder of this paper is organised as follows: Section 2 describes the WOOT approach and details the consistency model. Section 3 presents a formal definition of

CSCW'06, November 4–8, 2006, Banff, Alberta, Canada.

WOOT. Section 4 discusses about the correctness of the WOOT algorithm. Section 5 compares with related previous works. Section 6 summarises the contributions of this paper and presents some perspectives.

2. WOOT APPROACH

We consider a network of peers. In order to modify local data, a peer generates operations. Each operation is:

1. executed immediately by the peer;
2. broadcast through the P2P network to all other peers. The way that the new operation is broadcast is not under the scope of this paper. It can be realised by epidemic propagation [8] with anti-entropy protocol [6]. This protocol ensures that dissemination is achieved despite process crashes, disconnections, packet losses and dynamic network topology. This protocol is used for a long time by the Usenet network;
3. received and integrated by the other peers.

A group editor is consistent if it always maintains three properties [24]: intention preservation, causal consistency and convergence.

The *intention* of an operation is the effect observed on the state when it was generated. Intention preservation criterion as defined by Sun et al. [24] requires that for "any operation *op*, the effects of executing *op* at all sites are the same as the intention of *op*, and the effect of executing *op* does not change the effects of independent operations".

The operation intention definition depends of the data manipulated and of the primitives that affect this data. We consider peers replicating a linear structure. In this paper, for simplicity, we focus on a string, but we can apply the same algorithm for lists, blocks of texts, and ordered trees. Every editing action of a linear structure can be expressed in terms of the two following primitive operations:

- $ins(a \prec e \prec b)$ inserts the element e between the element a and the element b.
- $del(e)$ deletes the element e.

We choose this set of operations instead of the equivalent but more usual operations $ins(p, e)$ and $del(p)$ that inserts the element e at position p and suppresses the element at position p. The set we choose allows to execute the operations rather independently of the state of the peer. As shown in Figure 1 traditional operations can be directly executed when received. Our operations directly represent the operation effect relation as defined in [14].

Since our definitions of operation are based on elements rather than positions, the elements are uniquely identified. Thus an operation $del('c')$ can be executed unambiguously on the string "*acce*". Also, the case where a peer receives $ins('a' \prec 'b' \prec 'c')$ after '*c*' has been locally deleted must be treated. We reuse the approach of "death certificates" or "tombstones" used in Usenet. If an element is deleted we maintain useful informations about it but not its whole content. For a character, it is equivalent to make the character invisible. If we manage blocks instead of characters, it means that we maintain the identity of the block, but not its content.

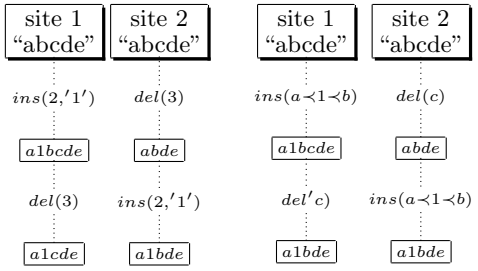
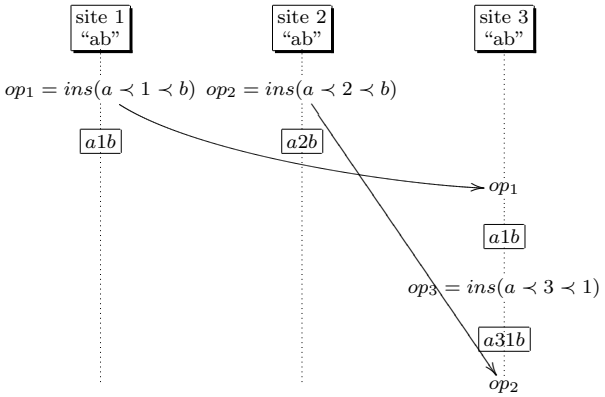


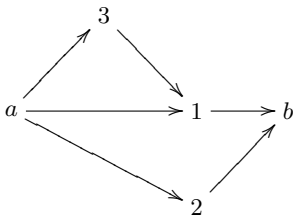
Figure 1: Sets of primitive operations

Operations as defined have also precondition on their execution. The execution of $del(e)$ requires e, or its tombstone, to be present. The execution of $ins(a \prec e \prec b)$ requires a and b, or their tombstones, to be present. The respect of preconditions ensures the *causal consistency* criteria [3]. All operations are executed on a state where they are legal.

Finally, *convergence* criterion states that peers with the same set of editing operations compute the same state of the replicated data. A direct way to ensure convergence is that the state of the data does not depend on the order that a peer executes received operations.



(a) Generating partial orders



(b) Ordering relationships

Figure 2: Partial orderings

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 depicted in Figure 2(b). From the diagram, we can notice there are multiple states respecting intentions. All of them are linear extensions of the partial order: "a312b",

"a321b", "a231b". To achieve convergence, all sites must compute 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. Indeed, if 'a' was placed before 'b', then a new linear extension must not place 'b' before 'a'. In Figure 2(a), execution of $[op_1, op_2, op_3]$ at site 1, must give the same result as $[op_1, op_3, op_2]$ at site 3 and $[op_2, op_1, op_3]$ at site 2. Traditionally, a topological sort is used to find a linear extension of a partial ordered set. However, a 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 a 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, this identifier order is not sufficient to achieve convergence. For instance, when site 3 receives the operation $ins(a \prec 2 \prec b)$ there is already '3' and '1' between 'a' and 'b'. If we suppose that the identifier order is '1'$<_{id}$'2'$<_{id}$'3', the system has to choose between placing '2' after '1' or before '3'. The way we resolve this problem is to take into account that on another site the order of execution can be $[op_1, op_2, op_3]$. On that hypothetical site, when '2' is received there is only '1' between 'a' and 'b'. Thus, '2' must be placed after '1'. WOOT uses the information that since the insertion of character '3' depends on the presence of '1', '3' will always appear after '1'. Thus, WOOT orders in priority '2' with '1' at every site.

Thus, to achieve convergence, *the method applied to place a new element is designed to 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 linearise characters. Finally, we describe the algorithms used by the WOOT framework.

This paper focuses on sequence of characters but it is clear that the WOOT framework can support any linear structure. Such a linear structure can be complex. An ordered tree – like XML documents – can be mapped into a linear structure.

A group editor is consistent if and only if, it satisfies the following 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 on its generation state.
- **Causality preservation** For any pair of operations O_a and O_b , if $O_a \rightarrow O_b$, then O_a is executed before O_b at all sites.

This consistency model is nearly equivalent to the model from Sun et al. [24]. The only difference concerns the definition of the relation $\rightarrow$. In [24], the relation used is the Lamport's *happened-before* relationship [13], and thus causality is based on time. Indeed, based on this definition, it is assumed one operation op_2 causally depends on another operation op_1 if op_2 is generated at one site after op_1 has been executed at that site. If this is a natural way of expressing causality, it only specifies a potential causal dependency between operations. op_2 might not depend on the execution effect of op_1. For instance, consider a file system on which two operations are performed $op_1 = createFile(a)$ next $op_2 = createFile(b)$. There is no causal dependence between these operations since op_2 could be executed before op_1. But, using the Lamport's happened-before relation, op_1 precedes op_2, and this ordering has to be preserved at every site. On the contrary, in WOOT, the definition of the relation $\rightarrow$ relies on the semantic causal dependency as in [12]. This dependence is explicitly declared regarding the preconditions of an operation. In this manner, in the previous example, op_1 could be executed before op_2 at one site, and op_2 before op_1 at another site. But, consider the two operations $op_1 = createDir(d)$ and $op_2 = createFile(d/a)$. There exists a causal dependency since op_2 requires that the directory d must exist before being executed. This dependence can be extracted from the preconditions of the operation op_2.

3.1 Data Model

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

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

- id *is the identifier of the character.*
- α *is the alphabetical value of the effect character,*
- $v \in \{True, False\}$ *indicates if the character is visible,*
- 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 on its generation state.

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

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

Each time a W-character is generated at site s, the local clock H_s is incremented. Since $numSite$ is unique, this 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 special W-characters that mark the beginning and the ending of the sequence.*

We define the following functions for a W-string S:

- $|S|$ denotes the length of S.
- $S[p]$ denotes the element at the position p in S. We state that the first element of W-string 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.
- $insert(S, c, p)$ inserts element c in S at position p.
- $subseq(S, c, d)$ returns the part of 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 user sees.

- $value(S)$ 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 precondition that the previous and next characters exist.
- $del(c)$ deletes W-character c providing that c exists.

3.2 Orders Notations

From the relation between a W-character and its previous and next characters, we can compute the precedence relation $\prec$. These relation represents the intention.

DEFINITION 5. *Let* a *and* b *be 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* $0 \leq j < i$.

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

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 6. *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* $ng_a < ng_b$.

DEFINITION 7. *Let* S *be a sequence, the relation* $\leq_S$ *is defined as* $a \leq_S b$ *if and only if* $pos(S, a) \leq pos(S, b)$. *The relation* $<_S$ *is defined as* $a <_S b$ *if and only if* $pos(S, a) < pos(S, b)$.

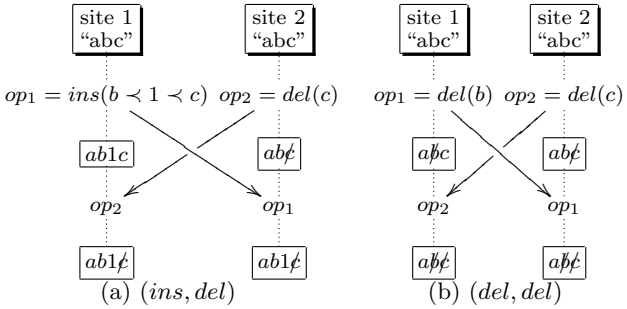


Figure 3: Commutation of (del, ins) **and** (del, del)

3.3 Algorithms

In order to ensure convergence whatever the reception order of operations is, pairs of operations (ins, del), (del, del) and (ins, ins) have to commute. Thanks to their definitions, pairs of operations (ins, del) and (del, del) commute as shown in Figure 3.

However, the definition of operation insert is not sufficient to make the pair (ins, ins) commutable. In this section, we then describe all the algorithms defined by WOOT. In particular we present how WOOT computes a unique linear extension of a partial order independently from the reception order of operations.

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

When a user interacts with the framework, she 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 $c_p := ithVisible(string_s, pos)$,
    $c_n := ithVisible(string_s, pos + 1)$,
    $wchar :=< (numSite_s, H_s), \alpha, True, c_p.id, c_n.id >$
  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 the position.

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

Reception. Sites may receive operations with unsatisfied preconditions. 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 its previous value.

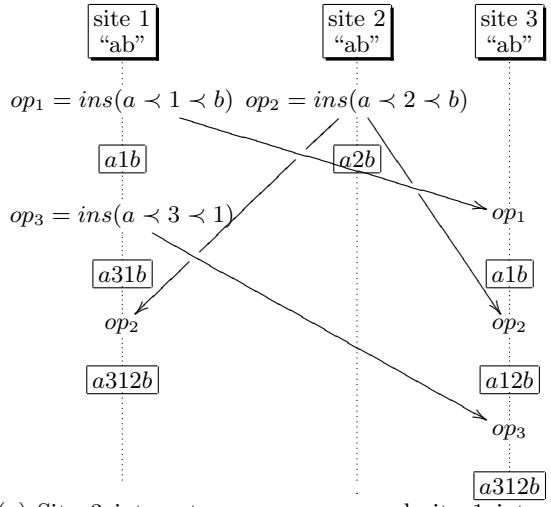
IntegrateDel(c)
$c.v := False$

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 at a site, procedure $IntegrateIns(c, c_p, c_n)$ is executed.

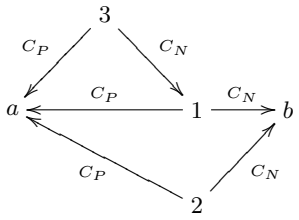
IntegrateIns(c, c_p, c_n)
let $S := string_s$
let $S' := subseq(S, c_p, c_n)$
if $S' = \varnothing$ **then**
$insert(S, c, pos(S, c_n))$
else
let $L := c_p d_0 d_1 \dots d_m c_n$ **where** $d_0 \dots d_m$ **are the**
W−char **in** S' **such that** $C_P(d_i) \leq_S c_p$
and $c_n \leq_S C_N(d_i)$
let $i := 1$
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' is 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 4(a). When op_2 is integrated at site 3, op_2 sees only the string "a1b". Character '2' is compared to character '1' and then inserted after '1'. When op_2 is integrated at site 1, op_2 sees the string "a31b". To be sure to make the same choice as at site 3, '2' must be 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 4(b)). Thus, another site may exist where character '2' is integrated and character '3' is not yet arrived as in Figure 4(a) at site 3.



(a) Site 3 integrates op_1, op_2, op_3 and site 1 integrates op_1, op_3, op_2



(b) C_p and C_n relationships

Figure 4: $IntegrateIns$ **and reception orders**

By applying this strategy, we are sure that all characters in L are sorted by the $<_{id}$ relation (see Theorem 3). Next, WOOT has just to insert c in this sorted list. i is the insert position of c in L and WOOT makes a recursive call to $IntegratedIns(c, L[i-1], L[i])$ where the subsequence of S bounded by $[L[i-1], L[i]]$ is strictly shorter than the sequence bounded by $[c_p, c_n]$.

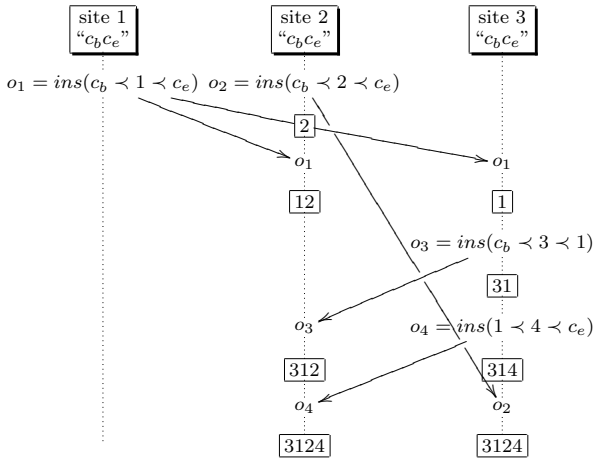
3.4 Example

Suppose that three sites are in the initial state "$c_b c_e$". We consider the scenario shown by Figure 5(a). It generates the orderings depicted by the Hasse diagram from Figure 5(b). The relation $<_{id}$ is defined as follows : '1' $<_{id}$ '2' $<_{id}$ '3' $<_{id}$ '4'.

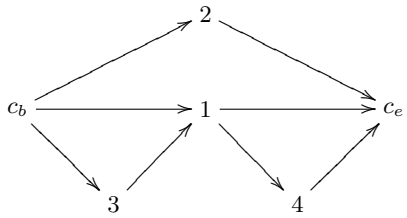
- **Integration at Site 3.**

  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 placed among the characters between c_b and c_e. Thus, S'



(a) Example scenario



(b) Hasse diagram

Figure 5: WOOT integration example

is “314”. $C_N(3)$ 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 at Site 2**

  Site 2 generates o_2 to get state “$c_b 2 c_e$”.

  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 c_b and ’1’, ’3’ is directly inserted. Site 2 obtains state “$c_b 312 c_e$”.

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

  Finally, site 2 gets in state “$c_b 3124 c_e$”.

- **Integration at Site 1**

  Site 1 generates o_1 to obtain state “$c_b 1 c_e$”. Whatever arrival orders of o_2, o_3 and o_4, WOOT computes $c_b 3124 c_e$. In all cases, the final string is “3124” and convergence and intention preservation are ensured.

4. CORRECTNESS

THEOREM 1. *The algorithm of integration terminates.*

PROOF. 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 $L = c_p c_n$. If $L = c_p c_n$, 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'. Indeed, 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 linearisation order must respect the precedence order defined when operations are generated.

THEOREM 2. *Relation $<_s$ built by WOOT at each site, is a linear extension of the relation $\prec$.*

PROOF. 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 $<_s$ 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.

WOOT is well-founded. The procedure $IntegrateIns$ is be well-founded if L is sorted by the $<_{id}$ relationship. All the characters in L have their previous and their next characters outside S'. Thus, each of them is between the previous and the next characters of each other. We say that such characters are W-concurrent.

DEFINITION 8. *Let x and y be two W-characters integrated at a site S. We say that x and y are W-concurrent when $C_P(x) <_S y <_S C_N(x)$ and $C_P(y) <_S x <_S C_N(y)$.*

The following theorem says that W-concurrent characters are ordered according $<_{id}$ total order.

THEOREM 3. *Let x and y be two W-concurrent characters, we get that $x <_S y$ if and only if $x <_{id} y$.*

PROOF. Proof by induction.

Let’s assume that every W-concurrent characters are ordered according to $<_{id}$. Now, we demonstrate by contradiction that the integration of a new W-character x will be consistent with the theorem.

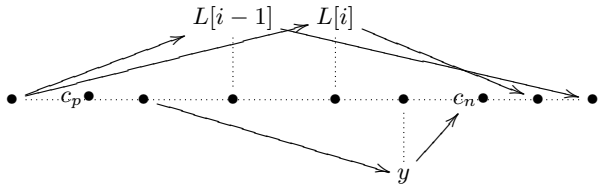


Figure 6: L is sorted by $<_{id}$

Let's suppose x and y are W-concurrent and $x <_S y$, but $y <_{id} x$. By definition of $IntegrateIns$, this is possible only if x is not compared to y, i.e. y neither belongs to any sequence L during the integration of x.

Thus, in all the recursive calls to $IntegrateIns(x, c_p, c_n)$, we get either:

(1) $y \in S'$ and $y \notin L$,

(2) or $y \notin S'$ and $C_P(y) \leq_S c_p <_S c_n \leq_S y <_S C_N(y)$.

The integration is a series of steps like (1) followed by a series of steps like (2). During the last step like (1), we get $C_P(y) \leq_S L[i-1] <_S L[i] \leq_S y <_S C_N(y)$ and $L[i-1] <_{id} x <_{id} L[i]$ (cf. Figure 6). Since $L[i]$ belongs to L, we get $C_P(L[i]) \leq_S c_p <_S y <_S c_n \leq_S C_N(L[i])$. Thus, $L[i]$ is W-concurrent to y and these both W-characters are ordered according to $<_{id}$.

However, we get $y <_{id} x <_{id} L[i]$ and $L[i] \leq_S y$. We obtain a contradiction. □

Thus, L is sorted by the $<_{id}$ relationship, and our algorithm is well-founded.

Convergence. To verify the correctness of our algorithm, we have used the TLC model-checker on a specification modelled on the TLA+ specification language [28]. 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. The TLA specification has been described in more details in a research report [18].

5. RELATED WORK

In the last decade, the operational transformation approach [7, 24] revealed as a suitable control mechanism for maintaining consistency of shared data in collaborative editing. In this approach, local operations are executed immediately after their generation, whilst remote operations must be transformed regarding concurrently executed operations. Algorithms based on this approach can be classified mainly in two categories. The first category regroups algorithms such as SOCT4 [27] and FORCE [20]. These algorithms use a central server for exchanging and timing operations, and thus, they are not suitable for deploying a group editor on a peer-to-peer network. The second category includes algorithms that are completely decentralised such as SOCT2 [21], GOTO [23], LBT [15] and ABT [16]. In these algorithms, after an operation is generated at one site, it is broadcast to all other sites. This diffusion relies on a group multicast protocol which generally need to know all other sites involved in the collaboration. Moreover, in order to integrate in their natural order operations which are causally related, each operation is timed using a vector clocks. The size of a vector clocks is proportional to the number of sites in the group. Consequently, these algorithms are not appropriate for peer-to-peer networks due to the huge number of sites constituting the system and the fact a site never knows all other sites participating to the collaboration. On the contrary, WOOT does not use vector clocks, it maintains a unique identifier for every character that ever appears in the system. However, maintaining such unique identifier should not affect the scalability of the approach. An identifier has a constant size which is equal to the size of only one component of the vector clocks. Indeed, a unique identifier is one couple $(numSite_s, H_s)$, while a vector clocks is a vector of couples $(numSite_i, H_i)$ where $0 \leq i < number\,of\,site$. As a future work, we plan to study in more details the impact of using unique identifiers comparing to vectors clocks to confirm this assertion.

Nevertheless, it is worth to point out that propositions based on the operational transformation were the first work to discuss about preserving intentions of operations. In [24], Sun et al. state that a consistency maintenance mechanism must ensure convergence of copies but also preserve the original effects of operations. Unfortunately, they did not define formally this criterion and did not provide any trail on how verifying if an algorithm preserves operations intentions. The model proposed by Li et al. [14] was the first proposition to define formally and explicitly the intentions of its operations. Our definition of local effect of our operations is quite similar to their definition – at all sites, a character will always be inserted after its previous character and before its next character –.

In collaborative editing, the *Mark & Retrace* technique [10] is the closest work to our proposal. This technique is based on two procedures executed consecutively: the *retrace* and *range-scan* procedures. The range-scan procedure is quite similar to our $IntegrateIns$ procedure. Its role is to insert a character between its previous c_p and next character c_n while arranging it among characters that have been concurrently inserted or deleted between c_p and c_n. However, in the Mark & Retrace technique, an insert operation does not store the previous and next characters between which it will insert. It stores only the insert position. Consequently, the algorithm has to seek these two characters. This is done by the retrace procedure which is called when a remote operation has to be integrated. Its goal is to retrace the document's address space to the state at the time the operation was generated. Characters which were present at the generation time – including the ones which have been deleted afterwards – are marked effective, while all other characters are set ineffective. To time operations, a vector clocks is associated to each operation. The use of vector clocks is a weak point in large-scale systems such as peer-to-peer systems, and the necessity to execute the retracing procedure each time a remote operation is received is too costly. On the contrary, WOOT identifies in a unique manner each character, and, the insert operation stores the previous and the next characters between which the insertion has to be performed. These features restrain the retracing mechanism.

IceCube [11] is able to ensure convergence and intention preservation. But, in IceCube, concurrent operations are merged at one site. It means that all concurrent operations must be sent at 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 reconcil-

iation is completed. These constraints are not compatible with peer-to-peer networks. Compared to IceCube, WOOT proposes a distributed merge. Each site merges its own copy as soon as operations are received. The result is independent of the reception order.

The Bayou system [25] was proposed to support collaboration among users who cannot be or decide not to be continuously connected. Operations are broadcast between site using an epidemic propagation protocol. It should make it suitable for deploying a collaborative application on a peer-to-peer network. Unfortunately, in order to ensure convergence of copies, Bayou has to arrange eventually operations in the same order. To achieve this, it relies on a primary site that will enforce a global continuous order on a growing prefix of history. Using such a primary site may constitute a congestion point, and, anyway is not suitable in a peer-to-peer system.

The Thomas' write rule [26] is heavily used to achieve convergence in system based on an epidemic propagation. To ensure scalability Thomas' write rule needs to implement the strategy of "Last Writer Wins". However, this strategy does not ensure intention preservation: the last update is applied while other concurrent updates are simply ignored. Consequently, it is not possible to build a collaborative editing system with Thomas' write rule.

CVS [5] is a popular configuration management tool. It relies on an optimistic replication algorithm to allow users working in isolation while ensuring convergence of copies. It also uses a central server that enforces a global continuous order on updates. 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 an optimistic replication algorithm that ensures convergence and intention preservation. It requires no vector clocks contrary to most operational transformation based algorithms. Unlike IceCube, it relies on a distributed merge. Moreover, WOOT does not require a primary site as Bayou. Unlike systems based on Thomas' write rule, WOOT ensures intentions preservation. Finally, compared to CVS, WOOT converges in one round. As far as we are aware, all these characteristics make WOOT the only model suitable for deploying group editors on peer-to-peer networks.

However, WOOT retains tombstones to record deleted characters. So, the space overhead of tombstones grows indefinitely. We can use an expiration period for garbaging deleted characters but this method is unsafe. We can also use a two-phase protocol to purge safely the tombstones as in [19] but all sites must be alive for the algorithm to make progress. For now, we prefer to keep tombstones for enabling group undo. Group undo is an important feature for collaborative editing [22, 9]. Group undo allows not only to undo the last operation, but any operation in the executed log. It requires to keep tombstones at least for a long time. Anyway, we need an algorithm to garbage old deleted characters.

6. CONCLUSION

WOOT is group editor framework suitable for peer-to-peer networks. P2P networks help group editors to function, scale and self organise in presence of a highly transient population of node, network and computer failures. Group editors allow P2P systems not only to distribute contents but also to edit contents.

Compared to decentralised distributed systems, WOOT does not require vector clocks to ensure convergence, intentions and causality.

WOOT is only a first step towards massive collaborative editing. It solves only the problem of data consistency on a P2P collaborative system. Many problems are now open: how to ensure security on these systems? how to provide awareness? how to enact collaborative processes? how to replicate just fragment of data? All these questions need answers before starting to port a system such as Wikipedia on a P2P network.

7. REFERENCES

[1] Wikipedia Statistics, (2006). Online http://meta.wikimedia.org/wiki/Statistics.

[2] Wikipedia. The Free Encyclopædia that Anyone Can Edit, (2006). Online http://www.wikipedia.org/.

[3] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. Causal Memory: Definitions, Implementation, and Programming. *Distributed Computing*, 9(1):37–49, March 1995.

[4] S. Androutsellis-Theotokis and D. Spinellis. A Survey of Peer-to-Peer Content Distribution Technologies. *ACM Computing Surveys*, 36(4):335–371, December 2004.

[5] B. Berliner. CVS II: Parallelizing Software Development. In *Proceedings of the USENIX Winter Technical Conference*, pages 341–352, Washington, D. C., USA, January 1990. USENIX Association.

[6] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic Algorithms for Replicated Database Maintenance. In *Proceedings of the ACM Symposium on Principles of Distributed Computing - PODC'87*, pages 1–12, Vancouver, British Columbia, Canada, August 1987. ACM Press.

[7] C. A. Ellis and S. J. Gibbs. Concurrency Control in Groupware Systems. In *Proceedings of the ACM SIGMOD Conference on the Management of Data - SIGMOD'89*, pages 399–407, Portland, Oregon, USA, May 1989. ACM Press.

[8] P. T. Eugster, R. Guerraoui, A.-M. Kermarrec, and L. Massoulié. Epidemic Information Dissemination in Distributed Systems. *IEEE Computer*, 37(5):60–67, May 2004.

[9] J. Ferrié, N. Vidot, and M. Cart. Concurrent Undo Operations in Collaborative Environments Using Operational Transformation. In *Proceedings on the Conference on Cooperative Information Systems - CoopIS 2004*, volume 3290 of *Lecture Notes in Computer Science*, pages 155–173, Agia Napa, Cyprus, October 2004. Springer Verlag.

[10] N. Gu, J. Yang, and Q. Zhang. Consistency Maintenance Based on the Mark & Retrace Technique in Groupware Systems. In *Proceedings of the ACM*

SIGGROUP Conference on Supporting Group Work - GROUP 2005, pages 264–273, Sanibel Island, Florida, USA, November 2005. ACM Press.

[11] A.-M. Kermarrec, A. Rowstron, M. Shapiro, and P. Druschel. The IceCube Approach to the Reconciliation of Divergent Replicas. In *Proceedings of the ACM Symposium on Principles of Distributed Computing - PODC 2001*, pages 210–218, Newport, Rhode Island, USA, August 2001. ACM Press.

[12] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Providing High Availability Using Lazy Replication. *ACM Transactions on Computer Systems*, 10(4):360–391, November 1992.

[13] L. Lamport. Times, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[14] D. Li and R. Li. Preserving Operation Effects Relation in Group Editors. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work - CSCW 2004*, pages 457–466, Chicago, Illinois, USA, November 2004. ACM Press.

[15] R. Li and D. Li. A Landmark-Based Transformation Approach to Concurrency Control in Group Editors. In *Proceedings of the ACM SIGGROUP Conference on Supporting Group Work - GROUP 2005*, pages 284–293, Sanibel Island, Florida, USA, November 2005. ACM Press.

[16] R. Li and D. Li. Commutativity-Based Concurrency Control in Groupware. In *Proceedings of the IEEE Conference on Collaborative Computing: Networking, Applications and Worksharing - CollaborateCom 2005*, San Jose, California, USA, December 2005. IEEE Computer Society.

[17] F. Mattern. Virtual Time and Global States of Distributed Systems. In *Proceedings of the International Workshop on Parallel and Distributed Algorithms*, pages 215–226, Château de Bonas, France, October 1989. Elsevier B.V.

[18] G. Oster, P. Urso, P. Molli, and A. Imine. Real-Time Group Editors Without Operational Transformation. Research Report RR-5580, LORIA – INRIA Lorraine, May 2005.

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

[20] H. Shen and C. Sun. Flexible Merging for Asynchronous Collaborative Systems. In *Proceeding of the Conference on Cooperative Information Systems - CoopIS 2002*, volume 2519 of *Lecture Notes in Computer Science*, pages 304–321, Irvine, California, USA, November 2002. Springer-Verlag.

[21] 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, Arizona, USA, November 1997. ACM Press.

[22] C. Sun. Undo as Concurrent Inverse in Group Editors. *ACM Transactions on Computer-Human Interaction*, 9(4):309–361, December 2002.

[23] 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, Washington, USA, November 1998. ACM Press.

[24] 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*, 5(1):63–108, March 1998.

[25] 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 ACM Symposium on Operating Systems Principles - SOSP'95*, pages 172–182, Copper Mountain, Colorado, United States, December 1995. ACM Press.

[26] R. H. Thomas. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *ACM Transactions on Database Systems*, 4(2):180–209, June 1979.

[27] N. Vidot, M. Cart, J. Ferrié, and M. Suleiman. Copies Convergence in a Distributed Real-Time Collaborative Environment. In *Proceedings of the ACM Conference on Computer-Supported Cooperative Work - CSCW 2000*, pages 171–180, Philadelphia, Pennsylvania, USA, December 2000. ACM Press.

[28] Y. Yu, P. Manolios, and L. Lamport. Model Checking TLA+ Specifications. In *Proceedings of the IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods - CHARME'99*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66, Bad Herrenalb, Germany, September 1999. Springer-Verlag.