Modeling and Implementing Collaborative Editing Systems with Transactional Techniques

Qinyi Wu Calton Pu

School of Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332, USA
Email: {qxw, calton}@cc.gatech.edu

Abstract—**Many collaborative editing systems have been developed for coauthoring documents. These systems generally have different infrastructures and support a subset of interactions found in collaborative environments. In this paper, we propose a transactional framework with two advantages. First, the framework is generic as demonstrated by its capability of modeling four types of existing products: RCS, MediaWiki, Google Docs, and Google Wave. Second, the framework can be layered on the top of a modern database management system to reuse its transaction processing capabilities for data consistency control in both centralized and replicated editing systems. We detail the programming interfaces and the synchronization protocol of our transactional framework and demonstrate its usage through concrete examples. We also describe a prototype implementation of this framework over Oracle Berkeley DB High Availability, a replicated transactional database management system.**

I. Introduction

Collaborative editing systems support geographically distributed users to work on a shared document. These systems in general have specialized implementations and only cover a subset of interactions found in collaborative environments. While it is tempting to develop new algorithms and infrastructures to cover the missing points in the full spectrum of collaborations, any such work will lead to ad hoc implementations and substantial investment of resources.

We have developed a transactional framework to model and implement the whole spectrum of collaborations. This new framework has two advantages. First, it provides primitives to program common editing actions (e.g., insert and delete) as well as to specify permissible interactions between users (e.g., cancel the effect of another user). These primitives allow us to conceptually specify different types of collaborations and reason about their behaviors in terms of granularity of sharing, time to release of individual edits to public, notification of editing conflicts, and conflict reconciliation strategy. The generality of our framework is tested by its capability of specifying four types of collaborative editing systems RCS [32], MediaWiki [6], Google Docs [4], and Google Wave [5]. We further test its generality by using this framework to specify the behavior of a new type of collaboration that is derived by combining features of Google Wave and the approach of acceptance test in handling conflict reconciliation in replicated database management systems (DBMS) [18].

In the second advantage, the framework can be entirely layered on the top of a modern database management system to reuse its transaction processing capabilities for data consistency control in both centralized and replicated editing systems. In centralized collaborative systems, a document is stored at a central server. Users take turns to modify the document [16]. In more recent collaborative editing systems, a document is replicated at geographically distributed sites. Each site is used by one user to modify its local copy. Users can simultaneously modify the document and read the changes of others. Due to network latency, users may modify different versions of the shared document. An important role of replicated editing systems is to bring all divergent document copies into a convergent and consistent state [15], [30]. Though successful, these early techniques require specialized implementations and only handle a subset of collaborations. Our framework supports the entire spectrum of collaborations by reusing the built-in database techniques in concurrency control, crash recovery, and automatic replica synchronization.

Within our framework, we use *partial persistent sequences* (PPSs) [35], a partially persistent data structure, to represent documents and manage them within a database management system. With the help of PPSs, we take the first initiative to define editing conflicts and establish a correctness criterion for collaborative editing systems based on the theory of serializability and the approach of acceptance test for data reconciliation. We also explain the usage of PPSs to support document processing and their implementation issues. We demonstrate the practicality of our framework by building it over Oracle Berkeley DB High Availability [7], a replicated transactional data management system.

In the rest of this paper, we start with an overview of existing collaborative systems and discuss their potential improvements in Section II. We describe the programming interfaces of the proposed framework and its synchronization protocol for data consistency guarantees in Section III. In Section IV, we illustrate the flexibility of our framework by modeling a variety of collaborative models. Then we explain the application of PPSs to data consistency guarantees in Section V. After that, we describe a prototype implementation over Oracle Berkeley DB High Availability in Section VI. The related work is discussed in Section VII.

COLLABORATECOM 2010, October 9-12, Chicago, USA
Copyright © 2011 ICST
DOI 10.4108/icst.collaboratecom.2010.10

II. Overview of Collaborative Editing Systems

We observe a wide spectrum of collaborative editing systems. At one end of the spectrum are version control systems that support only restricted collaboration [13]. At the other end of the spectrum are those "liberal" collaborative editing systems that support highly interactive collaboration [15]. In this section, we first describe four collaborative editing systems to give a brief coverage for the type of collaboration available in practice in Section II-A. For each system, we characterize it in terms of granularity of sharing, time to release of individual edits to public, notification of editing conflicts, and conflict reconciliation strategy. After that, we suggest potential improvements to these systems in Section II-B.

A. Existing Collaborative Editing Systems

Existing collaborative editing systems unanimously adopt the client-server architecture. The server node holds a persistent copy of a shared document. Each client node stores a copy of the shared document. A user at a client node updates the shared document through the local copy. All updates are synchronized to other users through the server node. Below, we describe four collaborative editing systems in the order of their restrictiveness on collaboration.

RCS. It is a version control system. In RCS, a user modifies a document through an explicit check-out step. The document can be checked out by multiple users. Editing conflicts occur if a user attempts to check in a new version whose modifications are based on a stale version. The granularity of sharing is the whole document. A user releases her edits through an explicit check-in step. RCS uses a locking mechanism to detect editing conflicts and notifies impacted users through diagnostic messages. Even though traditionally being used to handle source code in software development, RCS has been recently used to support wiki applications, e.g., Twiki [8].

MediaWiki. It supports fine-grained collaboration among a group of users who simultaneously edit a shared document. Users edit different parts of a document without interference. Editing conflicts occur if more than one user simultaneously edits the same paragraph. A user releases her edits by manually clicking a *save* button. MediaWiki automatically merges users' changes by *diff3* [1], provided that changes happened in different parts of the document. Otherwise, impacted users are notified with diagnostic messages. MediaWiki is the underlying engine for the largest online encyclopedia, Wikiepdia [9].

Google Docs. It supports fine-grained collaboration among a group of users who may simultaneously edit a shared document and at the same time read updates made by other users. Editing conflict occurs if more than one user simultaneously updates the same sentence. A user's updates are automatically synchronized to other users at a fixed time interval (about tens of seconds). Google Docs uses the *differential-synchronization* algorithm [3] to automatically merge changes from different users. The basic idea is similar to *diff3*, but in a streaming fashion. If an automatic merge fails, Google Docs notifies impacted users through diagnostic messages.

Google Wave. It represents the most "liberal" editing system in the sense that Google Wave allows users to edit a shared document anywhere and anytime. The system reconciles editing conflicts automatically under all situations even when users simultaneously edit overlapping areas. In other words, if more than one user simultaneously deletes the same data item, the data item is guaranteed to be deleted exactly once. If more than one user simultaneously inserts new data items at the same position, all the data items are preserved. Google Wave guarantees data consistency based on operational transformation (OT) [15], a non-blocking distributed concurrency control algorithm. Google Wave enforces both convergence and causality preservation properties. The causality preservation follows Lamport's logical clock [21], which require all operations be executed in their happened-before relationships.

In the rest of this paper, we refer the collaboration type supported by RCS as the *check-in/checkout* model. Since MediaWiki and Google Docs support similar level of collaboration except for the time to release a user's edits, they are referred to as the *block-exclusive* model. Finally, we refer to the collaboration type supported by Google Wave as the *update-anywhere-anytime* model.

B. Commentary of Existing Collaborative Editing Systems

We comment on existing systems from five aspects. We make it clear if an aspect is only pertinent to certain types of collaborative editing systems. The aspect list is by no means complete. Other aspects such as access control are not addressed in this paper since they are orthogonal to the problem of data consistency.

Atomicity of grouped operations. There are many cases that a user wants to release a sequence of changes in an atomic step, e.g., a *cut* operation followed by a *paste* operation. Current collaborative editing systems have already included or planned to include this feature in some form of *block edits* that allow users to release her edits in a batch. For example, the next release of Google Wave will enhance the current keystroke-by-keystroke synchronization mode with a block-edit mode. However, the block-edit mode is not atomic in the real sense in that it simply buffers a user's edits and sends them to other users in a batch. It is still possible that the buffered edits are only partially executed at remote sites due to system crash or network intermittence.

Undo An undo operation allows a user to go back to a previously edited document state. In a single-user setting, the implementation of undo can be done by logging adequate

information for the pre-image and post-image of a document transformed by each editing operation. In a multi-user setting, two problems arise. First, the choice of which operation to undo becomes ambiguous. When a user issues an undo, it is unclear whether the user intends to undo the last operation or undo the last operation received from other users. The problem becomes more difficult if the user wants to undo a sequence of changes which may be interleaved with operations from different users. Second, no standard techniques exist to evaluate and inform users of the impact of undo. In some situations, an undo may produce dangling text that was inserted into a paragraph which would disappear later on. In some other situations, undo can lead to loss of data. We cannot emphasize more in a collaborative environment the importance of making undo predictable and recoverable. For example, in Wikipedia, if a user replaces the current version of an article with one of its previous versions, some edits between these two versions may get lost.

Infrastructure development The four collaborative editing systems described previously differ a lot in the level of restrictiveness on collaboration. Therefore, it is not surprising that each of them uses different implementation techniques. For example, RCS uses a locking mechanism, while Google Wave uses operational transformation [15] for data consistency guarantees. However, it is important to avoid re-investing new resources each time a new type of collaboration comes out.

Automatic merging in a controlled manner. Collaborative editing systems that fall at the update-anywhere-anytime end of the collaboration spectrum normally do automatic merging of updates at best efforts. Even though this can minimize manual reconciliation from users, automatic merging may produce unintended results which may not get noticed immediately. It is therefore important for the system to be able to limit the amount of inconsistency introduced during a merging procedure.

III. A Transactional Framework for collaborative editing Systems

We describe a transactional framework for modeling and implementing collaborative editing systems. Our framework is based on standard transaction services in database management systems such as two-phase locking concurrency control, predicate locking, and write-ahead logging. This framework is applicable to documents consisting of a sequence of data objects. These objects can be instantiated to suit the requirement of a particular application domain. For example, a data object can be a word in a text document or be a XML element in a serialized XML document. Henceforth, we choose text documents to explain our ideas due to its commonality. But the presented ideas and techniques are applicable to all kinds of documents that bear sequential structures. We first describe the programming interfaces of our framework in Section III-A and then describe the synchronization protocol for the replicas of a shared document in Section III-B.

A. Programming Interfaces

There are two sets of programming interfaces for implementing a certain type of collaboration. The interfaces in the first set are used for interacting with a shared document, as described below:

- $Insert(pos, x)$: it inserts a new item 'x' at position pos.
- $Delete(pos)$: it deletes the item at position pos.
- $Read(pos_x, pos_y)$: it reads a range of text between the two items indexed at pos_x and pos_y respectively.

$Insert$ and $Delete$ are standard editing operations. Sometimes we call them *write* operation without differentiation. The $Read$ operation is new since a user may not explicitly tell the underlying collaborative editing system the dependent data items of new changes. However, the knowledge of the data items in a read operation can be obtained either automatically or manually. In an automatic approach, a collaborative editing system either infers the dependent data items based on application-specific knowledge or uses the standard technique *implicit locking* [25] to locate the area where the user's most recent editing activities took place. For example, in the check-in/check-out model, the read set is the whole document. In the block-exclusive model, the read set is the paragraph that contains the modified text. In the manual approach, a user selects a block of text and marks them as being read through a Graphical User Interface (GUI) menu entry.

The programming interfaces in the second set are used to instruct our framework to take transaction-related actions, as described below:

- *Release*: it releases a user's changes to other users since the last release point. All the changes are bracketed within a transaction whose execution is guaranteed with the ACID properties.
- *Save*: it saves the current state of the document and returns with a save-point identifier for later references. The $Save$ operation triggers the execution of a $Release$ as well.
- *SavePivot*: it saves the current state of the document and returns with a pivot-point identifier for later references. The $SavePivot$ operation triggers the execution of a $Release$ as well.
- *Cancel*: it cancels the last write operation (i.e., insert or delete) that has not been released to other users.
- *Revert*: it changes the current state of the document to a state identified by either a save-point or a pivot-point identifier.

A $Release$ operation is useful in controlling the frequency of synchronization with other users. For example, Google Docs may issue a $Release$ command each time a timeout event happens for starting the next round of synchronization with the server.

Both a $Save$ and a $SavePivot$ operation force the framework to save a persistent state of the shared document. These persistent states serve as reference points for a user to undo her changes. They are also useful to reduce the amount of work that a user has to redo during a collaborative editing system

failure or a system crash. The difference is that $SavePivot$ sends the framework an additional message that all edits occurring before this point will not be undone by this user. Usually, $Save$ is used to commit intermediate edits while $SavePivot$ is used to commit milestone edits.

Our framework explicitly differentiates two types of $Undo$ operations. A $Cancel$ undoes the last operation by the local user. Since it has not been released to other users, the last operation can be simply removed from the messaging sending queue of the client. However, a $Revert$ operation requires synchronizations with other users since it may undo the changes on which other users' edits depend. The save-points and pivot-points created by a user are globally visible, which means a user can bring the state of a shared document back to a point saved by other users as well. However, any save-point before the last pivot-point of a user becomes unavailable.

B. Synchronization Protocol Between Client and Server

Our framework uses an optimistic synchronization protocol based on the two-tier replication scheme in [18]. The server hosts the master copy of a shared document. Each client node hosts a copy of the shared document. The master copy reflects the most recent committed updates from all the users. The client copy may be the latest or an old version of the master copy. All transactions committed at the client nodes are *tentative*. They are sent to the server and executed under single-copy serializability in the order in which they are committed at the client node. A tentative transaction becomes a *base* transaction if it is committed at the server node and its effects are integrated into the master copy. The write set of all base transactions are sent to the client nodes and update their replicas in the order they are committed. Since the server node determines a global serializable order for all tentative transactions, document replicas converge to the same state and each of them has a consistent view of the document state.

Regarding the choice of concurrency control algorithm for enforcing the single-copy serializability at the server node, we choose the approach of *acceptance criterion test* in [18] instead of multiversion concurrency control algorithms. Under the master-slave replication scheme, it is possible for a tentative transaction to see a very stale version of the shared document. For example, a user may exit an editing session, edit offline, and re-join days later. During the user's absence, the shared document has gone through many rounds of revisions and many tentative transactions have already committed. To determine serializability for the tentative transactions the user committed offline, a multi-version scheme needs to check both active and committed transactions. The examination cannot simply be done by usual lock conflict check because these committed transactions no longer hold their locks.

The idea of acceptance criterion test is to check whether the result produced by a tentative transaction based on the version at the server node is within an acceptable threshold. We take the first initiative to define such a criterion for collaborative editing systems. In our acceptance criterion, a tentative transaction is considered to be *acceptable* if the difference between the set of data items that it reads at the client node and the set of data items that it reads at the server node is within a configurable threshold θ. We assume that a $write$ operation is always proceeded by a $read$ operation. There are no blind writes. Therefore, we can use the read set of data items to quantify the divergence between these two versions. A quantitative definition of $Accept^{\theta}$ is given in Section V-C after introducing the PPS data structure.

We use $Accept^{\theta}$ to mean the acceptance criterion is passed if the difference is within θ. $Accept^{0}$ means that a tentative transaction must read exactly the same set of data items at the server node. $Accept^{\infty}$ means a tentative transaction can tolerate arbitrary divergence between the data items read at the client node and those at the server node. Of course, there are cases that a write operation totally lost its context and cannot be applied at all. For example, a delete operation attempts to remove an already deleted item. We will come to this issue in Section V and show that all write operations can be precisely defined with the help of PPSs.

IV. MODELING OF COLLABORATIVE EDITING SYSTEMS

In this section, we demonstrate the usage of our framework in modeling three editing models described in Section II-A. To demonstrate the flexibility of our framework, the modeling of an artificial editing model is also described.

Check-in/Check-out Model. In this model, a user modifies a shared document through a sequence of editing operations and releases new changes through a check-in step. We synthesize this model as in Figure 1a. The acceptance criterion of the server node is configured to be $Accept^{0}$. Therefore, if someone modifies the shared document and creates a new version, this transaction will be aborted. In the synthesized code, there is only one $Release$ operation, which is the last operation within an editing session. In a standard check-in/check-out model, a user may save multiple versions before issuing the $Release$ command. These intermediate versions are not visible to other users. They are different from those versions created through $Save$ and $SavePivot$ operations. We assume that these intermediate versions are created in a private space of the user and are handled completely by a standard text editor.

Block-exclusive Model In this model, a user's edits are sent to the server either at a fixed time interval or through a manual click of a "send" button. Both events cause the execution of a $Release$ command. Users do not interfere unless they work on the same part of a document. We synthesize this model as in Figure 1b. A $Read$ operation is followed by a sequence of write operations that updated the text within the range of the $Read$ operation. A *bounding block* consists of the read text. Its content is application specific. For example, in MediaWiki it is the paragraph where these write operations took place. In Google Docs, it is the sentence. The acceptance criterion is set to be $Accept^{0}$.

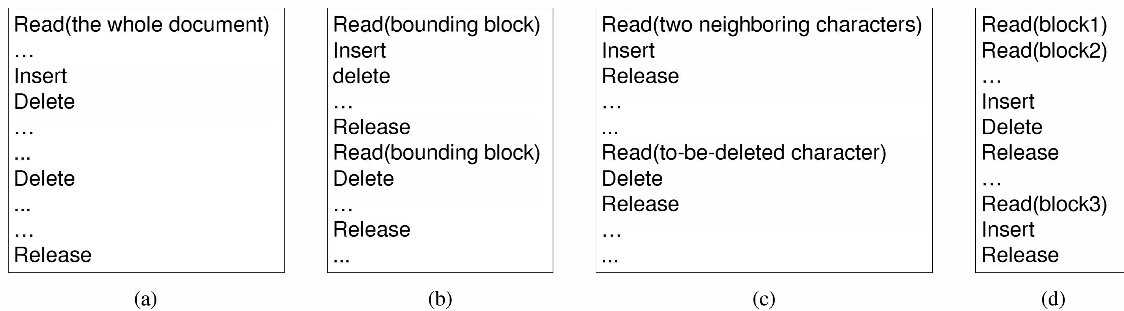

Fig. 1: Examples of synthesized code. a)Check-in/check-out; b)Block-exclusive; c)Update-anywhere-anytime; d)Read-from

Update-anywhere-anytime Model. In this model, users update the shared document without any restriction. All editing conflicts are automatically reconciled. We synthesize this model as in Figure 1c. Every write operation is followed by a $Release$ to synchronize the document replica at the frequency of every keystroke. Each transaction is essentially reduced to a read operation followed by a write operation. For an $Insert$, its read set contains only the two characters neighboring the insertion point. For a $Delete$, its read set is exactly the character to be deleted. The acceptance criterion is configured to be $Accept^{\infty}$. Since θ is set to be ∞, the framework essentially enforces read-committed isolation [33] because each tentative transaction only reads the data written by committed transactions based on our synchronization protocol described in Section III-B. Under read-committed isolation, transactions are susceptible to lost updates and phantom problems. More specifically, it is possible that two users simultaneously delete the same data item or insert new items at the same location. In Section V-C, we explain in detail how our framework is able to produce the same result as that of operational transformation when $\theta = \infty$. Since all document replicas are updated in a global serializable order and all tentative transactions are applied in the order they committed at the client nodes, both the convergence property and the causality preservation property are preserved.

Read-from Model. We introduce a new editing model to demonstrate the flexibility of our transactional framework. In this model, a user can select blocks of text by the mouse in different parts of a shared document and notify the system that the follow-up changes depend on them. The user releases new changes at a fixed time interval or the click of a "send" button. This model is synthesized as in Figure 1d. When the server merges the user's new edits, the user is willing to accept the result if the text the user read is only slightly different from the original. In this case, the θ is set to be a small positive integer. This model has two distinct features. First, a user can monitor the changes in other parts of the document without blocking other users from editing. Second, the model is able to quantify the discrepancy between what a user has viewed and what is actually produced. This feature is useful because it creates a smoother editing environment since the user will not be asked for manual reconciliation if other users only did minor changes to the text such as grammar or spelling corrections. Meanwhile, the user has the assurance of being notified for big changes.

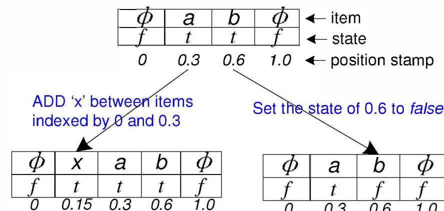

Fig. 2: A PPS example and its updates

V. Implementation Based On the PPS Data Structure

Partial persistent sequence (PPS) is a data structure that always preserves the previous version of a sequence when it is modified, but only the latest version can be modified [35]. We start by a background introduction for PPSs and then explain how to use it for document processing. After that we explain the usage of PPSs to realize the synchronization protocol of replicated collaborative editing systems and the handling of reverts. Finally, we discuss the implementation issues of PPSs.

A. Partial Persistent Sequences

A PPS represents a sequence as a list of ordered items indexed by rational numbers. Figure 2 gives an example for the character sequence "ab" represented as a PPS. ϕ is a special character used to mark the beginning and the end of a PPS. Within our framework, PPSs are used as the implementation data structure for document processing. In order to support both update and undo operations, we slightly change its earlier definition [35] and reintroduce its detail necessary for our explanation. A PPS is defined by a pair $PPS = (S, M)$:

- S: a set of rational numbers, called *position stamps*. $S = \{s_i \in \mathbb{Q}, 1 \leq i \leq n, n \in \mathbb{N}\}$.
- M: a function $M : \mathbb{Q} \rightarrow \Sigma \times \{true, false\}$. Σ consists of a set of data items. M maps each position stamp s to a pair $(item, state)$. Σ contains a null item ϕ different from any other items allowed in user applications. Let $\Sigma^{App} = \Sigma - \phi$. An item is $visible$ if the $state$ of its position stamp is $true$.

Position stamps are ordered by *less than* $<$ operator. s_{i-1} is the largest position stamp that is less than s_i. s_{i+1} is the smallest position stamp that is greater than s_i. The update history of a PPS is defined by $\{(S_k, M_k), 0 \leq k \leq n\}$, where each (S_k, M_k) is called a *version*. When a PPS is first created, its initial version is an empty PPS $S_0 = \{0, 1\}$, $M_0 = \{0 \mapsto (\phi, false), 1 \mapsto (\phi, false)\}$. PPSs support three operations: *Read*, *Add* and *SetState*:

- $Read(s_i, s_j)$: $s_i, s_j \in S_k$, $s_i < s_j$. It returns a set of position stamps with the range of s_i and s_j (inclusive).
- $Add(s_i, s_{i+1}, x)$: $s_i, s_{i+1} \in S_k$, $x \in \Sigma^{App}$. It adds the item x between the item indexed by s_i and the item indexed by s_{i+1}. Let s_{new} be an identifier satisfying the constraint of $s_i < s_{new} < s_{i+1}$. After the update, we have the newer version $S_{k+1} = S_k \bigcup \{s_{new}\}$ and $M_{k+1} = M_k \bigcup \{s_{new} \mapsto (x, true)\}$.
- $SetState(s_i, state)$: $s_i \in S_k$, $state \in \{false, true\}$. It sets $M_k(s_i).state = state$. After the update, we have the newer version $S_{k+1} = S_k$ and $M_{k+1} = (M_k - \{s_i \mapsto M_k(s_i)\}) \bigcup \{s_i \mapsto (M_k(s_i).item, state)\}$.

Figure 2 illustrates how a PPS is modified. The Add inserts a new item 'x' between the data items indexed by position stamps 0 and 0.3 respectively and adds a new position stamp 0.15. The $SetState$ sets the state of 0.6 to $false$.

B. Mapping Between A Document and A PPS

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 create 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 [2]. In our case, we choose the PPS data structure. 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*.

The physical view determines the logical view. The mapping from the physical view to the logical view is defined by:

$$LV((S, M)[s_i, s_j]) = \prod_{s_x \in [s_i, s_j]} M(s_x).item \wedge M(s_x).state$$

$\prod$ denotes concatenation, $[s_i, s_j] = \{s_x | s_i \leq s_x \leq s_j \wedge s_x \in S\}$ the set of position stamps falling between s_i and s_j, and $(S, M)[s_i, s_j]$ a consecutive portion of PPS starting with s_i and ending at s_j. Conceptually, LV is the concatenation of visible items indexed by position stamps within $[s_i, s_j]$.

On the other hand, an update at the logical view can also uniquely locate its position in the physical view. The first data item in the logical view corresponds to the first visible data item in the PPS. Similarly, the i-th data item in the logical view corresponds to the i-th visible data item in the PPS. Therefore, we can always map the editing operations of $Insert$, $Delete$, $Read$ to its corresponding forms on the physical view. The editing operations on the logical view are mapped to the physical view as follows:

- $Insert(pos, x)$ is mapped to $Add(s_i, s_{i+1}, x)$, where s_i satisfies the condition $|LV((S, M)[0, s_i])| = pos$, where $||$ denotes the length of a sequence.
- $Delete(pos)$ is mapped to $SetState(s_i, false)$, where s_i satisfies the condition $|LV((S, M)[0, s_i])| = pos$.
- $Read(pos_x, pos_y)$ is mapped to $Read(s_i, s_j)$, where s_i satisfies the condition $|LV((S, M)[0, s_i])| = pos_x$ and s_j satisfies the condition $|LV((S, M)[0, s_j])| = pos_y$.

A $Read$ only includes the position stamps at the two end points for the text within the range of pos_x and pos_y. This is important because it avoids communication overhead for moving data items between machines given that $Read$ operations are frequent and may involve a large amount of data items.

C. Enforcement of the Synchronization Protocol

The PPS data structure has two important properties which make it an attractive candidate for enforcing data consistency in collaborative editing systems. First, position stamps are unique and consistent to the sequential structure of a document. Therefore, they can be used as primary keys to store a document in a DBMS. All editing operations can be represented as standard database operations and executed by the DBMS in a conventional way. Second, a PPS never deletes any data items. This property makes it possible to reconstruct any version of the PPS to detect editing conflicts in a replicated setting. In this section, we explain how to efficiently validate the acceptance criterion mentioned in Section III-B based on PPSs.

Given a tentative transaction t defined on the version (S_u, M_u) of a document replica, let the version of the master copy at the server be (S_v, M_v). The acceptance criterion test checks whether the editing distance between the data items read on (S_u, M_u) and the data items read on (S_v, M_v) exceeds the threshold θ, as defined below:

Definition 1. Acceptance criterion $Accept^\theta$. Given a transaction t defined on (S_u, M_u), we say that t passes the acceptance criterion of $Accept^\theta$ on (S_v, M_v) if $\sum_{read(s_i, s_j) \in t} Diff(LV([s_i, s_j]_u), LV([s_i, s_j]_v)) \leq \theta$, where $Diff$ is a difference algorithm.

Since each $Read(s_i, s_j)$ only contains the position stamps at the two end points for the range of text a transaction read, it does not provide adequate information for correct validation. For example, in Figure 3 the logical view of PPS_1 is "ab" and the logical view of PPS_2 is "abc". They have different views between $[0.3, 0.6]$. With only $Read(0.3, 0.6)$, it is unsure whether they have the same set of visible data items. However, it turns out we can design a correct validation algorithm by introducing some version information.

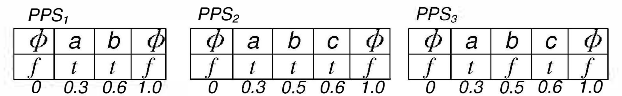

Fig. 3: Examples of PPSs with different logical views

```
AcceptTest(t, V_client, θ)
1  diverge ← 0
2  FOR each Read(s_i, s_j) ∈ t DO
3      A ← read all position stamps between s_i and s_j
4          FOR each s_x ∈ A DO
5              IF V_server(s_x) > V_client THEN
6                  diverge ← diverge+1
7              IF diverge > θ THEN
8                  abort
9  FOR each write operation o ∈ t DO
10     execute o
```

Fig. 4: The algorithm for validating $Accept^{\theta}$ for transaction t

In the client-server synchronization protocol, the server maintains a version counter V_{server}. We use $V_{server}(s_x)$ to represent the version that s_x was last written by a committed transaction. Each client maintains a local version counter V_{client}. When a tentative transaction is sent to the server node, it includes the value of V_{client} as well. The server validates all tentative transactions by the algorithm $AcceptTest$ in Figure 4. The $AcceptTest$ checks whether any position stamps within $[s_i, s_j]$ are updated by transactions committed after V_{client}. Each time it detects a new update, it increases the variable $diverge$ (line 5-6). If $diverge$ exceeds θ, the whole transaction is aborted (line 7-8). Otherwise, the transaction will be executed as normal (line 9-10).

$AcceptTest$ is executed as a standard transaction by the DBMS. In the prototype of our framework, position stamps are implemented by the access method B+-tree within the DBMS. Therefore, the range scan procedure (line 3-8) can be done atomically, which guarantees that the correctness of the acceptance criterion test is not compromised.

$AcceptTest$ provides a sufficient, but not necessary condition for validating $Accept^{\theta}$. It is possible that $AcceptTest$ aborts a transaction, which turns out to be acceptable by $Accept^{\theta}$. As shown in Figure 3, PPS_1 and PPS_3 have the same view, but $AcceptTest$ will abort a transaction if it reads $Read(0.3, 0.6)$ under $Accept^{0}$. However, $AcceptTest$ provides a practical solution because it adds negligible network communication overhead for $Read$ operations.

When $\theta \neq 0$, the editing system admits non-serializable interleaving of transactions. For example, a transaction tries to delete data items that have been deleted or do an insert at a position containing unseen items inserted by previously committed transactions. Our framework handles these situations as follows. For a $Delete$, it will be executed as normal because a $Delete$ operation is mapped to $SetState(s_x, false)$. In the PPS, it is mapped to write the $state$ of s_x to $false$ multiple times. From a user's perspective, the data item is deleted exactly once. When it is an $Insert$, the server first checks whether there are any items between s_i and s_{i+1}. If no new position stamps are present, it does the $ADD(s_i, s_{i+1}, x)$ by inserting a new position stamp s_{new} as usual. Otherwise, the server will query the DBMS to get the next position stamp s_k greater than s_i and does $ADD(s_i, s_k, x)$ instead.

D. Revert Handling

A $Revert$ operation reverts the state of a shared document to a previous *save-point* or *pivot-point*. When the server receives a $Revert$ operation, it checks its log entries and locates all the transactions committed after that point. If the revert point is located before the most recent *pivot-point* in the server's log, the server will abort this transaction and respond back to the client along with the identifier for the most recent *pivot-point*. The client can optionally resubmit the revert request with this new reference point. Let $o_1 o_2 ... o_n$ be the sequence of operations that need to be reverted. The compensating transaction is constructed as $\overline{o_n}\ \overline{o_{n-1}} ... \overline{o_1}$ based on the following rules:

- if o_i is a $Read$, its compensating operation is $\overline{o_i} = \phi$, which is simply ignored.
- if o_i is a $SetState(s_x, state)$, its compensating operation is $\overline{o_i} = SetState(s_x, \overline{state})$;

The compensating transaction undoes, from the user's perspective, any operations that are performed by the transactions committed after the reverted point. A big advantage of handling $Revert$ based on PPSs is that the construction of a compensating transaction is completely operational.

E. Implementation Issues for PPSs

The previous discussion for PPS assumes that data items are never removed and a machine has unbounded precision bits for representing position stamps. While this is valid from a theoretical point of view, which enables us to explain the framework in a concise way, it is rare in practice that collaborative editing systems allow its data to grow unbounded. Therefore, a garbage collection algorithm is used to periodically rebalance the PPS data structure and reassign visible data items with new position stamps.

The server starts the garbage collection process when any of the three events happens: 1) the data storage for the PPS exceeds a threshold; 2) the PPS runs out of precision bits; 3) all users exit an editing session. The server starts a distributed consensus algorithm such as two-phase commit to coordinate the garbage collection process. The server maintains the pre-image and post-image of a PPS at the end of the process and maintains the mapping between the old position stamps and the new position stamps for visible data items. Therefore, if a client node submits a transaction based on an old PPS, the server can use the mapping to determine the right data items to update. Each rebalanced PPS is uniquely identified by a *rebalance-identifier*. All document replicas maintain the *rebalance-identifier* for its local PPS and will include it in all the transactions sent to the server.

Even though the garbage collection process uses a distributed synchronization algorithm, we do not expect it to raise much concern. A user is able to continue her regular edits since all transactions are tentatively committed on its local copy. The garbage collection only delays the time of synchronizing new changes to the replicas of other users.

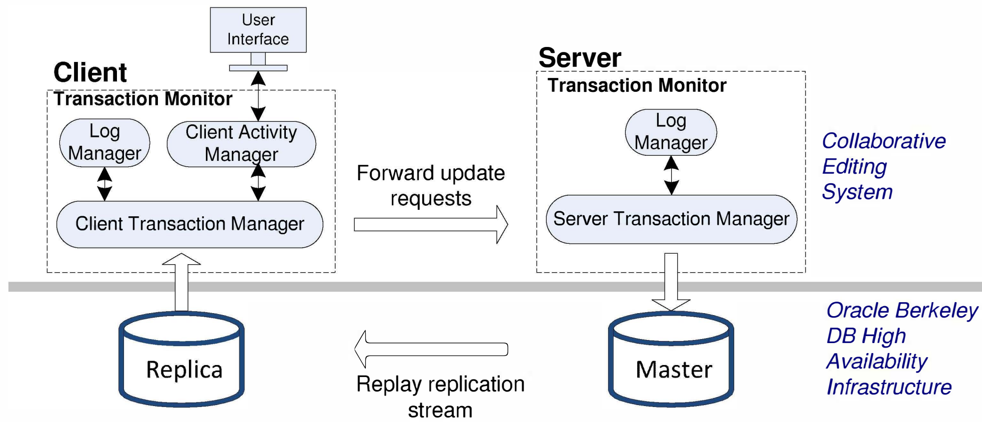

Fig. 5: System architecture

VI. Collaborative Editing System Prototype

We have implemented our transactional framework over Oracle Berkeley DB High Availability. In this section, we first provide a background description for this replicated DBMS in Section VI-A and give an overview of our system architecture in Section VI-B. We then explain different modules of our framework in Section VI-C.

A. *Oracle Berkeley DB High Availability Infrastructure*

Oracle Berkeley DB High Availability enables replication of a database across a collection of nodes. These nodes form a replication group. Within the group, one node is elected to be the master, while the rest of the nodes are referred to as replica. The master node accepts both read and write transactions, while the replica nodes accept read-only transactions. A replica node communicates with the master node through a logical replication stream that contains a description of the logical changes of the master node. The stream is replayed at the replica using an internal replay mechanism. In our implementation, a client node maintains the state of the shared document in a replica node, while the server node maintains the state of the shared document in a master node.

B. *System Architecture*

In our implementation, a shared document is replicated across a collection of client nodes and one server node. Each client node is used by one user to modify the shared document. The server node is responsible for integrating changes from all client nodes and replay these changes to all replicas. Figure 5 shows the system architecture between a client node and a server node. When a user issues new edits, the user sees their effect immediately. Meanwhile, these edits are wrapped in the form of transactions and forwarded to the server node. The server node processes each transaction in two steps: 1) run it against an acceptance test; and 2) execute the transaction in the master node if it passes the acceptance test, otherwise abort the transaction. Meanwhile, the changes at the master node streams to all replica nodes. Each client node periodically refreshes its document copy based on the latest state of its replica.

Oracle Berkeley DB High Availability provides several benefits for developing collaborative editing systems. First, atomicity is a given-in property in transactions. Second, our synchronization protocol can be completely implemented based on the available concurrency control algorithm. Third, the replicated DBMS simplifies recovery. If a client node restarts after a crash, its replica is automatically brought to the latest state of the master node. Finally, the DBMS handles durability automatically for a collaborative editing system. The update of a user is guaranteed to be persistent as soon as it commits at the master node.

C. *Implementation Modules*

We have implemented a transactional monitor at both the client side and the server side to synchronize distributed editing activities. The interaction of these modules is illustrated in Figure 5. Below we describe each of them.

Client Activity Manager (CAM). It receives a sequence of operations from the text editor. When it sees an operation of $Insert$, $Delete$ or $Read$, CAM appends it to a buffer. Otherwise, it takes the following actions:

- For a $Release$, CAM wraps all the operations in the buffer and brackets them within the two control operations *Begin-transaction* and *End-transaction* and sends it to the underlying transaction manager. Then CAM empties the buffer. The *Begin-transaction* and *End-transaction* are used to indicate the beginning and the end of a classic transaction.
- For a $Save$ or a $SavePivot$, CAM takes an action similar to the handling of $Release$, except that it additionally includes a $Save$ or $SavePivot$ as the last operation within the transaction.
- For a $Cancel$, CAM removes the last entry from its buffer.
- For a $Revert$, CAM brackets this operation parameterized with its $Save$ or $SavePivot$ within *Begin-transaction* and *End-transaction* and sends the transaction to its underlying module.

Client Transaction Manager (CTM). It is responsible for forwarding transactions received from CAM to the server and monitoring their progress. CTM maintains all pending transactions in a queue and waits for responses from the server. CTM assumes that the server responds to pending transactions in the order they are sent. On receiving a response from the server, it removes the transaction from the head of the queue. If the response is a commit, it takes no action since the transaction has committed at the master node and is going to be replayed at its local replica. If the response is an abort, it generates a diagnostic message to the user. The abort a transaction may cause the abort of subsequent pending transactions that read the results of the aborted transaction. If a cascading abort happens, all the aborted transactions are removed from the queue and their states will be included in the diagnostic message.

Server Transaction Manager (STM). It is responsible for processing all client transactions under single-copy serializability. Upon receiving a transaction, STM forwards the *Begin-transaction* and *End-transaction* as well as the document editing operations to its underlying DBMS where the transaction is processed in a conventional way. Due to simultaneous editing, a client transaction may see a different version of the shared document and produces different results. To quantitatively measure the divergent distance, STM runs all client transactions against the *acceptance criterion test* introduced in Section V-C. If passed, the transaction is committed, otherwise get aborted. STM then returns its state to its corresponding client node.

Log Manager (LM). It maintains log entries for the execution history of transactions. Each log entry contains the read and write set of a transaction. To support $Cancel$, the log entries of a transaction are backward chained to identify operations within a transaction. LM also maintains a special *save-point* or *pivot-point* log entry as a marker in its log for handling $Revert$ operations.

VII. Related Work

Many extended transaction models have been developed to establish a theoretical foundation for specifying correctness in cooperative applications [10], [11], [20], [23], [27]. Even though these advanced models are capable of modeling open-ended and dynamic editing activities, their applicability to collaborative editing systems is handicapped by the mismatch between the set-based relational data model and the sequential structure of a document. Moreover, standard definitions have not been established for editing operations regarding transaction boundaries and editing conflicts. A lot of successful efforts have been attempted for managing XML documents within relational DBMSs [28], [14], [31] and apply these advanced transaction model to support editing activities [19]. But not much work has been done for documents with sequential data items. The few efforts we are aware of are the work in [12], [22], [29]. The major reason is that a sequential document are indexed by ephemeral keys which are prone to change due to document modification. The PPS data structure address this issue by assigning immutable and ordered identifiers to the data items of a document. Our framework adopts two techniques from the earlier work. First, the handling of revert follows the compensation technique in Sagas [17]. Second, the introduction of a pivot-point to define an irreversible reference point for handling backward recovery of transaction processing was first proposed by Mehrotra et.al. [24].

Our framework is based on the persistent data structure PPS. There are two other persistent data structures [26] [34] that might be alternative choices to PPS. These two structures create ordered path-based indexes for unstructured text documents. Both approaches provide optimization techniques for reducing the growing length of indexes. The major concern for path-based indexes is the space overhead because they may grow very long at the places where the text was updated frequently. Another concern is the matching cost, which is proportional to the length of the paths. To prevent indexes from growing unbounded, both approaches have to rebalance their data structures at some points. Certainly, solid experimental studies are needed to determine the best data structure for our framework.

VIII. Conclusion

We propose a transactional framework for modeling and implementing collaborative editing systems. Our framework demonstrates its advantages in two ways. First, it provides a conceptual framework to specify the entire spectrum of collaborations. We demonstrate its generality and flexibility through its capabilities of specifying four types of collaborative editing systems and a new collaboration model. In the second advantage, our framework can be layered on the top of a database management system to reuse its transactional techniques for data consistency guarantees in both centralized and replicated collaborative editing systems. This is demonstrated through a prototype implementation over Berkeley DB High Availability, a replicated database management system. As the next step, we will study the issue of system scalability and the impact of this technology transfer.

ACKNOWLEDGMENTS This research has been partially funded by National Science Foundation by IUCRC, CyberTrust, CISE/CRI, and NetSE programs, National Institutes of Health grant U54 RR 024380-01, PHS Grant (UL1 RR025008, KL2 RR025009 or TL1 RR025010) from the Clinical and Translational Science Award program, National Center for Research Resources, and gifts, grants, or contracts from Wipro Technologies, Fujitsu Labs, Amazon Web Services in Education program, and Georgia Tech Foundation through the John P. Imlay, Jr. Chair endowment. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or other funding agencies and companies mentioned above.

REFERENCES

[1] Comparing and merging files. http://www.gnu.org/software/diffutils/manual/html_mono/diff.html, 2010.

[2] Data structures for text sequences. http://www.cs.unm.edu/~crowley/papers/sds/sds.html, 2010. Charles Crowley.

[3] Differential synchronization overview. http://neil.fraser.name/writing/sync/, 2010.

[4] Google docs basics. http://docs.google.com/support/bin/static.py?hl=en&page=guide.cs&guide=20322, 2010.

[5] Googlewave white paper. http://www.waveprotocol.org/whitepapers/operational-transform, 2010.

[6] Mediawiki 1.15.1. http://www.mediawiki.org/wiki/MediaWiki, 2010.

[7] Oracle berkeley db java edition high availability. http://www.oracle.com/technology/products/berkeley-db/pdf/berkeleydb-je-ha-whitepaper.pdf, 2010.

[8] Twiki system requirements. http://twiki.org/cgi-bin/view/TWiki/TWikiSystemRequirements, 2010.

[9] Wikipedia: a free, web-based, collaborative, multilingual encyclopedia. http://en.wikipedia.org/wiki/Wikipedia, 2010.

[10] François Bancilhon, Won Kim, and Henry F. Korth. A model of cad transactions. In *VLDB '1985: Proceedings of the 11th international conference on Very Large Data Bases*, pages 25–33. VLDB Endowment, 1985.

[11] Panayiotis K. Chrysanthis and Krithi Ramamritham. Acta: a framework for specifying and reasoning about transaction structure and behavior. *SIGMOD Rec.*, 19(2):194–203, 1990.

[12] Chris Clifton and Hector Garcie-Molina. The design of a document database. In *DOCPROCS '88: Proceedings of the ACM conference on Document processing systems*, pages 125–134, New York, NY, USA, 1988. ACM.

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

[14] Fang Du, Sihem Amer-Yahia, and Juliana Freire. Shrex: managing xml documents in relational databases. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 1297–1300. VLDB Endowment, 2004.

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

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

[17] Hector Garcia-Molina and Kenneth Salem. Sagas. In *SIGMOD '87: Proceedings of the 1987 ACM SIGMOD international conference on Management of data*, pages 249–259, New York, NY, USA, 1987. ACM.

[18] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha. The dangers of replication and a solution. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 173–182, New York, NY, USA, 1996. ACM.

[19] Francis Gropengieβer, Katja Hose, and Kai-Uwe Sattler. An extended transaction model for cooperative authoring of xml data. *Computer Science - Research and Development*, 24(1):85–100, 2009.

[20] George T. Heineman. A transaction manager component for cooperative transaction models. In *CASCON '93: Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research*, pages 910–918. IBM Press, 1993.

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

[22] 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.

[23] Luigi V. Mancini, Indrajit Ray, Sushil Jajodia, and Elisa Bertino. Flexible transaction dependencies in database systems. *Distrib. Parallel Databases*, 8(4):399–446, 2000.

[24] S. Mehrotra, R. Rastogi, A. Silberschatz, and H.F. Korth. A transaction model for multidatabase systems. pages 56 –63, jun 1992.

[25] R. E. Newman-Wolfe, M. L. Webb, and M. Montes. Implicit locking in the ensemble concurrent object-oriented graphics editor. In *CSCW '92: Proceedings of the 1992 ACM conference on Computer-supported cooperative work*, pages 265–272, New York, NY, USA, 1992. ACM.

[26] Nuno Preguica, Joan Manuel Marques, Marc Shapiro, and Mihai Letia. A commutative replicated data type for cooperative editing. *Distributed Computing Systems, International Conference on*, 0:395–403, 2009.

[27] Calton Pu, Gail E. Kaiser, and Norman C. Hutchinson. Split-transactions for open-ended activities. In *VLDB '88: Proceedings of the 14th International Conference on Very Large Data Bases*, pages 26–37, San Francisco, CA, USA, 1988. Morgan Kaufmann Publishers Inc.

[28] Jayavel Shanmugasundaram, Kristin Tufte, Chun Zhang, Gang He, David J. DeWitt, and Jeffrey F. Naughton. Relational databases for querying xml documents: Limitations and opportunities. In *VLDB '99: Proceedings of the 25th International Conference on Very Large Data Bases*, pages 302–314, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[29] Michael Stonebraker, Heidi Stettner, Nadene Lynn, Joseph Kalash, and Antonin Guttman. Document processing in a relational database system. *ACM Trans. Inf. Syst.*, 1(2):143–158, 1983.

[30] 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.

[31] Igor Tatarinov, Stratis D. Viglas, Kevin Beyer, Jayavel Shanmugasundaram, Eugene Shekita, and Chun Zhang. Storing and querying ordered xml using a relational database system. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 204–215, New York, NY, USA, 2002. ACM.

[32] Walter F. Tichy. Rcs—a system for version control. *Softw. Pract. Exper.*, 15(7):637–654, 1985.

[33] Gerhard Weikum and Gottfried Vossen. *Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[34] Stephane Weiss, Pascal Urso, and Pascal Molli. Logoot: A scalable optimistic replication algorithm for collaborative editing on p2p networks. In *ICDCS '09: Proceedings of the 2009 29th IEEE International Conference on Distributed Computing Systems*, pages 404–412, Washington, DC, USA, 2009. IEEE Computer Society.

[35] Qinyi Wu, Calton Pu, and João Eduardo Ferreira. A partial persistent data structure to support consistent shared access in collaborative editing applications. In *ICDE*, 2010.