Data Consistency for P2P Collaborative Editing
Gérald Oster, Pascal Urso, Pascal Molli, Abdessamad Imine

To cite this version:

HAL Id: inria-00108523
https://hal.inria.fr/inria-00108523
Submitted on 6 Nov 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
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 encyclopaedia 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 clock is associated to each operation. A vector clock 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
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 properties [14]. All operations are executed on a state where they are legal. The execution of an operation is the effect observed on 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.

Operations as defined have also precondition on their execution. The execution of \( \text{del}(e) \) requires \( e \), or its tombstone, to be present. The execution of \( \text{ins}(a \prec e \prec b) \) 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.

![Figure 1: Sets of primitive operations](image1)

![](image1)

Operations as defined have also precondition on their execution. The execution of \( \text{del}(e) \) requires \( e \), or its tombstone, to be present. The execution of \( \text{ins}(a \prec e \prec b) \) 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.

![Figure 2: Partial orderings](image2)

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 [op1, op2, op3] at site 1, must give the same result as [op1, op3, op2] at site 3 and [op2, op1, op3] 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, a topological sort is used to find a linear extension. Thus, a total order based structure is set to (< id, α, v, id) where

- id is the identifier of the character.
- α is the alphabetical value of the effect character.
- v ∈ {True, False} indicates if the character is visible,
- idcp is the identifier of the previous W-character of c.
- idcn 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 CP(c). The next W-character of c is denoted CN(c).

Each site s has a unique identifier numSite, a logical clock Hs, a sequence stringA of W-characters and a pool of pending operations poolA. 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, Hs).

Each time a W-character is generated at site s, the local clock Hs is incremented. Since numSite is unique, this pair forms a unique identifier for a character.

This consistency model is nearly equivalent to the model from Sun et al. [24]. The only difference concerns the definition of the relation →. 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 op2 causally depends on another operation op1 if op2 is generated at one site after op1 has been executed at that site. If this is a natural way of expressing causality, it only specifies a potential causal dependency between operations. op2 might not depend on the execution effect of op1. For instance, consider a file system on which two operations are performed op1 = createFile(a) next op2 = createFile(b). There is no causal dependence between these operations since op2 could be executed before op1. But, using the Lamport’s happened-before relation, op1 precedes op2, and this ordering has to be preserved at every site. On the contrary, in WOOT, the definition of the relation → 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, op1 could be executed before op2 at one site, and op2 before op1 at another site. But, consider the two operations op1 = createDir(d) and op2 = createFile(d/a). There exists a causal dependency since op2 requires that the directory d must exist before being executed. This dependence can be extracted from the preconditions of the operation op2.

### 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, α, v, idcp, idcn > where

- id is the identifier of the character.
- α is the alphabetical value of the effect character.
- v ∈ {True, False} indicates if the character is visible,
- idcp is the identifier of the previous W-character of c.
- idcn is the identifier of the next W-character of c.

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 Oa and Ob , if Oa → Ob , then Oa is executed before Ob at all sites.
Definition 4. A W-string is an ordered sequence of W-characters $c_0 c_1 c_2 \ldots c_n c_e$ where $c_0$ 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 relations represent the iteration.

Definition 5. Let $a$ and $b$ be two W-characters. $a \prec b$ if and only if, there exists a set of characters $\{c_0, c_1, \ldots c_i\}$ such that $a = c_0$, $b = c_i$ and $CN(c_j) = c_{j+1}$ or $c_j = CP(c_{j+1})$ 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 identifiers.

Definition 6. Let $a$ and $b$ be two W-characters with their respective identifiers $(ns_a, ng_a)$ and $(ns_b, ng_b)$. $a \prec id b$ 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 $\prec_S$ is defined as $a \prec_S b$ 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 $ins(x < a < y)$.

<table>
<thead>
<tr>
<th>GenerateIns $pos, \alpha$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H_\alpha := H_\alpha + 1$</td>
</tr>
<tr>
<td>let $c_p := ithVisible(string_s, pos)$</td>
</tr>
<tr>
<td>$c_o := ithVisible(string_s, pos+1)$</td>
</tr>
<tr>
<td>wchar := &lt; (numSite_s, $H_\alpha$), $\alpha$, $True$, $c_p.id$, $c_o.id$ &gt;</td>
</tr>
<tr>
<td>IntegrateIns (wchar, $c_p.c_o$)</td>
</tr>
<tr>
<td>broadcast $ins(wchar)$</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>GenerateDel $pos$</th>
</tr>
</thead>
<tbody>
<tr>
<td>let wchar := $ithVisible(string_s, pos)$</td>
</tr>
<tr>
<td>IntegrateDel (wchar)</td>
</tr>
<tr>
<td>broadcast $del(wchar)$</td>
</tr>
</tbody>
</table>

Reception. Sites may receive operations with unsatisfied preconditions. The $isExecutable$ function checks preconditions of an operation.

<table>
<thead>
<tr>
<th>isExecutable $(op)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>let $c := char(op)$</td>
</tr>
<tr>
<td>if $type(op) = del$ then</td>
</tr>
<tr>
<td>return contains(string_s, c)</td>
</tr>
<tr>
<td>else</td>
</tr>
</tbody>
</table>


return contains(string_, C_P(c))
and contains(string_, 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′ = ∅ then
insert(S, c, pos(S, c_n))
else
let L := c_p d0 d1 . . . d_m c_n where d0 . . . d_m are the W-char in S′ such that C_P(d_i) ≤ S c_p
and c_n ≤ 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 < 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.

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 IntegrateIns(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_0 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_3 314 c_e”.

It integrates o_2 = ins(c_0 < 2 < c_e). ‘2’ must be placed among the characters between c_0 and c_e. Thus, S′

\[
\begin{align*}
\text{site 1} & \quad \text{"ab"} \\
\text{site 2} & \quad \text{"ab"} \\
\text{site 3} & \quad \text{"ab"} \\
\end{align*}
\]

\[
\begin{align*}
op_1 & = \text{ins}(a < 1 < b) \quad \text{op}_2 = \text{ins}(a < 2 < b) \\
op_3 & = \text{ins}(a < 3 < 1) \\
op_4 & = \text{ins}(a < 4) \\
\end{align*}
\]

(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
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 ‘4’.

\[ \text{IntegrateIns} \]

\( o_1 = \text{ins}(c_b < 1 < c_e) \)

\( o_2 = \text{ins}(c_b < 2 < c_e) \)

\( o_3 = \text{ins}(c_b < 3 < 1) \)

\( o_4 = \text{ins}(1 < 4 < c_e) \)

Finally, state of Site 3 becomes “c_b3124c_e”.

\[ \text{Integration at Site 1} \]

Site 1 generates \( o_1 \) to obtain state “c_b1c_e”. Whatever arrival orders of \( o_2, o_3 \) and \( o_4 \), WOOT computes c_b3124c_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 \( L = c_pc_0 \). If \( L = c_pc_0 \), every character in this \( S' \) has its predecessor or successor in \( S' \). At least, the first integrated character between \( c_p \) and \( c_0 \) 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 \( < \).

Proof. Generation of an operation does not modify relation \( < \). This relation is only modified through \( \text{IntegrateIns} \).

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 \( < \).

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 \( \text{IntegrateIns} \) is 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.
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 they are generated, while remote operations must be transformed regarding concurrently executed operations. Algorithms based on this approach can be classified mainly into 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 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_i, H_i) where 0 ≤ 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 Integral procedure. Its role is to insert a character between its previous cj and next character cn, while arranging it among characters that have been concurrently inserted or deleted between cj and cn. 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-
commit and so on. Thus, convergence will be achieved in its changes. During round 2 all other sites have to update change, during round 1 only one site will be able to publish characters.

Anyway, we need an algorithm to garbage old deleted cuted log. It requires to keep tombstones at least for a long time. So, the space overhead of tombstones grows indefinitely. We can use an expiration period for garbaging characters. So, the space overhead of tombstones grows in-definitely. We can use an expiration period for garbaging 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


