# Consistency in Real-time Collaborative Editing Systems Based on Partial Persistent Sequences

Qinyi Wu
CERCS, Georgia
Institute of Technology
Atlanta, USA
qxw@cc.gatech.edu

Calton Pu
CERCS, Georgia
Institute of Technology
Atlanta, USA
calton@cc.gatech.edu

## ABSTRACT

In real-time collaborative editing systems, users create a shared document by issuing insert, delete, and undo operations on their local replica anytime and anywhere. Data consistency issues arise due to concurrent editing conflicts. Traditional consistency models put restrictions on editing operations updating different portions of a shared document, which is unnecessary for many editing scenarios, and cause their view synchronization strategies to become less efficient. To address these problems, we propose a new data consistency model that preserves convergence and synchronizes editing operations only when they access overlapped or contiguous characters. Our view synchronization strategy is implemented by a novel data structure–partial persistent sequence. A partial persistent sequence is an ordered set of items indexed by persistent and unique position identifiers. It captures data dependencies of editing operations and encodes them in a way that they can be correctly executed on any document replica. As a result, a simple and efficient view synchronization strategy can be implemented.

## Keywords

collaborative editing system, data consistency model, persistent data structure

## 1. INTRODUCTION

With the technological advance in real-time collaborative editors (such as SubEthaEdit [2], Coward [34] and TeNDaX [15]), more and more documents are coauthored. The collaboration includes the process of preparing a presentation with colleagues [1] to the more structured sharing of documents in business process management [13]. As a result, a document is no longer shared as an atomic unit. Instead, it is processed at a finer granularity such as sections, paragraphs, or even sentences. To maintain data consistency, this type of communication must be carefully coordinated due to concurrent editing conflicts.

This paper focuses on a data-dependency preservation (DDP) consistency model (Section 3) and its corresponding view synchronization strategy for collaborative editing systems. In the targeted editing scenario, a shared document is replicated at multiple sites connected by communication networks. The user at each site can update his/her local replica by issuing insert, delete, and undo operations anytime and anywhere. Local updates are executed immediately for fast response time. The underlying editing system is responsible for view synchronization among all replicas by broadcasting local updates to other sites.

Several data consistency models for collaborative editing systems have been proposed in the literature [10, 17, 20, 29]. They require 1) all document replicas converge to the same view; and 2) execution of editing operations conform to the happen-before precedent order defined by Lamport's Clock Condition [14]. These earlier data consistency models are conservative in the sense that the happen-before relation captures potential causality, not necessarily real causality. For example, if two accountants work on the financial reports of different departments through a shared spreadsheet, there is no real causality between the editing operations from these two accountants. We are aware that there are scenarios that people work closely on a shared document. For example, the web page for the 2008 Olympic Game at Wikipedia was updated more than a thousand times during August 2008 [33]. In these scenarios, the happen-before relationships may closely mimic real causalities. However, the temporary violation of cause-effect order is generally regarded acceptable in these non-mission-critical scenarios as long as the final version contains all the updates.

View synchronization strategies have been proposed in the past to resolve editing conflicts such as locking, transaction mechanisms, and turn-taking protocols [11]. These approaches have limited use in real-time collaborative editing scenarios due to high overhead in lock management and restricted collaboration. A well-accepted approach is called Operational Transformation (OT) [10] due to its non-blocking feature in view synchronization. The limitation of OT is that each operation has to be transformed with all its concurrent operations and all operations that happen after it if they reach other sites out of order. Li et al. [16] analyzed one of OT algorithms and shows that when an editing operation is broadcast to a remote site, the complexity of its transformation cost is $O(n^2)$ on average and $O(n^3)$ in the worst case, where $n$ is the number of editing operation in the history.

To address the above problems, the first contribution of this paper is the DDP consistency model that requires *convergence* and preserves *data-dependency* precedent order on execution of editing operations. The data-dependency precedence preservation property is more relaxed than the happen-before precedence preservation property (as defined in OT). Therefore, the DDP consistency model allows higher concurrency and can be enforced more efficiently (see below). Concretely, a data dependency is defined on a pair of operations if they modify overlapped or contiguous characters. Users working on different portions of the document can collaborate efficiently without any interference.

The second contribution of the paper is an efficient and simple method to enforce the DDP consistency model, based on a view synchronization strategy that uses a novel data structure–*partial persistent sequence* (PPS). A PPS is an ordered set of items indexed by persistent and unique position identifiers. This new data structure is able to create a global position identifier for every character, which carries enough information to directly locate its right position in any document replica. With the help of this data structure, it is also straightforward to construct an undo operation to cancel the effect of any editing operation. This greatly simplifies our view synchronization strategy for shared documents edited by insert, delete, and undo operations. Furthermore, the position identifiers are totally ordered, we can maintain a PPS in a search tree (such as a B-tree). In section 5.1, we show that the computational complexity of our view synchronization strategy is bounded by $O(m)$, where $m$ is the number of operations that are data-dependency related in an editing history. Finally. the PPS data structure allows us to easily capture the data dependency between any pair of editing operations that modify overlapped or contiguous characters. As a result, editing conflicts due to shared access can be efficiently detected and resolved.

**Roadmap**. The paper is organized as follows. Section 2 first gives a brief background for collaborative editing systems. Then we introduce the notion of logical view and physical view of a document. Based on these two levels of view, we describe two types of logical view synchronization approaches and explain why we choose one of them. Section 3 defines the DDP consistency model. Section 4 first gives a formal definition for PPSs, and then shows how this new data structure maintains the two levels of view of a document and its properties. In Section 5, we show the application of PPSs to view synchronization and undo handling. Section 6 covers related work. We conclude in Section 7.

## 2. COLLABORATIVE EDITING SYSTEMS

A collaborative editing system (CES) consists of a set of *users* editing a shared document, located on *sites* connected by communication networks. For simplicity, each site has one user. The users collaborate in a real-time fashion to create a shared document. Each site maintains a replica of the shared document and executes four types of activities: 1) generate local insert, delete, and undo operations; 2) broadcast local updates to other sites; 3) receive operations from other sites; 4) execute operations (either local or remote update).

A collaborative editing scenario is shown in Figure 1. Three users work on a shared document with initial content "*abcd*". The vertical lines represent the elapse of time. Circles represent locally generated operations, which are executed immediately. Arrows represent the propagation of local operations to other sites. In this scenario, $user_1$ adds '$e$' at offset 0, and simultaneously $user_2$ adds '$f$' at offset 2. After executing $o_1$ and $o_2$'s updates, $user_3$'s document replica is modified to "$eabfcd$". $o_3$ deletes '$c$' at offset 4. The three operations are executed in different order at each site. As a result, appropriate view synchronization strategies must be used to maintain a consistent view among users.

CESs must also support undo operations so that users can reverse their recent changes. We handle undo operations through the notion of compensation, which does not remove the footprint of an editing operation from the editing history. Instead, a new operation is created to cancel its effect. The advantage of this compensated-undo approach is that an undo operation is treated like a normal insert or delete operation. Therefore, the view synchronization strategy does not need to be changed. We will devote a separate section for discussing the compensated undo approach in Section 5.2.

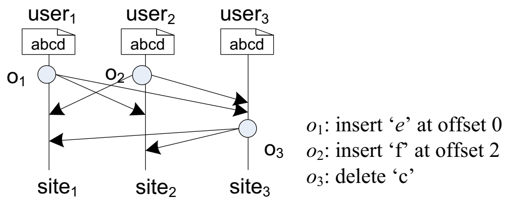

**Figure 1: A collaborative editing scenario**

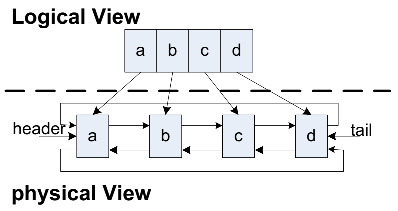

**Figure 2: Two views of a document**

### 2.1 Two Levels of View on Documents

From a user's perspective, a document consists of a sequence of characters. If a new character is inserted, a portion of the sequence will be shifted right to vacate the space for the new character. Correspondingly, if a character is deleted, a portion of the sequence will be shifted left to reclaim the space. On the other hand, the underlying editing system keeps the characters of the document in a selected data structure, such as an array and a linked list. Figure 2 illustrates the idea by a linked list. We call the sequence data structure from the user's perspective *logical view* and the implementation data structure from the editing system's perspective *physical view*. In the rest of the paper, we use the term *document* as abbreviation for the logical view of a document and the term *physical document* as abbreviation for the physical view of a document.

At the logical view, a document is defined by a sequence of characters $E = \langle c_i \in \Sigma^c, 1 \leq i \leq n, n \in \mathbb{N}\rangle$, where $\Sigma^c$ is the alphabet of the document. $\langle\rangle$ denotes an empty sequence of characters, $E[i]$ the $i$-th element of $E$, $E[i,j]$ the subsequence $\langle c_i, c_{i+1}, ..., c_j\rangle$, $|E|$ the length of $E$.

When a document is first created, it is initialized to an empty sequence of characters $E_0 = \langle\rangle$. Each INSERT or DELETE operation transforms the document from $E_i$ to $E_{i+1}$. We use $E_k$ to denote the version of $E$ as transformed by a sequence of $INSERT$ and $DELETE$ operations of cardinality $k$. An INSERT operation adds a new character into the sequence. A DELETE operation removes a character from the sequence. The sequence of operations $o_0o_1...o_n$ that creates the history of document is its *logical-view editing history.* These two operations are defined as follows:

- $INSERT(p, x)$: $p \in \mathbb{N}$, $x \in \Sigma^c$. It adds the character $x$ at the $p$-th position in $E_i$. This operation updates

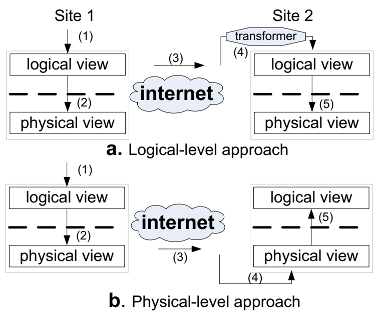

**Figure 3: Two types of approaches in logical view synchronization**

$E_i$ to $E_{i+1}$ such that

- $E_{i+1} = E_i[1, p-1] \circ x \circ E_i[p, n]$, where $n = |E_i|$. $\circ$ denotes concatenation of sequence.

- $DELETE(p)$: $p \in \mathbb{N}$. It removes the character at the $p$-th position in $E_i$. This operation updates $E_i$ to $E_{i+1}$ such that
  - $E_{i+1} = E_i[0, p-1] \circ E_i[p+1, n]$, where $n = |E_i|$.

As suggested in Figure 2, the physical view reflects how a document is really implemented in the editing system and stored on disk. Clearly, editing operations defined on the logical view do not necessarily match the operations defined on its physical view. For example, an insertion at the logical view will be translated to an operation of pointer redirection at the physical view if the linked-list data structure is used. An operation defined on the logical view is said to be in its *logical form*. An operation defined on the physical view is said to be in its *physical form*. Even though a document uses the sequence abstraction to represent its logical view, there are many choices of data structures to represent its physical view such as linked-lists, arrays, and piece tables [6]. Depending on the chosen data structure, we will have a different set of rules for mapping editing operations between the logical view and the physical view. We will give an example for the mapping after we introduce PPSs in Section 4.

## 2.2 Logical View Synchronization

In a CES, users update a document by issuing INSERT and DELETE operations. The editing system is responsible for synchronizing the logical view of all document replicas in a timely fashion. There are two types of approaches to do the synchronization.

- The *logical-level* approach is illustrated by Figure 3.a. An operation is generated on its logical view in step (1). The operation is mapped to its physical form in step (2). Meanwhile, it is broadcast to $site_2$ in its logical form in step (3). After the arrival at $site_2$, the logical view may not be the same as the one when it is generated at $site_1$. Therefore, in step (4) the remote update operation has to be transformed by the *transformer* in a way such that it can correctly locate its right position. Finally, the operation is merged into its physical view in step (5). Since view update operations are exchanged in their logical forms, we call this *logical-level* approach.

- The *physical-level* approach is illustrated by Figure 3.b. The local operation follows the same path in the step (1) and step (2) of the logical-level approach. However, the operation's mapped form is broadcast to $site_2$ in step (3). After receiving this remote update, $site_2$ updates its physical view in step (4) and appends it to its editing history. Finally, the system refreshes its logical view at step (5). Since view update operations are exchanged in their physical forms, we call this *physical-level* approach.

In the logical-level approach, an editing operation defined at the logical view does not have a global position identifier. When it is broadcast to other sites, it is possible (when under contention) that its position identifier becomes invalid. For example, in Figure 1, when $o_2$ is generated, it is in the form of $INSERT(2, f)$. The intended position is the one between '*b*' and '*c*'. However, after broadcast to $site_1$, the offset 2 is no longer the originally intended position. Instead, the offset has to be adjusted based on the logical-view editing history at $site_1$, which consists of $INSERT(0, e)$. We can shift right the offset of $o_2$ right by 1 to decide the right offset. Similarly, if there is a DELETE in the past history, we have to shift left the offset of $o_2$ by 1. This is the basic idea behind the OT approach. The transformation could become fairly complicated when there are multiple sites involved in the editing session. In a well-known example, suppose that three users start to edit a document with initial content "*abc*". The following operations are generated simultaneously. $user_1$ inserts '*x*' between '*a*' and '*b*'; $user_2$ inserts '*y*' between '*b*' and '*c*'; $user_3$ deletes '*b*'. The expected correct final version would be "*axyc*" because we have the relative order of '*a*', '*x*', '*b*' and '*b*', '*y*', '*c*'. However, if not processed appropriately, a view synchronization algorithm may produce "*ayxc*" at all the sites [29]. Due to the necessity of transformation, the logical-level approach has three limitations:

- The transformation procedure is computation expensive because each editing operation has to be transformed against all concurrent operations and the operations that happened after it if they arrive at other sites out of order. For example, in Figure 1, $o_3$ have to be transformed against $o_1$ and $o_2$ at $site_1$.

- Each operation has to encode some type of data structure to maintain the timing information in order to detect concurrency and decide the right transformation path. The most used data structure is version vector [22]. Both its size and its maintenance are concerns, especially in mobile collaborative editing in which users can come and go during an editing session. Even though we can apply some dynamic space management to shrink its size [24], it adds significant complexity during implementation.

- It is hard to undo an operation. First, an undo operation has to go through similar transformation procedure. Second, additional efforts have to be taken in order to identify the do-undo pair in the history because we can not determine the undo form of an editing operation by looking at its logical form directly. For example, in Figure 1, after executing $o_2$ and $o_1$ in this order at $site_2$, the logical view is updated to "*eabfcd*". If $user_2$ wants to undo its own last update $o_2$, we can not simply inverse its effect by $DELETE(2)$ because 2 does not point to '$f$' any more. Therefore, we either need to analyze the history to figure out the right position parameter for undoing $o_2$, or maintain additional information to identify the do-undo pair by creating some types of identifiers [28].

In the following sections, we show that by using appropriate data structures to implement the physical view, the physical-level approach to logical view synchronization can bypass the limitations described above for the logical level approach. If designed properly, editing operations defined on the physical view can carry enough information such that each site can directly locate its right position, avoiding the cost for transforming operations and the cost for carrying timing information. Since the physical-level approach to synchronization requires a data consistency model that is independent of the underlying physical view, we define such a data consistency model in Section 3, followed by the introduction of the *PPS* data structure in Section 4.

## 3. DDP CONSISTENCY MODEL

A data consistency model defines correctness criteria for execution of CESs from the users' perspectives. Therefore, it is defined on the logical view of a shared document. Our consistency model, called DDP, consists of two properties:

- *convergence property*: it states that a shared document replicas converge to the same logical view after executing replica updates if no new operation arrives, and if all nodes are connected. The resulting view contains all generated updates.
- *data-dependency precedence preservation property*: it states that if one operation $o_j$ data-depends on $o_i$, then $o_i$ should be executed before $o_j$ at all the sites. We say $o_j$ data-depends on $o_i$, denoted as $o_i \rightarrow o_j$, if
  - $o_j$ deletes the character inserted by $o_i$.
  - $o_j$ inserts a character contiguous to the character inserted by $o_i$.

The convergence property requires that all document replicas converge to the same value. This is different from *semantic consistency*, which demands that the converged value is also meaningful in the application context [7]. Semantic consistency requires domain specific knowledge, which is hard to verify by relying on pure system approaches. An example is two users trying to fix a grammar error in the sentence "*There should be student*" at the same time [29]. One inserts '*a*' after "*be*" while the other inserts a '*s*' after "*student*". After the modification, the sentence becomes "*There should be a students*", which is not semantically correct. Therefore, current CESs enforce converged views and leave semantic consistency to the interpretation of end users.

The data-dependency precedence preservation property guarantees that editing operations are executed within their surrounding context. For example, if a user writes "*a day*", then inserts a "*nice*" between these two words, he expects the execution of the insertion for the phrase "*a day*" is executed first, the insertion for the word "*nice*" second. Here the "*a day*" is the context for "*nice*". Another example is if the user inserts a word first, then deletes it, he also expects the execution of the insertion first and the deletion second at all the sites. Editing operations not data-dependency related are allowed to be executed in any order. Therefore, users working on different portions of the document can collaborate efficiently without any interference.

## 4. PARTIAL PERSISTENT SEQUENCES

A PPS is an ordered set of items indexed by persistent and unique position identifiers that are rational numbers. For example, the position identifiers for the PPS "*abc*" could be 0, 0.5 and 1 respectively. We call the position identifiers in a PPS *position stamps* to differentiate them from the indexes in traditional sequences.

### 4.1 Data Model

A PPS is defined by a pair $(S, M)$, where

- $S$: a set of unique rational numbers, which are called position stamps. $S = \{s_i \in \mathbb{Q}, 1 \leq i \leq n, n \in \mathbb{N}\}$.
- $M$: a mapping function from $S$ to $\Sigma$, where $\Sigma$ is a finite set of characters. $\Sigma$ contains a null character $\phi$ that is different from any other characters allowed in user applications. Let $\Sigma^c = \Sigma - \phi$.

The position stamps in $S$ are totally ordered by *less than* $<$ defined on $\mathbb{Q}$. For $s_i \in S$, we use $s_{i+1}$ to denote the next position stamp in $S$ such that $s_i < s_{i+1}$ and $\neg(\exists s_x \in S, s_i < s_x < s_{i+1})$. Similarly, we use $s_{i-1}$ to denote the previous position stamp of $s_i$ in $S$. We use $S[s_i, s_j] = \{s_x | s_x \in S, s_i \leq s_x \leq s_j\}$ to denote the set of position stamps that fall within the range of $s_i$ and $s_j$ (inclusive).

A physical document is defined by a PPS. The history of physical view is defined by a set of physical documents $\{(S_k, M_k), 0 \leq k \leq n\}$, where each physical document in the history is called a *version* of $(S, M)$. When a physical document is first created, its initial version is an empty PPS $S_0 = \{0, 1\}$, $M_0 = \{0 \mapsto \phi, 1 \mapsto \phi\}$. A physical document is updated by $ADD$ and $HIDE$ operations. An ADD operation adds a new position stamp into $S_k$ and a new mapping into $M_k$. A HIDE operation changes the mapping in $M_k$. Each ADD and HIDE operation $(\tilde{o}_i)$ transforms $(S_k, M_k)$ to $(S_{k+1}, M_{k+1})$. The sequence of operations $\tilde{o}_0\tilde{o}_1...\tilde{o}_n$ that creates the history of physical document is its *physical-view editing history*. These two operations are defined as follows:

- $ADD(s_i, s_{i+1}, x)$: $s_i, s_{i+1} \in S_k$, $x \in \Sigma^c$. It adds the character $x$ between the character indexed by $s_i$ and the character indexed by $s_{i+1}$. Let $s_{new} \in \mathbb{Q}$ be a position stamp that satisfies the constraint of $s_i < s_{new} < s_{i+1}$. It updates $(S_k, M_k)$ to $(S_{k+1}, M_{k+1})$, where
  - $S_{k+1} = S_k \bigcup \{s_{new}\}$
  - $M_{k+1} = M_k \bigcup \{s_{new} \mapsto x\}$
- $HIDE(s_i)$: $s_i \in S_k$. It changes the mapping of $s_i$ from its old value $x$ to the null character $\phi$. It updates $(S_k, M_k)$ to $(S_{k+1}, M_{k+1})$, where
  - $S_{k+1} = S_k$
  - $M_{k+1} = M_k - \{s_i \mapsto x\} \bigcup \{s_i \mapsto \phi\}$.

From the above definitions, it can be seen that a HIDE operation does not change the set of position stamps in a

PPS. (This is similar to the notion of "tombstone" in distributed systems to solve the ambiguity on update/delete operations [27]). But an ADD operation will add a new element, $s_{new}$, into the set. The value of $s_{new}$ must fall within the range between $s_k$ and $s_{k+1}$ defined in $S_k$. This is important in order to maintain the uniqueness of each position stamp and the right position of the newly inserted character in $S_{k+1}$. The algorithm that computes the value of $s_{new}$ is called an *encoding scheme*. Partial persistent sequences leave the freedom of choosing a particular encoding scheme. For example, we can compute $s_{new}$ in dyadic rational numbers, which halving the interval between $s_i$ and $s_{i+1}$ into two, or in Farey rational numbers, which choose mediant of $s_i$ and $s_{i+1}$ [31]. We assume that all sites use the same encoding scheme in the rest of the paper.

*Example* 1. Suppose we choose dyadic rational numbers as the encoding scheme. For a newly created PPS, we have $S_0 = \{0, 1\}, M_0 = \{0 \mapsto \phi, 1 \mapsto \phi\}$. The operation $ADD(0, 1, a)$ updates the PPS to $S_1 = \{0, 0.5, 1\}, M_1 = \{0 \mapsto \phi, 1 \mapsto \phi, 0.5 \mapsto a\}$. Then the operation $HIDE(0.5)$ further updates it to $S_2 = \{0, 0.5, 1\}, M_2 = \{0 \mapsto \phi, 1 \mapsto \phi, 0.5 \mapsto \phi\}$.

## 4.2 Mapping Between Two Levels of View

In this section, we show how to establish the mapping between these two levels of view for documents implemented by PPSs. The mapping from the physical view to the logical view is defined by a $PIECE$ operation that returns the sequence of characters whose position stamps are not mapped to $\phi$. Even though a PPS keeps the position stamps of all the characters inserted in its physical-view editing history, a user only works on those characters whose position stamps are not mapped to $\phi$ by the HIDE operation. We call the characters returned by PIECE *visible characters* of the PPS. The concatenation of a character sequence *str* with $\phi$ is still a character sequence such that *str* $\circ \phi =$ *str*. The concatenation of two null characters is still the null character such that $\phi \circ \phi = \phi$. PIECE is defined as below:

- $PIECE(S_k[s_i, s_j], M_k) = M_k(s_i) \circ M_k(s_{i+1}) \circ, ..., \circ M_k(s_j)$, where $s_i, s_{i+1}..., s_j \in S_k$.

$PIECE(S_k[s_i, s_j], M_k)$ returns the sequence of visible characters whose position stamps falls with the range of $s_i$ and $s_j$ (inclusive). $PIECE(S_k[s_1, s_n], M_k)$, where $n = |S_k|$, returns the sequence that contains all visible characters. For brevity, we use $PIECE(S_k, M_k)$ to denote $PIECE(S_k[s_1, s_n], M_k)$ without writing the range explicitly. If $PIECE(S_k [s_i, s_j], M_k) = \phi$, then it denote the empty sequence $\langle\rangle$ from a user's point of view.

Next we show how to map the logical view to the physical view. As discussed in Section 2.1, users edit a document by issuing editing operations defined at its logical view, which need to be mapped into the forms on its physical view. The mapping rules are defined below:

- Rule 1: an $INSERT(p, x)$ on $E_k$ is mapped to $ADD(s_{i-1}, s_i, x)$ on $(S_k, M_k)$, where $s_i$ is the smallest position stamp satisfying $p = |PIECE(S_k[s_1, s_i], M_k)|$ if $E_k \neq \langle\rangle$ or $s_i = 1$ if $E_k = \langle\rangle$.

- Rule 2: a $DELETE(p)$ on $E_k$ is mapped to $HIDE(s_i)$ on $(S_k, M_k)$, where $s_i$ is the smallest position stamp satisfying $p = |PIECE(S_k[s_1, s_i], M_k)|$.

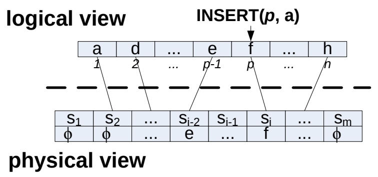

**Figure 4: Mapping between two levels of view**

Figure 4 gives an example for Rule 1. When a user issues an operation at offset $p$, Rule 1 locates the smallest position stamp $s_i$ that satisfies $p = |PIECE(S_k[s_1, s_i], M_k)|$. This constraint establishes the correspondence between the character indexed by $s_i$ in the physical view and its counterpart indexed at $p$ on the logical view. An exception is when $E_k = \langle\rangle$. In this case, we require $s_i$ to be the rightmost position stamp, which is 1.

**Lemma 1.** Given the initial document version $E_0 = \langle\rangle$ and the initial physical document version $S_0 = \{0, 1\}$, $M_0 = \{0 \mapsto \phi, 1 \mapsto \phi\}$, let $H = o_0 o_1 ... o_n$ be the logical-view editing history and $\tilde{H} = \tilde{o}_0 \tilde{o}_1 ... \tilde{o}_n$ be the physical-view editing history, obtained by applying Rule 1 and Rule 2. We have $E_n = PIECE(S_n, M_n)$.

Proof by induction:

1. Upon initialization, $E_0 = \langle\rangle$, $PIECE(S_0, M_0) = M(0) \circ M(1) = \phi \circ \phi = \phi$.

2. Assume the lemma holds for the $k$-th update $o_k$, such that $E_k = PIECE(S_k, M_k)$

3. For the $k+1$-th update,

   - assume $o_{k+1} = INSERT(p, x)$ and its mapped form $\tilde{o}_{k+1} = ADD(s_{i-1}, s_i, x)$ by applying Rule 1. Based on the definition of INSERT in Section 2.1, we have $E_{k+1} = E_k[0, p-1] \circ x \circ E_k[p, n]$, where $n = |E_k|$. On the other hand, after applying $ADD(s_{i-1}, s_i, x)$ to the physical view $(S_k, M_k)$, we have $PIECE(S_{k+1}, M_{k+1}) = M_{k+1}(s_1) \circ ... \circ M_{k+1}(s_{i-1}) \circ M_{k+1}(s_{new}) \circ M_{k+1}(s_i) \circ ... \circ M_{k+1}(s_m)$, where $m = |S_k|$. Based on the definition of ADD in Section 4.1, we know that $\tilde{o}_{k+1}$ does not change the mappings in $M_k$. Therefore, we get $PIECE(S_{k+1}, \quad M_{k+1}) = M_k(s_1) \circ ... \circ M_k(s_{i-1}) \circ M_k(s_{new}) \circ M_k(s_i) \circ ... \circ M_k(s_m)$. Based on the assumption of $E_k = PIECE\ (S_k, M_k)$ and $p = |PIECE(s_1, si)|$, we know that $E[1, p-1] = M_k(s_1) \circ ... \circ M_k(s_{i-1})$ and $E[p, n] = M_k(s_i) \circ ... \circ M_k(s_m)$. Therefore, $PIECE(S_{k+1}, M_{k+1}) = E[1, p-1] \circ x \circ E[p, n] = E_{k+1}$.

   - assume $o_{k+1} = DELETE(p)$ and its mapped form $\tilde{o}_{k+1} = HIDE(s_i)$ by applying Rule 2. Based on the definition of DELETE in Section 2.1, we have $E_{k+1} = E_k[0, p-1] \circ E_k[p+1, n]$, where $n = |E_k|$. On the other hand, after applying $HIDE(s_i)$ to the physical view $(S_k, M_k)$, we have $PIECE(S_{k+1}, M_{k+1}) = M_{k+1}(s_1) \circ ... \circ M_{k+1}(s_{i-1}) \circ M_{k+1}(s_i) \circ M_{k+1}(s_{i+1}) \circ ... \circ M_{k+1}(s_m)$, where $m = |S_k|$. Based on the definition of HIDE in Section 4.1, we know that $\tilde{o}_{k+1}$ does not change the mapping of any position stamp in $M_k$ except $s_i$. Therefore, we get $PIECE(S_{k+1}, M_{k+1}) = M_k(s_1) \circ ... \circ M_k(s_{i-1}) \circ \phi \circ M_k(s_{i+1}) \circ ... \circ M_k(s_m) = M_k(s_1) \circ ... \circ M_k(s_{i-1}) \circ M_k(s_{i+1}) \circ ... \circ M_k(s_m)$. Since $E_k = PIECE(S_k, M_k)$ and $p = |PIECE(s_1, si)|$, we know that $E[1, p-1] = M_k(s_1) \circ ... \circ M_k(s_{i-1})$ and $E[p+1, n] = M_k(s_{i+1}) \circ ... \circ M_k(s_m)$. Therefore, $PIECE(S_{k+1},\ M_{k+1}) = E[1, p-1] \circ x \circ E[p+1, n] = E_{k+1}$.

The above lemma guarantees that whenever the document is updated by an editing history defined on its logical view, the mapping rules will correctly map the operations to their physical forms such that the version of the PPS produced by the mapped history will maintain the same view as the one from users' perspective.

## 4.3 Global Uniqueness of Position Stamps

Based on the way we create a new position stamp for a newly added character, each character has a unique position stamp. However, this uniqueness property can be violated if users happen to simultaneously modify the same position.

*Example* 2. Given a PPS with $P_0 = \{0, 1\}$ and $M_0 = \{0 \mapsto \phi, 1 \mapsto \phi\}$ at two sites, $site_1$ executes $ADD(0, 1, a)$, while $site_2$ executes $ADD(0, 1, b)$ simultaneously. If we use dyadic fraction encoding scheme by halving the interval, both '$a$' and '$b$' would be assigned 0.5.

The global uniqueness of position stamps can be resumed if we make some assumption and slightly modify the encoding scheme for single-threaded editing. We assume that each site is identified by a unique identifier and that the number of sites involved in a CES is bounded by an integer. Depending on its value, necessary decimal digits will be appended to each position stamp. For instance, in example 2, if the site identifier for $site_1$ is 1 and for $site_2$ is 2, the position stamp for '$a$' would be 0.51, for '$b$' would be 0.52. Both position stamps still fall within the range of 0 and 1. The tie is broken by ordering the numbers based on their site identifiers. Under normal editing scenarios, the number of users are relatively small. If we use two decimal digits to represent site identifier, this modified encoding scheme can support up to a hundred users already.

We informally sketch the proof for the global uniqueness of position stamps under this modified encoding scheme. Position stamps generated at different sites are always differentiated by their site identifier. Position stamps generated at the same site are guaranteed by the original single-threaded encoding scheme. This modified encoding scheme is the one we are currently considering. There may be other options as well. In the rest of the paper, we assume the global uniqueness of position stamps without repeating this property.

## 4.4 Position Stamps Recycling

Persistent stamps are rational numbers. We could run out of precision bits for newly inserted characters. Another concern is disk space consumption because a PPS never deletes its position stamps. We address both problems by recycling position stamps at system quiescent time that no editing is detected and all generated operations are executed. When such a moment is detected, each site creates a new PPS based on the invocation result of the PIECE operation. The new PPS contains only visible characters and will reassign values to its position stamps by evening the range between 0 and 1 for visible characters. Further editing operations will be redirected to the new version. This approach is practical because writing is a complex process involving more than just sitting down and typing words. Empirical study [18, 30] shows that collaborative editing involves a large amount of time for coordination and discussion, and it normally happens within a small group of people. Therefore, there is a good chance that a quiescent moment happens before we run out of precision bits. A quiescent moment can be detected easily by maintaining a counter at each site, which counts how many local and remote operations have been executed so far. The system announces the occurrence of a quiescent moment if all counters have the same value and no local editing is detected. The system could also force a quiescent moment if the length of some position stamps exceeds a threshold. At this point, each site finishes all unprocessed remote updates and puts newly generated local updates in a temporary queue. After all the sites finishes processing on remote updates, which can be inferred by the local counters, each site creates a new PPS and starts processing operations in the temporary queue. We expect this process has a negligible impact to end users because it does not block users from local editing, only delay their updates to be propagated to other users. Another benefit of this procedure is that it cleans up invisible characters from the old PPS. Therefore, functions such as searching keyword will not be impacted by these invisible characters when scanning the physical document.

## 5. APPLICATIONS AND ADVANTAGES OF PPS

## 5.1 PPS Implementation of DDP Consistency Model

A view synchronization strategy resolves editing conflicts when users simultaneously edit the overlapped or contiguous characters. In CESs, a widely accepted design choice is that only user-requested cancellation should be taken when editing conflicts occur [11]. This is different from transaction processing in database management systems in which concurrency control algorithms (such as timestamp ordering) automatically abort some transactions to resolve conflicts. of such a strategy, the effect of victim transactions gets lost, which is regarded undesirable by end users. Therefore, the traditional view synchronization strategies in CESs [10, 17, 20, 29] adopt the following design choice:

- Keep all inserted characters if users happen to update the same position in the document at the same time. In Example 2, both $site_1$ and $site_2$ inserts a new character. Both '$a$' and '$b$' will be kept in this case.

- Remove a deleted character exactly once if users happen to delete the same character at the same time.

Our view synchronization strategy adopts this design choice and chooses the physical-level approach for the reasons discussed in Section 2.1. In the physical-level approach, a user-issued operation on the logical view will be mapped to its physical view by Rule 1-2 in Section 4.2. After the mapped operation is broadcast to a remote site, it updates its physical view, which is a PPS. The physical view then refreshes its logical view by the result of PIECE operation. In Section 4.2, we already prove that Rule 1-2 and PIECE operation correctly maintain the correspondence between these two levels of view. In this section, we show that the final versions of PPS at all the sites will converge to the same value if their physical-view editing histories preserve the data-dependency precedent order. In the rest of this section, without pointed out otherwise, all operations are defined on physical view, and all histories are physical-view editing histories.

A CES is defined by a triple $CES =< U, D, \tilde{O} >$, where

- $U$: a set of unique site identifiers. $U = \{u_i, 1 \leq i \leq n, s_i \in \mathbb{N}, n \in \mathbb{N}\}$
- $D$: a set of PPSs. $D = \{ps_i, 1 \leq i \leq n, n = |U|\}$. $ps_i$ is the PPS at $u_i$.
- $\tilde{O}$: a set of parameterized editing operations. $\tilde{O} = \{\tilde{o}_i, 1 \leq i \leq \mathbb{N}\}$, where $\tilde{o}_i$ is one of the kinds
  - $(ADD(s_i, s_{i+1}, x), u_k)$: an ADD generated by $u_k$.
  - $(HIDE(s_i), u_k)$: a HIDE generated by $u_k$.

We assume that each site must finish the execution of current operation before issuing the next one. Based on this assumption, operations generated at the same site always have different forms because each position stamp is unique and they can not issue the same editing operation twice. Furthermore, operations generated at different sites are differentiated by their $u_k$ parameter. Therefore, each operation $\tilde{o}$ has a globally unique form.

An execution of a CES at a particular site is modeled by an editing history $H = \tilde{o}_1\tilde{o}_2...\tilde{o}_n$, which is an intermixed $ADD$ and $HIDE$ operations. We use $op(H)$ to denote the set of operations in $H$ and $<_H$ to denote their ordering. Starting with the initial version $(S_0, M_0)$, we use $(S_H, M_H)$ to denote the version produced by history $H$.

Next we show that if every site executes the same set of operations in an order that preserves their data-dependencies, the PPS at each site will converge to the same value. We first define data-depends relation, and then define data-dependency preserving history.

**Definition 1.** Data Depends Relation $\rightarrow$. Given two operations $\tilde{o}_p$ and $\tilde{o}_q$, we say $\tilde{o}_q$ depends on $\tilde{o}_p$, denoted as $\tilde{o}_p \rightarrow \tilde{o}_q$, if one of the following conditions is satisfied:

1. $\tilde{o}_p = (ADD(s_i, s_{i+1}, x), u_m)$ and $\tilde{o}_q = (HIDE(s_j), u_n)$. Let $s_{new}$ be the position stamp generated for $x$ by $\tilde{o}_p$. We have $s_{new} = s_j$. In other words, $\tilde{o}_q$ maps the character inserted by $\tilde{o}_p$ to $\phi$.
2. $\tilde{o}_p = (ADD(s_i, s_{i+1}, x), u_m)$ and $\tilde{o}_q = (ADD(s_j, s_{j+1}, y), u_n)$. Let $s_{new}$ be the position stamp generated for $x$ by $\tilde{o}_p$. We have either $s_{new} = s_j$ or $s_{new} = s_{j+1}$. In other words, $\tilde{o}_q$ inserts a character next to the one inserted by $\tilde{o}_p$.
3. $\exists \tilde{o}_x, \tilde{o}_p \rightarrow \tilde{o}_x$ and $\tilde{o}_x \rightarrow \tilde{o}_q$.

**Definition 2.** Data-dependency-preserving History. A history $H = \tilde{o}_1\tilde{o}_2...\tilde{o}_n$ is said to be data-dependency-preserving if $\tilde{o}_i \rightarrow \tilde{o}_j$ then $\tilde{o}_i <_H \tilde{o}_j$.

**Definition 3.** Data-dependency Equivalence. Let $H$ and $H'$ be two histories. $H$ and $H'$ are called data-dependency equivalent, denoted as $H \approx_d H'$, if the following holds:

1. $op(H) = op(H')$, and
2. $H$ and $H'$ are data-dependency preserving.

**Theorem 1.** If $H$ and $H'$ are data-dependency equivalent, then starting with the same initial empty PPS, we have $S_H = S_{H'}$ and $M_H = M_{H'}$.
Proof

1. Prove $S_H = S_{H'}$. Based on the definition of PPS, $S$ is only updated by ADD operations. Since $op(H) = op(H')$ and each operation being unique, the set of ADD operations in $H$ is the same as the one in $H'$. Therefore, the $S_H$ and $S_{H'}$ will be expanded with the same set of position stamps. Since by definition $S$ is a set, the updating order does not matter. Therefore, we have $S_H = S_{H'}$.
2. Prove $M_H = M_{H'}$. Assume that $M_H \neq M_{H'}$. Then $\exists s_i, M_H(s_i) \neq M_{H'}(s_i)$. Since each $ADD$ operation will add a unique position stamp, $s_i$ must be added by the same $ADD$ operation. Let the ADD operation that adds $s_i$ be $\tilde{o}_i$ in $H$ and be $\tilde{o}_j$ in $H'$. Since $M_H(s_i) \neq M_{H'}(s_i)$, one of them must be changed to $\phi$ by $HIDE(s_i)$. Let the HIDE operation be $\tilde{o}_u$ in $H$ and $\tilde{o}_v$ in $H'$. Without loss of generality, let $M_H(s_i) = \phi$. Then we have $\tilde{o}_i \rightarrow \tilde{o}_u$ and $\tilde{o}_i <_H \tilde{o}_u$. Since $M_{H'}(s_i) \neq \phi$, this could only happen if $\tilde{o}_v <_{H'} \tilde{o}_j$. Thus the assumption that $H'$ is data-dependency preserving is violated.

Theorem 1 guarantees that if each site executes all updates (both local and remote) in their data-dependency precedent order, the final versions of PPSs at each site will converge to the same value. Since the data dependency can be precisely captured by position stamps encoded in editing operations, the synchronization strategy can easily maintain this precedence order.

Finally we describe our view synchronization strategy. Each site maintains two queues: *READY queue* and *WAITING queue*. The site runs two threads, one for receiving operation, and one for executing operation. When a new operation arrives, the thread for receiving operation puts it in the READY queue if it is local. If the new operation is a remote update, the thread first checks whether the operations it depends on have been processed or not by looking up the set of position stamps in the current version of its PPS. If they were there, the operation will be put in the READY queue, otherwise in the WAITING queue. Meanwhile, the thread for executing operation continuously pops a new operation from the READY queue if it is not empty. After the execution, it will check whether any operations depending on it are in the WAITING queue. If none of them exist, the thread will execute the next one in the READY queue. Otherwise, the thread will move those operations whose data-dependencies are satisfied from the WAITING queue to the READY queue.

Since persistent stamps are totally ordered, we can use a search tree B-tree to maintain a PPS by using its persistent stamp as keys. Give a physical-view editing history with length $n$, the number of position stamps in the PPS is bounded by $n$. The computational complexity for PPS's update and look-up is therefore $O(t \log_t n)$, where $t$ is the upper bound on the number of keys in a B-tree node . The en-queue and de-queue operations take $O(1)$. In our view synchronization strategy, the thread for receiving operation does one look-up in the PPS. The thread for executing operation does one B-tree key insertion for ADD operation or one

B-tree key look-up for HIDE operation, which is bounded by $O(t \log_t n)$. Besides, the executing thread also need to do a traversal for the dependency check on the WAITING queue, which takes $O(m)$ where $m$ is the number of operations that are data-dependency related. Therefore, if $t$ is big enough, the computational complexity of our view synchronization strategy is bounded by $O(m)$.

### 5.2 PPS Implementation of Compensated Undo

Undo is an essential feature in any collaborative editors [3, 23]. In this section, we explain how to use PPSs to support it so that a shared document can recover from accidentally incorrect editing and possible vandalism in loosely controlled editing environment. A single-user editing system normally un-does operations in the reverse of their chronological order. In a CES, its undo policy may choose the last operation performed by the local user, or the last operation performed so far, or any operation in the history [23]. Partial persistent sequences are powerful in their support to undo any operation. First, the data structure maintains the effect of whole editing history. It is easy to determine the undo effect of any editing operation. Second, editing operations defined on this data structure encode enough information to construct its undo operation. We use the notion of compensation to handle selective undo [32]. A compensated undo does not physical remove the footprint of an editing operation from the history. Instead, a new operation is created to cancel its effect from users' perspective.

We assume its physical-view editing history $H = \tilde{o}_0\tilde{o}_1...\tilde{o}_n$. For any editing operation $\tilde{o}_i$, its compensated undo is constructed as follows:

- if $\tilde{o}_i$ is an $ADD(s_k, s_{k+1}, x)$, its undo takes the form $HIDE\ (s_{new})$, where $s_{new}$ is the position stamp assigned to $x$. The HIDE is applied to the current version $(S_n, M_n)$ as a normal editing operation.
- if $\tilde{o}_i$ is a $HIDE(s_k)$, assume its original mapping is $M(s_k) = c$. Its undo takes the form $ADD(s'_k, s'_{k+1}, c)$, where $s'_k, s'_{k+1} \in S_n$ and $s_k = s'_k$. The ADD is applied to the current version $(S_n, M_n)$ as a normal editing operation.

The above construction guarantees that if a user inserts a character, its compensated undo will remove the character from the logical view of the document. And if the user deletes a character, its compensated undo will insert the same character at its original position in the logical view. The proof for the correctness of this approach is similar to the one for the mapping between two levels of view of a document in Section 4.2. We skip it in this paper. Note that in order to compute the undo form for a HIDE operation, the old mapping needs to be maintained. This can be easily done if we keep the old mapping value when we save the HIDE in the log. The advantage of compensated undo is that undo operations are treated like any other ordinary operations. Therefore, the view synchronization strategy does not change.

## 6. RELATED WORK

**View synchronization algorithms** Current view synchronization algorithms can be categorized into the *logical-level* approach or the *physical-level* approach. All the OT-based algorithms [10, 17, 20, 25, 29] belong to the *logical-level* approach. As stated in Section 2.2, the logical-level approach needs to maintain and scan historical log to decide right transformation paths for remote update operations. The transformation procedure becomes expensive under heavy workloads [16]. Furthermore, operations defined on the logical view lose their correct position indexes as the document is edited, which creates difficulties for undo operations. The OT-algorithm [20] also introduces "tombstone" as part of their data structure. But the purpose is to resolve transforming ambiguities on operations that update the same portion of a document. We use "tombstone" to create unique position identifiers. The WOOT [19] algorithm and the technique used in the editor TeNDaX [15] belong to the physical-level approach. The WOOT algorithm defines a pair $\langle$*site identifier*, *local counter*$\rangle$ to create a unique identifier for each character. The *site identifier* is unique to each site. The *local counter* is incremented each time a new operation is executed. In TeNDax, each character is assigned a unique integral identifier by a central server. In both approaches, character identifiers are indexed (such as hash-tables or B-trees) so that they can be quickly searched to determine data dependencies between operations. The major problem in their approaches is that the identifiers do not carry ordering information. Therefore, characters that are logically consecutive may not be stored physically consecutive on disk. To maintain the ordering information, for each character, its before- and after- character's identifiers need to be explicitly maintained. When reconstructing a document's content, the system has to follow these before- and after- identifiers iteratively and maps them into their corresponding characters. Consequently, performance of sequential reading is likely to be severely impacted. Position stamps avoid these problems because they are totally ordered and can be efficiently managed for insert, delete, and search in a B-tree. Sequential reading can be quickly done by traversing the leaf nodes of the B-tree. We argue that their approaches are more appropriate to synchronize documents at coarser-granularity such as paragraphs, as demonstrated in their recent work [21].

**Data consistency models** Several consistency models have been proposed in the literature [10, 17, 20, 29]. Those models require convergence and define precedence of operations based on the happen-before relation. Our data-depends relation is a subset of the happen-before relation because if $o_i \to o_j$, $o_i$ must happen before $o_j$, but not vice versa. Therefore, The DDP consistency model is more relaxed than these earlier models. A different type of model is proposed by Oster et al. [19]. Our model is equivalent to theirs at the conceptual level except that we formalize it in a different data structure and prove its correctness.

**Version control systems** Version control systems facilitate collaborative creation of documents by maintaining complete revision history and full revision tracking capabilities. Traditional version control systems [5] are mainly used for source code tracking in which a source repository maintains and merges newly committed changes. Recent decentralized version control systems [9, 12] replicate documents on multiple sites to improve availability and scalability. In these systems, concurrent update conflicts will be presented to users for manual resolution. These systems do not target real-time collaborative editing scenarios in which local updates on shared documents need to be automatically propagated to other sites by their editing tools and automatic conflict resolution strategies are required for concurrent editing.

**Persistent data structures** A data structure is partially persistent if all versions can be accessed but only the newest version can be updated [8]. A PPS is partially persistent because it never deletes any data, and updates are only applied to the latest version. We will be able to dynamically reconstruct any version in its editing history if necessary timestamp information is associated with the position stamps in the sequence. This capability will not be impacted by the reinitialization procedure described in Section 4.4 as long as we maintain the mapping between consecutive PPSs. Various persistent data structures have been proposed in the literature, including stacks, queues, search trees, temporal XML document processing [4, 8, 26]. We take the first initiative to make sequences persistent. Furthermore, our approach is declarative in that a PPS creates descriptive identifiers for its data without relying on a particular storage structure for maintaining position stamps. By comparison, the earlier research is *constructive* in that they maintain adequate meta-information and pointers to old copy of data such that an old version can be dynamically constructed by following the pointers based on the meta-information.

## 7. CONCLUSION

In real-time collaborative editing systems, the collaboration between geographically distributed users needs to be carefully coordinated to maintain data consistency due to editing conflicts. We propose the DDP consistency model that requires convergence of document replicas and preserves the precedence order of editing operations defined by data-dependencies. The DDP consistency model allows users who work on different portions of a shared document collaborate efficiently without interference and allows them to synchronize their editing only when they update overlapped or contiguous characters. For applications that want to support non-interference collaboration on different parts of a shared document and are not sensitive to temporary violation of causality, the DDP consistency model allows higher concurrency and can be enforced more efficiently. Our view synchronization strategy relies on a new data structure–PPS. This data structure is able to create global unique position identifiers for all the characters. As a result, we can easily capture the data-dependencies between any pair of editing operations. The global position identifiers, called position stamps, carry enough information to directly locate their positions in a document. Furthermore, editing operations defined on a PPS can be undone easily due to the fact that it keeps the execution effect of whole editing history. Therefore, our view synchronization strategy can be easily implemented for distributed document editing by insert, delete and undo operations.

Currently, we are looking into the implementation of CESs based on PPSs and many other areas of distributed document processing such as data lineage and fine-granularity access control [13] with the support of position stamps.

## 8. REFERENCES

[1] 7 things you should know about collaborative editing. `http://connect.educause.edu/Library/ELI/7ThingsYouShouldKnowAbout/39386?time=1202527547`, 2005. Educause Connect.

[2] Subethaedit. `http://www.codingmonkeys.de/subethaedit/`, 2009.

[3] Gregory D. Abowd and Alan J. Dix. Giving undo attention. *Interact. Comput.*, 4(3):317–342, 1992.

[4] Bruno Becker, Stephan Gschwind, Thomas Ohler, Bernhard Seeger, and Peter Widmayer. An asymptotically optimal multiversion b-tree. *VLDB J.*, 5(4):264–275, 1996.

[5] Reidar Conradi and Bernhard Westfechtel. Version models for software configuration management. *ACM Comput. Surv.*, 30(2):232–282, 1998.

[6] Charles Crowley. Data structures for text sequences.

[7] Paul Dourish. Consistency guarantees: exploiting application semantics for consistency management in a collaboration toolkit. In *CSCW '96: Proceedings of the 1996 ACM conference on Computer supported cooperative work*, pages 268–277, New York, NY, USA, 1996. ACM.

[8] J R Driscoll, N Sarnak, D D Sleator, and R E Tarjan. Making data structures persistent. In *STOC '86: Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 109–121, New York, NY, USA, 1986. ACM.

[9] Bowei Du and Eric A. Brewer. Dtwiki: a disconnection and intermittency tolerant wiki. In *WWW '08: Proceeding of the 17th international conference on World Wide Web*, pages 945–952, New York, NY, USA, 2008. ACM.

[10] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. *SIGMOD Rec.*, 18(2):399–407, 1989.

[11] Clarence A. Ellis, Simon J. Gibbs, and Gail Rein. Groupware: some issues and experiences. *Commun. ACM*, 34(1):39–58, 1991.

[12] Git. `http://git-scm.com/`. [Online; accessed March-2009].

[13] Thomas B. Hodel, Harald Gall, and Klaus R. Dittrich. Dynamic collaborative business processes within documents. In *SIGDOC '04: Proceedings of the 22nd annual international conference on Design of communication*, pages 97–103, New York, NY, USA, 2004. ACM.

[14] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[15] Stefania Leone, Thomas B. Hodel-Widmer, Michael H. Böhlen, and Klaus R. Dittrich. Tendax, a collaborative database-based real-time editor system. In *EDBT*, pages 1135–1138, 2006.

[16] Du Li and Rui Li. A performance study of group editing algorithms. In *ICPADS '06: Proceedings of the 12th International Conference on Parallel and Distributed Systems*, pages 300–307, Washington, DC, USA, 2006. IEEE Computer Society.

[17] Rui Li and Du Li. A new operational transformation framework for real-time group editors. *IEEE Trans. Parallel Distrib. Syst.*, 18(3):307–319, 2007.

[18] Sylvie Noël and Jean-Marc Robert. Empirical study on collaborative writing: What do co-authors do, use, and like? *Comput. Supported Coop. Work*, 13(1):63–89, 2004.

[19] Gérald Oster, Pascal Urso, Pascal Molli, and Abdessamad Imine. Data consistency for p2p

collaborative editing. In *CSCW '06: Proceedings of the 2006 20th anniversary conference on Computer supported cooperative work*, pages 259–268, New York, NY, USA, 2006. ACM.

[20] Grald Oster. Tombstone transformation functions for ensuring consistency in collaborative editing systems. In *In: The Second International Conference on Collaborative Computing: Networking, Applications and Worksharing (CollaborateCom 2006*. IEEE Press, 2006.

[21] Grald Oster, Pascal Molli, Sergiu Dumitriu, and Rubn Mondjar. UniWiki: A Reliable and Scalable Peer-to-Peer System for Distributing Wiki Applications. Research Report RR-6848, LORIA – INRIA Nancy Grand Est, Feb 2009.

[22] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline. Detection of mutual inconsistency in distributed systems. *IEEE Trans. Softw. Eng.*, 9(3):240–247, 1983.

[23] Atul Prakash and Michael J. Knister. A framework for undoing actions in collaborative systems. *ACM Trans. Comput.-Hum. Interact.*, 1(4):295–330, 1994.

[24] David Ratner, Peter Reiher, and Gerald J. Popek. Dynamic version vector maintenance. Technical Report CSD-970022, UCLA, June 1997.

[25] Matthias Ressel, Doris Nitsche-Ruhland, and Rul Gunzenhäuser. An integrating, transformation-oriented approach to concurrency control and undo in group editors. In *CSCW '96: Proceedings of the 1996 ACM conference on Computer supported cooperative work*, pages 288–297, New York, NY, USA, 1996. ACM.

[26] Flavio Rizzolo and Alejandro A. Vaisman. Temporal xml: modeling, indexing, and query processing. *VLDB J.*, 17(5):1179–1212, 2008.

[27] Yasushi Saito and Marc Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1):42–81, 2005.

[28] Chengzheng Sun. Undo any operation at any time in group editors. In *CSCW '00: Proceedings of the 2000 ACM conference on Computer supported cooperative work*, pages 191–200, New York, NY, USA, 2000. ACM.

[29] 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 Trans. Comput.-Hum. Interact.*, 5(1):63–108, 1998.

[30] S. G. Tammaro, J. N. Mosier, N. C. Goodwin, and G. Spitz. Collaborative writing is hard to support: A field study of collaborative writing. *Comput. Supported Coop. Work*, 6(1):19–51, 1997.

[31] Vadim Tropashko. Nested intervals tree encoding in sql. *SIGMOD Rec.*, 34(2):47–52, 2005.

[32] Stphane Weiss, Pascal Urso, and Pascal Molli. Compensation in collaborative editing. 2007.

[33] Wikipedia. 2008 summer olympics. `http://en.wikipedia.org/wiki/2008_Summer_Olympics`, 2008. [Online; accessed Nov-2008].

[34] Steven Xia, David Sun, Chengzheng Sun, David Chen, and Haifeng Shen. Leveraging single-user applications for multi-user collaboration: the coword approach. In *CSCW '04: Proceedings of the 2004 ACM conference on Computer supported cooperative work*, pages 162–171, New York, NY, USA, 2004. ACM.