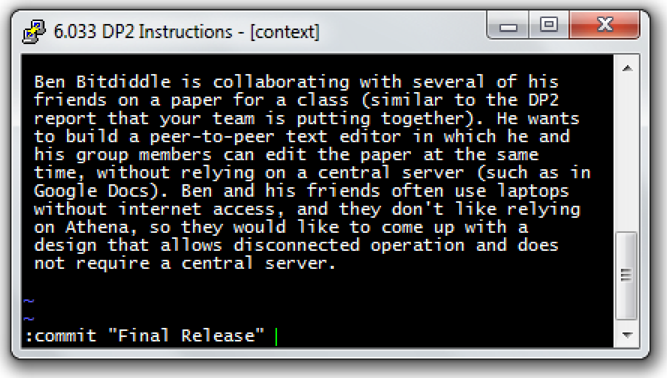

```
Ben Bitdiddle is collaborating with several of his
friends on a paper for a class (similar to the DP2
report that your team is putting together). He wants
to build a peer-to-peer text editor in which he and
his group members can edit the paper at the same
time, without relying on a central server (such as in
Google Docs). Ben and his friends often use laptops
without internet access, and they don't like relying
on Athena, so they would like to come up with a
design that allows disconnected operation and does
not require a central server.

~
~
:commit "Final Release"
```

Context: A P2P Concurrent Text Editor

http://mit.edu/das/www/context/

Ian Chan, Somak Das, Vineet Gopal
{ianchan0, das, vineetg}@mit.edu
Katabi (R04)
6.033 Design Project 2
May 10, 2012

1 Introduction

This report explores the design of *Context*, a <u>con</u>current <u>text</u> editor for a peer-to-peer network of users. Context is fault-tolerant and operational even with failed nodes and unstable networks, making this design ideal for collaborative work within groups. Our system allows users to simultaneously edit local copies of documents before broadcasting the updates to other online users through pairwise synchronization. Context's reconciliation process automatically merges non-conflicting changes and prompts users to resolve conflicting ones. Finally, the design utilizes a two-phase commit protocol to support initializing and finalizing commits.

2 Design

2.1 Document Representation

In Context, each group member has his own "working copy" of the shared document. This local document is stored on disk as a Context File (with extension `.ctx`). Its main data structure is the *doc-list*, a doubly linked list of sentence records, which stores the bulk of the information needed for user editing and system management.

2.1.1 Version Tracking

A user ID uniquely identifies the user; in our design, it is his machine's IP address. The machine stores his local copy of the document as a Context File. The file maintains a local *version number*, initialized to 0 and incremented when the user makes an update to the document. Any change to document text (e.g., a keystroke) counts as an update.

Let us now take a global view. To track updates to the document by multiple users at different times, Context employs *version vectors*, a list of key–value pairs of the form (user ID, version

Table I. Operations on version vectors (IP addresses omitted for simplicity).

Definition	Example
A > B, *greater than*: each version number in A is greater than or equal to the corresponding version number in the B, and at least one is strictly greater than	$\langle 1, 1, 1\rangle > \langle 1, 0, 0\rangle$
A ~ B, *concurrent to*: at least one version number in A greater than its counterpart in B, and at least one version number in A is less than its counterpart in B	$\langle 1, 0, 1\rangle \sim \langle 1, 1, 0\rangle$
A $\gtrsim$ B, *greater than or concurrent to*: A > B or A ~ B	$\langle 1, 0, 1\rangle \gtrsim \langle 1, 1, 0\rangle$
A = B, *equal to*: each version number in A is equal to the its counterpart in B	$\langle 1, 2, 1\rangle = \langle 1, 2, 1\rangle$
max(A, B), *maximum of*: take the maximum of each version number in A and its counterpart in B	$\max(\langle 1, 0, 2\rangle, \langle 1, 1, 1\rangle) = \langle 1, 1, 2\rangle$

number) for all users in the group. This mechanism allows the system to determine the relative timing of updates. To facilitate the merge and commit protocols, the local document keeps track of its current version vector (in addition to version number). These protocols use several operations on the version vectors of two document copies, defined in Table I.

2.1.2 Doc-list

In the anatomy of a Context File, the *doc-list* represents the actual document. It is a doubly linked list of *sentence records* (see Figure 1). The concept of a "sentence" is left to the users to decide; during document creation, they are allowed to set the sentence separator: for example, `"\n"` (newline) for a code file or `"[.!\?]\s"` (punctuation mark + space) for English text. To simplify the placing of leading and trailing sentences within the doc-list, the head and tail serve as dummy nodes for the beginning and end of the list, respectively. The *last-modified* field of the head stores the document version vector, for reasons made clear in §3.6.2.

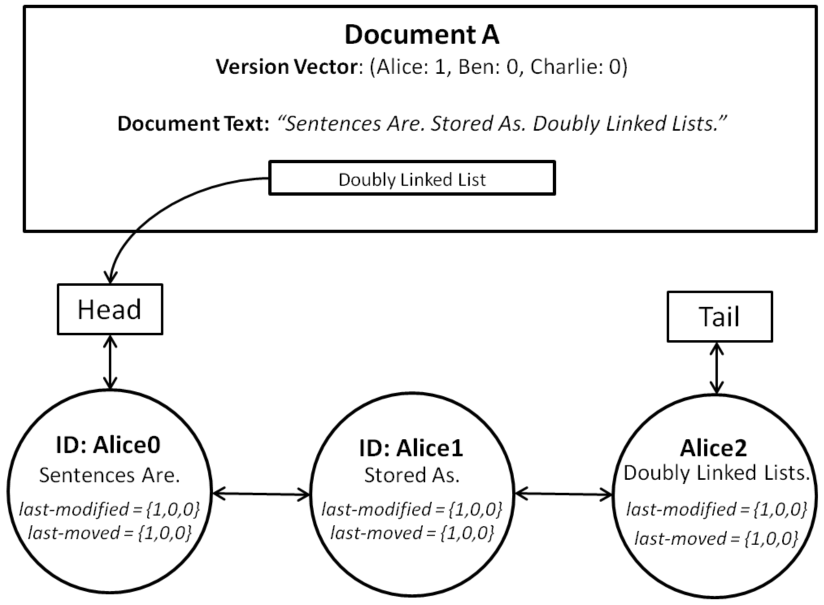

Figure 1. Context represents each document as a doubly linked list of sentence records. A head node precedes the first sentence, and a tail node follows the last sentence. There is no direct pointer to the tail for reasons discussed in §3.6.2.

2.1.3 Sentence Records

Context tracks changes to the document by reference, not by value. Each sentence record in the doc-list has fields corresponding to its content (for user editing) and version tracking (for system management), bypassing the need for duplicate of storage within a log (shown in Figure 2). This design decision allows us to more easily construct an idempotent reconciliation procedure (see §2.2) and reduces disk usage.

```
struct sentence-record:
    integer id;
    string-buffer content;
    version-vector last-modified;
    version-vector last-moved;
```

Figure 2. Sentence record structure.

ID number	At time of creation, each sentence is associated with an ID: a concatenation of (i) the IP address of the peer where the sentence originated and (ii) the value of a local auto-incrementing counter. This ensures that IDs from different peers do not conflict; they are globally unique.
Content	The content of each sentence is a string buffer—variable-length strings optimized for insertion and deletion operations.
Last modified, last moved	These fields store version vectors. They effectively track the changes in sentence content (modified) and positioning within the document (added or moved), along with when those changes happened, via the document's version vector at the time. This enables us to separate sentence modifications from sentence movements, thus reducing the number of potential conflicts.

2.1.4 Deleted-set

The doc-list cannot store sentences removed by the user, but the system must still track those updates for merging purposes. They are put in a hash-set of sentence records, aptly named *deleted-set*. However, if not managed properly, the set could grow infinitely after a long period of editing. To address this issue, §2.2.6 explains techniques for pruning the set while still maintaining the ability to notify other users about the deletes.

2.1.5 Id-map

In many places, the merge procedure takes as input a reference to a sentence record by ID. Thus, an $O(1)$ lookup is preferable over an $O(N)$ scan through a doc-list of size N. To achieve this performance, the local document maintains a lookup table, *id-map*; each entry maps a sentence ID to a pointer to the corresponding sentence record in the doc-list.

2.1.6 Cursor Position

The system tracks the user's current cursor position as a ⟨sentence ID, index⟩ tuple in memory. Similarly, a cursor selection has a start and an end position. For example, ⟨34, 5⟩ represents an insertion point before the fifth character in the sentence with ID 34. This scheme is well-suited to our design, which manages text at the sentence level.

A user's document may be constantly synchronized with the documents of other users, but our scheme avoids cursor misalignment. Suppose multiple users are writing text to the same location in the document. If they are inserting new sentences, then Context directs their text input to different sentences with unique IDs (and so the cursors are kept separate too). On the other hand, if they are modifying the same sentence, then their cursor positions are a nonissue. Once they merge, the changes should conflict and require manual resolution anyway.

If a user's cursor is idle (no activity) in a sentence being modified by another user, then the cursor might be arbitrarily moved after the merge. However, this is a problem with many current text editors [1], so we leave it as a future improvement.

2.1.7 Putting It All Together

The doc-list, deleted-set, and sentence records provide the necessary framework to support a vast number of editing operations. Table II provides a small sample of operations and lists the in-memory and on-disk data structures that would be changed in each operation (see also Figure 3). For instance, adding a word to a sentence results in (i) updating the sentence's *content* and (ii) setting its *last-modified* to the document's current version vector. These internal updates ensure that the system tracks all document changes for merges with other users.

Table II. Example updates that a user can make to the local document.

Operation	Description	Internal Updates
Insert word	Add new text at the cursor location	existing sentence's *last-modified* and *content*
Delete word	Remove existing text at the cursor location	existing sentence's *last-modified* and *content*
Insert sentence	Add a sentence of new text (i.e., contains a sentence separator) at cursor location	new sentence's *last-modified*, *last-moved*, and *content* (add to *doc-list*)
Delete sentence	Remove a sentence of existing text in cursor selection	existing sentence's *last-moved* (move from *doc-list* to *deleted-set*)
Move sentence	Move a sentence of existing text in cursor location to a new location	existing sentence's *last-moved* and position in *doc-list*, i.e., the node's *prev*/*next* links

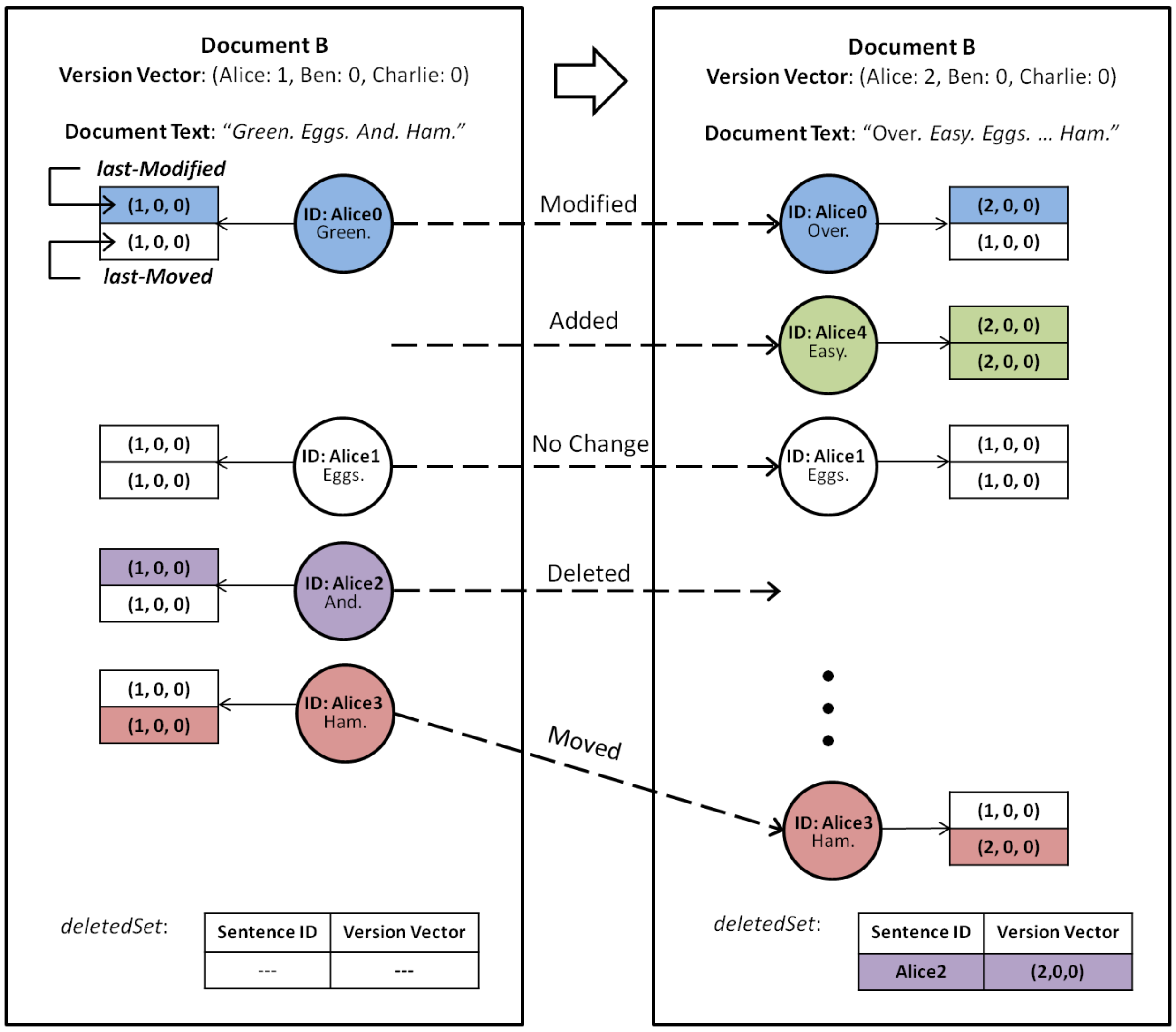

Figure 3. Each node of the linked list is augmented with *last-modified* and *last-moved* version vectors. Between version ⟨1, 0, 0⟩ and ⟨2, 0, 0⟩, Alice has **modified Alice0**, **added Alice4**, **deleted Alice2**, and **moved Alice3**. Note that the deleted-set is actually sentence ID → record map, but we have only displayed the *last-moved* version vector for clarity.

2.2 Merge Protocol

Context implements an auto-save feature, similar to Google Docs. Every user merges his changes with every other connected user shortly after making an update—in our design, once every second. We selected one second because it is fast enough for users to see updates in "real-time," yet slow enough so that the network does not become congested. This merge period can be varied, though, by the Context implementation or user. Also, when any direct or online connection is first established, both peers attempt to merge their documents.

The system can send and receive messages over any network stack that provides reliable, ordered delivery, either through the Internet or via direct connection. (Examples of technologies that directly connect computers together include Ethernet crossover cables and Bluetooth.)

We now describe the algorithm used to merge two document revisions. We define a *conflict* as any change made by users that Content cannot resolve without additional information. Our design minimizes the number of conflicts upon merging. It also minimizes the amount of data sent across the network. Both of these aspects are discussed in our analysis (§3).

2.2.1 Overview

Before going into detail, let us give a brief summary of the merge algorithm. Suppose peers A, B, C, D, and E are all online and are editing the same document. Now, suppose one second has passed since A updated his document. Peer A attempts to merge his document with each of the other four users. The following steps describe how this merge happens between A and B:

1. **Request:** A sends a merge request to B, along with his document version vector VV_A
2. **Send:** B finds changes that A does not have and sends them to A, along with VV_B
3. **Update:** A updates his document with the changes from B and updates VV_A accordingly
4. **Send:** A finds changes that B does not have and sends them to B, along with VV_A
5. **Update:** B updates his document with the changes from A and updates VV_B accordingly

If no other changes were made during this process, then users A and B will have the same document and document version vector. But, if either user makes changes during this process, then the documents may not be identical. However, the version vectors will reflect the differences. Version vectors are also updated differently if conflicts must be resolved.

All merges are performed sequentially and asynchronously. After peer A *sends* changes to B, he may process other *sends* and *updates* before B responds. However, he will not process a request from B (to merge with A) unless his IP address is higher than B's—this peer ordering guarantees that only one user drives the pairwise merge. The integrity of the local document is ensured by the updated version vectors. Moreover, this process works asynchronously with correct behavior because all merges are idempotent (at each step, we check if new changes have been made by the user or another merge).

The implementations of the procedures in Steps 1–3 are now described (Steps 4 & 5 are mirrors of 2 & 3). We provide pseudocode where necessary.

2.2.2 Request

Peer A sends a message to B containing a request to merge and VV_A.

2.2.3 Send

Suppose peer B wants to send peer A any applicable changes. Peer B knows VV_A from *request*. If $VV_A \geq VV_B$, then there are no newer changes and B immediately sends VV_B. Otherwise, Peer B would like to send *modified*, *deleted*, and *moved* sets to A. These sets are created as follows.

$$\textit{modified} = \{\, (s.\textit{id},\ s.\textit{content}) \text{ for } s \in \textit{doc-list} \text{ if } s.\textit{last-modified} \gtrsim VV_A \,\}$$

$$\textit{deleted} = \{\, (s.\textit{id}) \text{ for } s \in \textit{deleted-set} \text{ if } s.\textit{last-moved} \gtrsim VV_A \,\}$$

The creation of the *moved* set is harder. Concurrent moves are troublesome when only examining *prev* and *next* links within the linked list. To ensure that two merged documents are exactly identical, we design a scheme using *blocks*. We say that a sentence *s* has been *shifted* if *s.last-moved* $\gtrsim VV_A$. Consecutive shifted sentences continue a single block. A block contains the

```
$moved = \emptyset$
$block$ = empty block
for $s \in doc\text{-}list$ do
    if $s.last\text{-}moved \gtrsim VV_A$ then
        if $block$ is empty then
            $block.prev = s.prev.id$                    ▷ Start new block
        end if
        $block.sentences$.add($s.id$)                   ▷ Fill current block
    else if $block$ is not empty then
        $block.next = s.id$                             ▷ End current block
        $moved$.add($block$)
        $block$ = empty block
    end if
end for
```

Figure 4. Linear-time algorithm to create the *moved* set of blocks.

IDs of each sentence within it, as well as the IDs of the preceding sentence (*prev*) and following sentence (*next*). The *moved* set stores a list of blocks, and is dynamically populated with just a single iteration through the doc-list (see Figure 4). We send *modified*, *deleted*, and *moved* to peer A, along with VV_B.

2.2.4 Update

Suppose peer A has received changes from peer B. A now integrates these changes into his document using the *update* mechanism. A has three sets of changes to integrate: $deleted_B$, $moved_B$, and $modified_B$ (in that order).

We now describe the criteria for performing these updates. After A merges with B, we set the *last-moved* or *last-modified* field of each changed sentence record in A to VV_B. After each conflict that the user A resolves, we set the *last-moved* or *last-modified* field of the corresponding sentence record (and the document version vector) to max(VV_A, VV_B), but with A's version number incremented. This increment ensures that any two documents with the same version vector have the same content and positioning.

To merge $deleted_B$ into document A, we simply delete any sentences that were modified more recently by B and raise conflicts otherwise. Specifically, we perform the following checks for each *id* in $deleted_B$:

- If *id-map* contains *id*, and *id-map*[*id*].*last-moved* ~ VV_B or *id-map*[*id*].*last-modified* ~ VV_B, then ask the user to resolve the conflict
- Else if *id-map* contains *id*, and *id-map*[*id*].*last-moved* < VV_B or *id-map*[*id*].*last-modified* < VV_B, then delete the sentence from document A
- Else (*id-map* does not contain *id*), do nothing

To merge $moved_B$ into our document, we move sentences by relative position. That is, we use *block.prev* and/or *block.next* as anchors and, from there, insert the moved sentences into the correct location. If both *prev* and *next* have been moved, then we raise a conflict. Also, if the same sentence has been moved to different places by different users, then we raise a conflict. Otherwise, we move sentences as expected. Specifically, we perform the following checks for each *block* in $moved_B$:

- If both *block.prev* and *block.next* have been moved, then ask the user to resolve the conflict
- Else if for any *id* ∈ *block*, *id-map* contains *id* and *id-map*[*id*].*last-moved* ~ VV_B and the relative locations are different (as determined by comparing the *block* order to *doc-list*), then ask the user to resolve the conflict

- Else if for any $id \in block$, *deleted-set* contains *id* and *deleted-set*[*id*].*last-moved* $\sim \mathrm{VV}_B$, then ask the user to resolve the conflict
- Else for each $id \in block$:
  - If *id-map* contains *id* and *id-map*[*id*].*last-moved* $< \mathrm{VV}_B$, then move *id-map*[*id*] to the appropriate location in the *doc-list* (after *block.prev* and/or before *block.next*)
  - Else (*id-map* does not contain *id*) create a new sentence *s* with $s.id = id$ and insert it appropriately

To merge $modified_B$ into our document, we simply raise a conflict if the sentence has concurrent modifications and the modifications are different, or replace the contents otherwise. Specifically, we perform the following checks for each (*id*, *content*) pair in $modified_B$:

- If *id-map* contains *id*, *id-map*[*id*].*last-modified* $\sim \mathrm{VV}_B$, and the sentence contents are not equal, then ask the user to resolve the conflict
- Else if *id-map* contains *id*, *id-map*[*id*].*last-modified* $< \mathrm{VV}_B$, and the sentence contents are not equal, then set *id-map*[*id*].*content* = *content*
- Else if *deleted-set* contains *id* and *deleted-set*[*id*].*last-moved* $\sim \mathrm{VV}_B$, then ask the user to resolve the conflict
- Else—given our representation, this case will never occur

2.2.5 Resilience to Failures

Context ensures that *all-or-nothing* atomicity with merges, even in the face of failures. Instead of merging changes directly into the doc-list, we create a new temporary linked list (*temp-list*), copy all the information over, and merge changes with the temp-list. We store the new document version vector in the dummy head node of temp-list and, atomically, point doc-list to temp-list after the merge completes. This is functionally equivalent to creating shadow files. However, it increases performance by reducing the number of disk seeks necessary. We do not allow users to edit the document while this takes place. We justify this decision by noting that the average running time is about 10 ms per *update*, since this part of the merge process is not dependent on the network (see §3.2).

2.2.6 Deleted-set Recreation

The deleted-set is necessary to track deletes for merges, but it can grow arbitrarily long. Instead of imposing rules or storing additional state to know when to prune, we allow the Context implementation or user to reduce the size of the deleted-set, by removing its oldest entries, whenever he wants. This design decision means that Step 3's $deleted_B$ might not be correct if (i) B's deleted-set no longer has the entries for sentences deleted far back and (ii) A has not merged

To make sure that our merge procedure really works, we implemented it as a shell text editor in Java. Visit http://mit.edu/das/www/context/ to download and run this Context prototype.

in a while. So, during *send*, if B sees that $VV_A \precsim$ the minimum version vector in B's deleted-set, then he sends a set B of all sentence IDs in his doc-list to A. Then, A reconstructs the missing deleted-set—every ID in his own doc-list but not in B—and treats it as $deleted_B$ during the *update* step. Thus, the original merge procedure still works with this extra check.

2.3 Commit

We introduce the idea of *commit points*, which allows users to verify the consistency of a target document amongst all users. A user can initiate a commit point by designating a commit-name for a target document. The commit will only succeed if the committed document reflects the changes from all users. Context uses a two-phase commit protocol to support this feature.

2.3.1 Storage and Processing

Temporary and committed documents. All files on disk are stored as Context Files. We store committed documents with a `.ctx` extension and temporary documents with a `.ctx.tmp` extension. Each Context File has an associated *committed* folder, which contains any temporary and committed documents.

Sequential processing. Context uses a single thread to handle all changes and commits of a document. This means that all data received from the network and changes from the user are sequentially processed. If we later add multithreading to increase efficiency, then we will have to introduce read and write locks on the document.

2.3.2 Commit Protocol

As mentioned above, Context uses a two-phase commit protocol to support commit points. The *voting phase* verifies that all users agree on the current contents of a document. The *completion phase* propagates the commit results to all users in the network. Figures 5 and 6 demonstrate two possible outcomes of this commit protocol.

Voting phase – coordinator. When a user initiates a commit, we refer to him as the *coordinator* and all other users as *voters*. The coordinator starts the process by selecting a unique commit-name, e.g., "Final" (we use this name as reference for the remainder of this section). Context verifies that the commit-name is unique in the committed folder (neither `Final.ctx.tmp` nor `Final.ctx` exist). If the name is unique, then Context copies the contents of the document into `Final.ctx.tmp`. The coordinator's IP address is also stored within this temporary file to indicate that he was the initial committer. This temporary file is used so that the coordinator can make changes to his document while the commit is pending. Finally, the coordinator sends a *prepare* message to all voters. This message contains the coordinator's IP address, the commit-name, and the document version vector.

Voting phase – voters. Each voter will now need to perform two checks. The first check compares the coordinator version vector (VV_c) to the voter's version vector (VV_v), and there are three possible cases:

1. $VV_c \overset{\sim}{<} VV_v$ voter *rejects* the commit since the coordinator's document is not up-to-date
2. $VV_c > VV_v$ the voter requests a merge with the coordinator's temporary file, `Final.ctx.tmp`, then proceeds to the second check
3. $VV_c = VV_v$ the voter proceeds to the second check

The second check requires the voter to validate the uniqueness of commit-name within its own committed folder. Again, there are three possible cases:

1. Committed folder contains `Final.ctx`—the voter *rejects* the commit. The voter also sends the contents of `Final.ctx` to the coordinator.
2. Committed folder for `Final.ctx.tmp`—the voter replies *reject* if the IP address associated with the existing temporary file comes numerically before the coordinator's IP address. Otherwise, it overwrites the temporary with the target document and *accepts.*
3. Commit-name is unique—voter copies the contents of the target document into `Final.ctx.tmp`. The voter *accepts.*

The numerical ordering of IP addresses in the temporary file case provides a natural tiebreaking procedure in order to avoid concurrent commits under identical commit-names.

Context will allow for a window of 30 seconds (adjustable) between the initial transmission of an instruction and the response before the commit is said to timeout. The coordinator handles timeouts by aborting the initiated commit, and voters handle timeouts by deleting their local temporary file.

Completion phase – coordinator. The coordinator waits for all replies from the voters and decides whether or not to finalize the commit.

- If any voter rejects the commit, then the commit is aborted. If any voter does not respond for more than 30 seconds, then the commit is aborted. The coordinator broadcasts an *abort* message to all voters, instructing them to discard the attempted commit. The temporary file, `Final.ctx.tmp`, is then removed from the coordinator's committed folder.

- If all voters accept the commit, then the coordinator finalizes the commit by renaming `Final.ctx.tmp` to `Final.ctx` (assumed to be an atomic operation). This is the commit point. The coordinator finally sends a *commit* message to all voters, instructing them to finalize their temporary commits as well.

Completion phase – voters. Each voter responds to the instructions received from the coordinator. If the voter is instructed to *commit*, then it renames `Final.ctx.tmp` to `Final.ctx`. If the voter is instructed to *abort*, or does not receive a message for 30 seconds, then it deletes the *temporary* file from the committed folder.

2.3.3 Routing

Context allows commits to succeed even if the coordinator is not directly connected to each voter. A simple routing procedure is used to forward *prepare* and *commit* messages to all users. We will refer to the connection between any two "online" peers as a direct connection as well. Before the coordinator sends a message, he creates a set of IP addresses, *contacted-users*, and adds all users he is directly connected to. He appends *contacted-users* to the message before sending. Each voter is responsible for forwarding the message to any directly connected users not listed in *contacted-users*, by adding these users to *contacted-users* prior to sending. This ensures that messages get propagated through the network. Voters respond and react to all messages the same way, even if they are duplicates.

Context ensures that voter responses are correctly routed back to the coordinator by sending each response backwards along the same path. If the network topology changes *during* this commit process, then it is *possible* that the commit will fail.

2.3.4 Resilience to Failures

Context's commit system guarantees that each user will either be able to commit a file under a certain name, or have access to the file that was committed under that name. We ensure this capability even with possible machine failures.

Each user attempts to maximize consistency upon crash recovery by pinging other users for information. Specifically, the crashed user looks through each temporary file in his committed folder, and asks other users if that file was committed. Users respond *yes* if that *committed* file exists in their committed folder. If any user responds *yes*, then the crashed user renames the temporary file to the *committed* file. Otherwise, the temporary file is deleted.

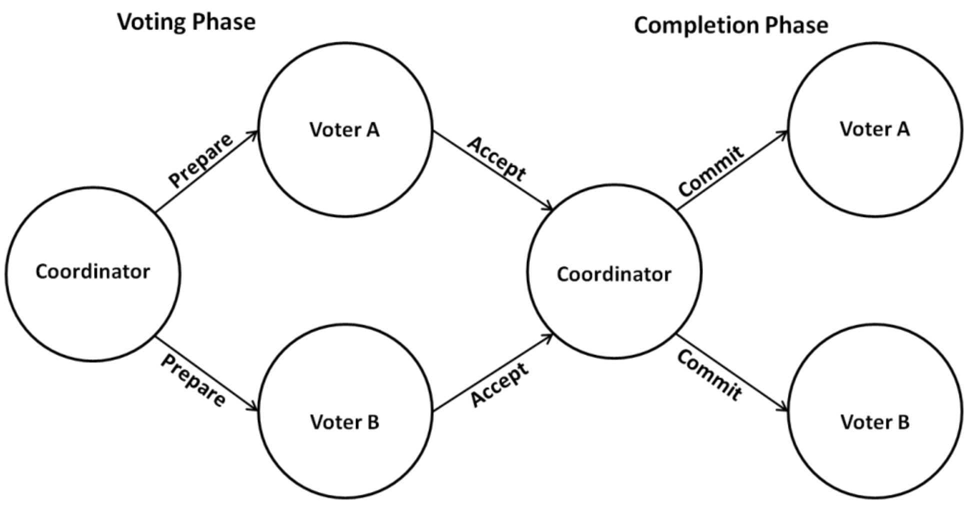

Figure 5. *Flowchart depicting Context's two-phase commit protocol during a successful commit.* It should be noted that the voters do not acknowledge the coordinator's request to commit because, in light of failures, they can simply re-ask the coordinator for the state of the initiated document. Additionally, correctness is not compromised since any attempts by a voter to recommit under the same commit-name will be rejected by the coordinator.

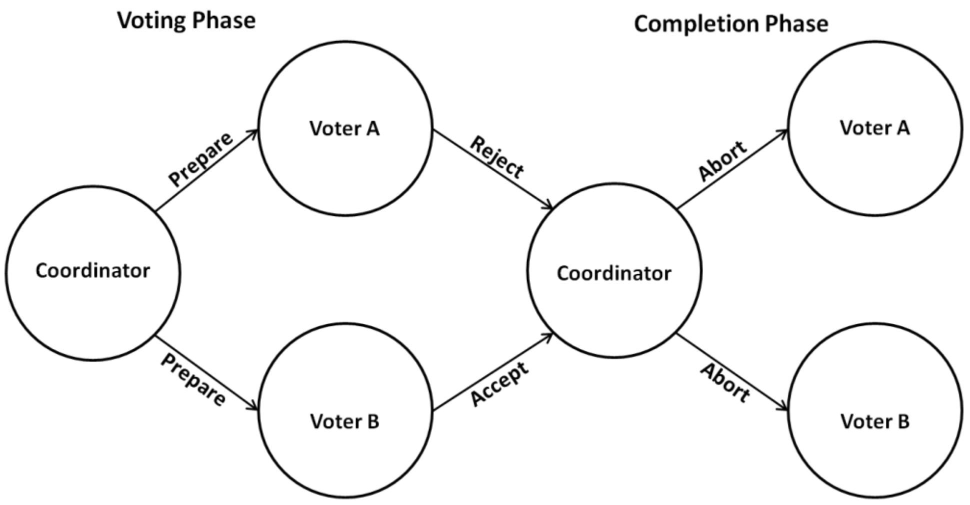

Figure 6. *Flowchart depicting Context's two-phase commit protocol during a failed commit.* Again, voters do not acknowledge the coordinator's request to commit because, in light of failures, they can simply re-ask the coordinator for the state of the initiated document. Additionally, correctness is not compromised since the voters will delete the temporary file upon recovery (no user has the finalized commit), and a voter can then reinitiate a commit under that commit-name.

3 Analysis

3.1 Conflicts

Context minimizes the number of conflicts reported to the user by separating modifications from moves, and using relative sentence locations rather than absolute locations. In particular, Context only reports a conflict in the following cases:

- Multiple users modify the same sentence with different changes
- Multiple users move the same sentence to different relative locations
- A user moves a sentence *s* in between sentences *r* and *t*, but another user moves both sentences *r* and *t* to different locations (no longer next to each other)
- One user deletes a sentence, while another user modifies or moves the same sentence

Context does not allow changes to be dropped silently. Thus, raising a conflict is the correct system response in the four cases described above.

We will discuss several use cases and show in detail why Context produces the correct result. Suppose there are three users, Alice, Bob, and Charlie, and each starts with the same copy of the document and version vector ⟨ (Alice, 1), (Bob, 1), (Charlie, 1) ⟩ (denoted as ⟨1, 1, 1⟩ in the following cases). For simplicity, assume that the user whose name comes first alphabetically first has control over merging conflicts.

We will refer to *s*(*i*, User) as the *i*th sentence in User's document. For each case, we show the appropriate version vectors and how they change throughout the process. Only changed version vectors are shown.

Case 1. Bob and Alice both modify sentence 2, but each sets the contents to "This is a sentence." Even though *s*(2, Alice).*last-modified* ~ VV_{Bob}, no conflicts are reported upon merging since they have the same contents.

	Original	**Bob modifies**	**Alice modifies**	**Alice updates with Bob's changes**	**Bob updates with Alice's changes**
VV_{Alice}	⟨1, 1, 1⟩		**⟨2, 1, 1⟩**	**⟨2, 2, 1⟩**	
VV_{Bob}	⟨1, 1, 1⟩	**⟨1, 2, 1⟩**			**⟨2, 2, 1⟩**
s(2, Alice).*last-modified*	⟨1, 1, 1⟩		**⟨2, 1, 1⟩**		
s(2, Bob).*last-modified*	⟨1, 1, 1⟩	**⟨1, 2, 1⟩**			

Note that while the document version vectors are updated, *last-modified* version vectors are not changed.

Case 2. Bob modifies sentence 3, Alice modifies sentence 1, and they both simultaneously change sentence 2 to have different contents. Prior to merging, sentence 2 is the only sentence with $s(2, \text{Bob}).\textit{last-modified} \gtrapprox \text{VV}_{\text{Alice}}$ and $s(2, \text{Alice}).\textit{last-modified} \gtrapprox \text{VV}_{\text{Bob}}$. All other sentences have either $s(i, \text{Bob}).\textit{last-modified} \leq \text{VV}_{\text{Alice}}$ or $s(i, \text{Alice}).\textit{last-modified} \leq \text{VV}_{\text{Bob}}$. So, upon merging, only Alice is asked to resolve the content conflict in sentence 2.

	Original	**Bob modifies**	**Alice modifies**	**Alice updates with Bob's changes**	**Bob updates with Alice's changes**
VV_{Alice}	⟨1, 1, 1⟩		**⟨2, 1, 1⟩**	**⟨3, 2, 1⟩**	
VV_{Bob}	⟨1, 1, 1⟩	**⟨1, 2, 1⟩**			**⟨3, 2, 1⟩**
$s(1, \text{Alice}).\textit{last-modified}$	⟨1, 1, 1⟩		**⟨2, 1, 1⟩**		
$s(1, \text{Bob}).\textit{last-modified}$	⟨1, 1, 1⟩				**⟨3, 2, 1⟩**
$s(2, \text{Alice}).\textit{last-modified}$	⟨1, 1, 1⟩		**⟨2, 1, 1⟩**	**⟨3, 2, 1⟩**	
$s(2, \text{Bob}).\textit{last-modified}$	⟨1, 1, 1⟩	**⟨1, 2, 1⟩**			**⟨3, 2, 1⟩**
$s(3, \text{Alice}).\textit{last-modified}$	⟨1, 1, 1⟩			**⟨1, 2, 1⟩**	
$s(3, \text{Bob}).\textit{last-modified}$	⟨1, 1, 1⟩	**⟨1, 2, 1⟩**			

Alice increments her version number by 1 when she updates because she resolves the conflict in sentence 2. Bob accepts all changes from Alice since Alice has a higher version vector.

Case 3. Alice and Bob are connected, but Charlie is offline. Bob modifies sentence 3, and Alice and Bob then merge, and Alice disconnects. Offline, Charlie changes sentence 3 in a different way, and then merges with Alice. Alice resolves the conflicting sentence, and Charlie accepts her decision. When Bob merges with Charlie, there is no conflict to resolve. This process is depicted in Figure 7.

Case 4. Bob modifies the contents of sentence 2, while Alice moves sentences 2 to the end of the document. When Alice receives Bob's changes, $s(2, \text{Alice}).\textit{last-modified} < \text{VV}_{\text{Bob}}$, so she modifies her sentence. When Bob receives Alice's changes, $s(2, \text{Bob}).\textit{last-moved} < \text{VV}_{\text{Alice}}$, so he moves his sentence. There are no conflicts.

	Original	**Bob modifies**	**Alice moves**	**Alice updates with Bob's changes**	**Bob updates with Alice's changes**
VV_{Alice}	⟨1, 1, 1⟩		**⟨2, 1, 1⟩**	**⟨2, 2, 1⟩**	
VV_{Bob}	⟨1, 1, 1⟩	**⟨1, 2, 1⟩**			**⟨2, 2, 1⟩**
$s(2, \text{Alice}).\textit{last-modified}$	⟨1, 1, 1⟩			**⟨1, 2, 1⟩**	
$s(2, \text{Alice}).\textit{last-moved}$	⟨1, 1, 1⟩		**⟨2, 1, 1⟩**		
$s(2, \text{Bob}).\textit{last-modified}$	⟨1, 1, 1⟩	**⟨1, 2, 1⟩**			
$s(2, \text{Bob}).\textit{last-moved}$	⟨1, 1, 1⟩				**⟨2, 2, 1⟩**

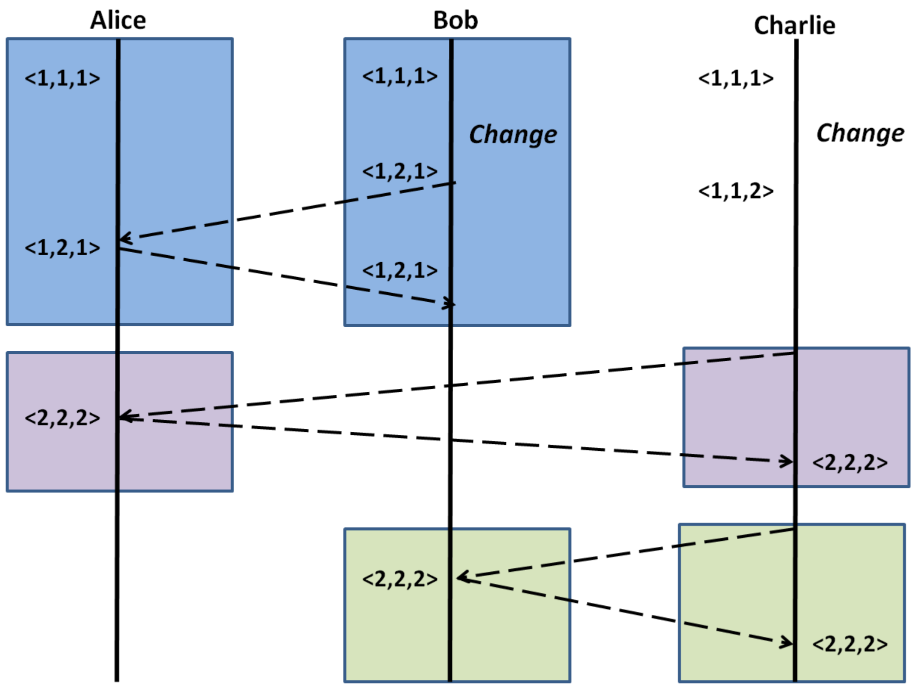

Figure 7. Timeline depicting document synchronization and the updated version vectors for Alice, Bob, and Charlie in Case 3. In the blue, purple, and green regions: Alice and Bob, Alice and Charlie, and Bob and Charlie are online, respectively. The directed dashed lines indicate which individual is receiving document updates.

3.2 Availability

Context minimizes the time in which users cannot edit the document. Users may not edit the document while the local copying and merging takes place. Let us assume that disk seeks take about 10 ms, and we can write sequentially to disk at 50 MB/s. If the average file size is about 50 KB and the average number of users editing a file is 10, then the total time that a user cannot edit the document every second is $10 \cdot [10 \text{ ms} + 50\text{KB} / (50\text{MB/s})] = 110$ ms. This means that Context is available 90% of the time. Though this is reasonable (and unobservable for most users), this is an area that we are looking to improve. If availability is important to the user, then a merge period of 11 seconds gives 99% availability.

3.3 Network Performance

Context minimizes the amount of bandwidth used by sending only the necessary information across the network. Since sentence delimiters are user-specified, they will generally result in short sentences (approximately 100 characters, or 100 bytes). We assume that a typical user will modify or move one sentence per second, on average. To analyze the typical amount of bandwidth used by any particular user, we also assume that a typical document has 10 users concurrently editing it. Each version vector will then be approximately 10 users · (4 bytes / IP + 4 bytes / integer) = 80 bytes. Each sentence ID is 8 bytes. Adding up the cost of each step in the merge protocol and multiplying it by the number of users, we estimate the average upload and download bandwidth used to be on the order of 10 · (80 bytes + 100 bytes + 8 bytes) = 2 kB/s.

3.4 Storage

Context attempts to minimize total storage on disk, while still maintaining enough information to track changes. Context does not maintain a log, so the total space used by the document is linear in the amount of text in the document. Though the deleted-set can grow arbitrarily large, we prune it a manageable size when necessary. Even though the recreation process requires extra network usage, users are (i) not likely to continuously delete sentences, (ii) not likely to disconnect from other users for long periods of time before reconnecting, and (iii) not likely to do both i and ii simultaneously. Therefore, assuming that an average file has about 500 sentences (100 bytes each), and the deleted-set has about 100 sentences, then the average Context File is about (500 sentences · (100 bytes / content + 8 bytes / ID + 2 · 80 bytes/version-vector) + 100 sentences · (8 bytes / ID + 80 bytes / version vector) = 142 kB. This is reasonable compared to the 50 kB necessary just to store the plain text.

3.5 Scalability

Context is a peer-to-peer system, so the total network usage is divided amongst the different users. However, since each user is responsible for communicating with all other users, there are practical limits on the total number of people editing the same document. If 100 people edit the document, then the average upload and download bandwidth would be more than 200 kB/s. However, to decrease network usage, we give the user the ability to adjust the merge period.

Storage becomes infeasible as the number of users increase. If 100 people edit the document, then the average size of the document becomes almost 1 MB, even though the total amount of text is just 50 KB. To alleviate these storage problems, we may look into reducing the size of version vectors; such techniques are discussed in [2] and [3].

3.6 Failures

3.6.1 Editing

Machine failures while the user is editing the document will result in a minimal loss of recent changes. All data is written directly to disk, so the amount of data lost is dependent on how the system handles disk writes.

3.6.2 Merging

Context guarantees all-or-nothing behavior while merging documents. All changes are first merged into a temporary list (functionally similar to a shadow file). If the machine crashes before the atomic overwrite of the doc-list pointer, then the merge will not be visible to the user (and the version vector will not be updated). If the machine crashes after overwriting the doc-list pointer, then the merge will be visible to the user upon recovery (and the version vector will be updated). The document version vector is stored in the head dummy node, and the tail pointer of doc-list is removed, to allow for these atomic overwrites.

3.6.3 Committing

We will now discuss several failure scenarios during the commit process and how Context handles them correctly. In all cases, once a file has been committed, it may never be committed again. Each user may access a committed file either by looking in the committed folder, or asking another user. If a file has been committed, at least one user is guaranteed to have a copy (even if all machines crash afterwards).

Coordinator failure. Suppose a coordinator `A` wants to commit a file with the name "Final." The coordinator can fail in the following three places:

Crash…	**Recovery**
Before sending prepare	Commit failed. No users know about the attempted commit, and the coordinator's temporary file is deleted.
Before renaming	Commit failed. All users abort the commit because of timeout. Upon recovery, coordinator `A` asks other users about committed state of "Final," and eventually deletes `Final.ctx.tmp`.
After renaming	Commit succeeded. Some users may not have received commit messages. If another user becomes coordinator `B` and attempts to recommit "Final" (after the timeout), coordinator `A` rejects the commit, and sends `Final.ctx` to `B`.

Voter failure. Suppose coordinator `A` attempts to commit a file with name "Final," and voter `B` is one of the voters. The voter can fail in the following three places:

Crash…	**Recovery**
Before receiving prepare	Commit times out and fails.
After creating temporary file but before sending accept	Commit times out and fails. Upon recovery, voter `B` asks other users about committed state of "Final," and eventually deletes `Final.ctx.tmp`.

Crash…	Recovery
Before renaming `Final.ctx.tmp` → `Final.ctx`	Upon recovery, voter `B` asks other users about committed state of "Final." If any user responds *yes*, then voter `B` renames `Final.ctx.tmp` to `Final.ctx`. Otherwise, voter `B` deletes `Final.ctx.tmp`. *Suppose the commit succeeded.* Voter `B` can attempt to recommit "Final," but coordinator `A` will reject the commit and send `Final.ctx` to `B`. If `A` and `B` are not connected, then the commit will fail regardless. *Suppose the commit failed.* Voter `B` is able to recommit under "Final," since `Final.ctx` does not exist for any user.

Even with multiple simultaneous failures, Context guarantees either access to the committed file, or the ability to initiate a recommit. It is possible that, given a particular network partition and sequence of machine failures, access to a committed file may not be immediately available. However, any attempts to recommit this file will fail, and eventually, all users will gain access to this file.

4 Conclusion

In summary, Context offers fully functional support for a peer-to-peer concurrent text editor. The document-sharing application minimizes conflicts, supports intermittent connectivity, and provides reliable merge and commit services. Future directions for this work include handling dynamic group membership and improving Context's scalability (network bandwidth, disk space, and speed are major factors to consider in this area). Additional considerations that must be handled in the future are security concerns like user authentication and document confidentiality.

5 References

[1] R. Strandh, M. Villeneuve, T. Moore, "Flexichain: An Editable Sequence and Its Gap-Buffer Implementation," 1st European Lisp and Scheme Workshop, 2004.

[2] D. Malkhi and D. Terry, "Concise Version Vectors in WinFS," International Symposium on Distributed Computing, 2005.

[3] Y.-W. Huang, P. S. Yu, "Lightweight Version Vectors for Pervasive Computing Devices," Proceedings of the International Workshop on Parallel Processing, 2000.