Design Project 2: ReWordi

A Collaborative Peer-to-peer Text Editor

Mary Linnell	mlinnell@mit.edu
Bridger Maxwell	bridger@mit.edu
Jennifer Wang	jewang@mit.edu
Rudolph 11 am	
May 10, 2012	

Overview

This document describes a design for a collaborative peer-to-peer text editor, called ReWordi. The text editor will allow multiple people to edit the same document at once even when disconnected, providing for sensible merging when conflicts arise between different versions of the same document.

Designed for small user groups, ReWordi aims to be lightweight and favors correctness over performance. ReWordi is built on top of git's remote references set-up and relies on git to handle data (git fetches, git diffs, and git merges). ReWordi transfers data entirely through pulls; there are no git pushes. To customize git, ReWordi's design has two major features. First, ReWordi stores files in a special text document structure to support automatic interfacing to git diffs. Second, ReWordi uses a specific change propagation protocol that picks leaders who behave like temporary servers and that determines how users interact.

The following summarizes ReWordi's major design decisions. ReWordi:

- *Uses git for merging changes and recording commits*
  Git provides a powerful system of distributed content synchronization.

- *Separates paragraph ordering from paragraph content*
  This streamlines and minimizes ambiguity during diffs, even though it requires slightly more memory and disk space.

- *Allows only the leader to merge changes*
  This prevents redundant conflict merging and bypasses the alternative of a complicated system of locks, sacrificing some performance.

- *Allows only one leader in a group of connected users*
  This simplifies ReWordi's operation and greatly helps understanding network traffic, sacrificing some performance.

- *Uses a two-phase commit and requires all users to be online for commits*
  This ensures the reliability of successful commits with little overhead, sacrificing scalability to large user groups.

Design

The goal of ReWordi's design is to be able to track changes accurately, to automatically merge changes whenever possible, and to minimize the number of conflicts which must be resolved manually when automatic merging is not possible.

ReWordi uses git as a backbone for transmitting document content and for merging changes across users. This design decision was made because git was designed for collaboration between several disconnected repositories, where any repository can interact with any other repository. This allows ReWordi to work when any subset of users are connected to each other. Additionally, git is fast and efficient. Committing changes to git or merging changesets are fast enough to be non-disruptive. Git has its own protocol for compressing and transmitting changesets over TCP.

Document Structure

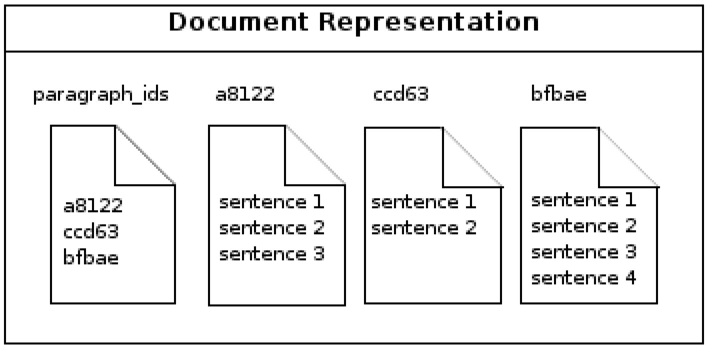


Figure 1. *An example of the internal structure of a document.* A text document has one paragraph_ids file that lists paragraph ID numbers. Each of the document's paragraphs has a paragraph_text that contains the actual sentences.

In order to effectively track changes, ReWordi represents text documents as the sum of individual paragraphs, and each paragraph as the sum of individual sentences. *Paragraphs* are delimited by hard returns. *Sentences* are delimited by periods, exclamation points, or questions marks. On each user's local disk, ReWordi persists the text document as a series of files: each document has one *paragraph_ids* file for the entire document and a *paragraph_text* file for each paragraph. See Figure 1.

The paragraph_ids file stores the paragraph ordering of the text document. When a new paragraph is created, it is assigned a universally-unique identifier (UUID). This ensures that the paragraph id is unique without coordinating with all other users. The paragraph_ids file keeps a list of all of the paragraph ID numbers, tracking the order that the paragraphs appear in the document. If a paragraph is moved in the document, then the list file is modified to reflect the new order of paragraphs. In this way, moving a paragraph is done through line edits in the paragraph_ids file.

The paragraph_text files store the actual words and content of the text document. ReWordi parses each paragraph into a list of sentences. For each paragraph, a new paragraph_text file is created (named as its unique ID), and each line of the file contains one sentence (that is, a new line character follows each sentence). The purpose of this structure is to be able to easily run a *diff* command between versions to determine which sentences have changed.

Diff

ReWordi uses and extends git's native diff operation to detect changes. Git diffs are adequate to detect most paragraph content and paragraph ordering change cases. However, there is one special case that a git diff will not pick up, so ReWordi performs an extra check after a git diff.

In ReWordi, the unit of comparison is the sentence (two users cannot edit the same sentence without a conflict). In git, the unit of comparison is the line. For paragraph content, ReWordi's document structure automatically interfaces between sentences and lines so ReWordi can use the native git diff operation: ReWordi stores each paragraph in separate paragraph_text files, and inserts a line break for tokens that end a sentence. Similarly, for paragraph ordering, ReWordi's document structure interfaces between paragraphs and lines: ReWordi stores each paragraph ID number on a separate line in the paragraph_id file.

By separating paragraph ordering in a paragraph_ids file and paragraph content in paragraph_text files, paragraph ordering and paragraph content changes are isolated. As a result, paragraphs can be reordered by one user, and modified by another user concurrently without conflict.

Some sentences may be incorrectly split into two lines in the representation if the sentence contains a period (for example, "Mr. Jones loves ReWordi.") This means more than one edit can be allowed in a single sentence without a conflict. This is an acceptable side-effect, but could later be fixed with a more intelligent grammar tokenizer.

ReWordi extends git diffs by implementing a special check after running git diff to detect paragraph reordering conflicts. Moving a paragraph is done by deleting the line in the paragraph_ids file, and inserting it in a new location. If two users were to move the same paragraph to two different locations, a git diff would recognize this as a single deletion, and *two* insertions of the same line in different locations. Git has no knowledge of what the paragraph_ids file is used for, so these edits are seen as non-conflicting. For this reason, ReWordi performs an additional check to ensure that a merge did not result in duplicate lines in the paragraph_ids file. If duplicate lines are found, this is flagged as a conflict and must be resolved by keeping only one of each duplicate line.

Propagating Changes Across Multiple Users

Each user has a git repository that stores the history of the document. When the document is first created, each repository has a git remote reference to every other repository for that document. Periodically as a user edits the document, these edits are stored in the local repository as a git commit, both when the user is online and offline. When more than one user is online, the git commits need to be propagated. ReWordi stores document history, currently doesn't expose past versions to the user. The history is only used for the purpose of merging changes.

Every document's group of users will have a chain of command, which will be set when the document is created. When more than one user is online, the user that is highest in the chain of command becomes the leader. To determine this, each user machine holds a copy of the chain of command and will periodically perform the following steps over TCP. These steps are performed every few minutes:

1. Poll for who is online starting from the highest authority
2. If the user reaches itself first, then it takes action as a leader.
3. On the other hand, if the user identifies another machine first, the user behaves as a peer and initiates a git fetch from the identified leader.

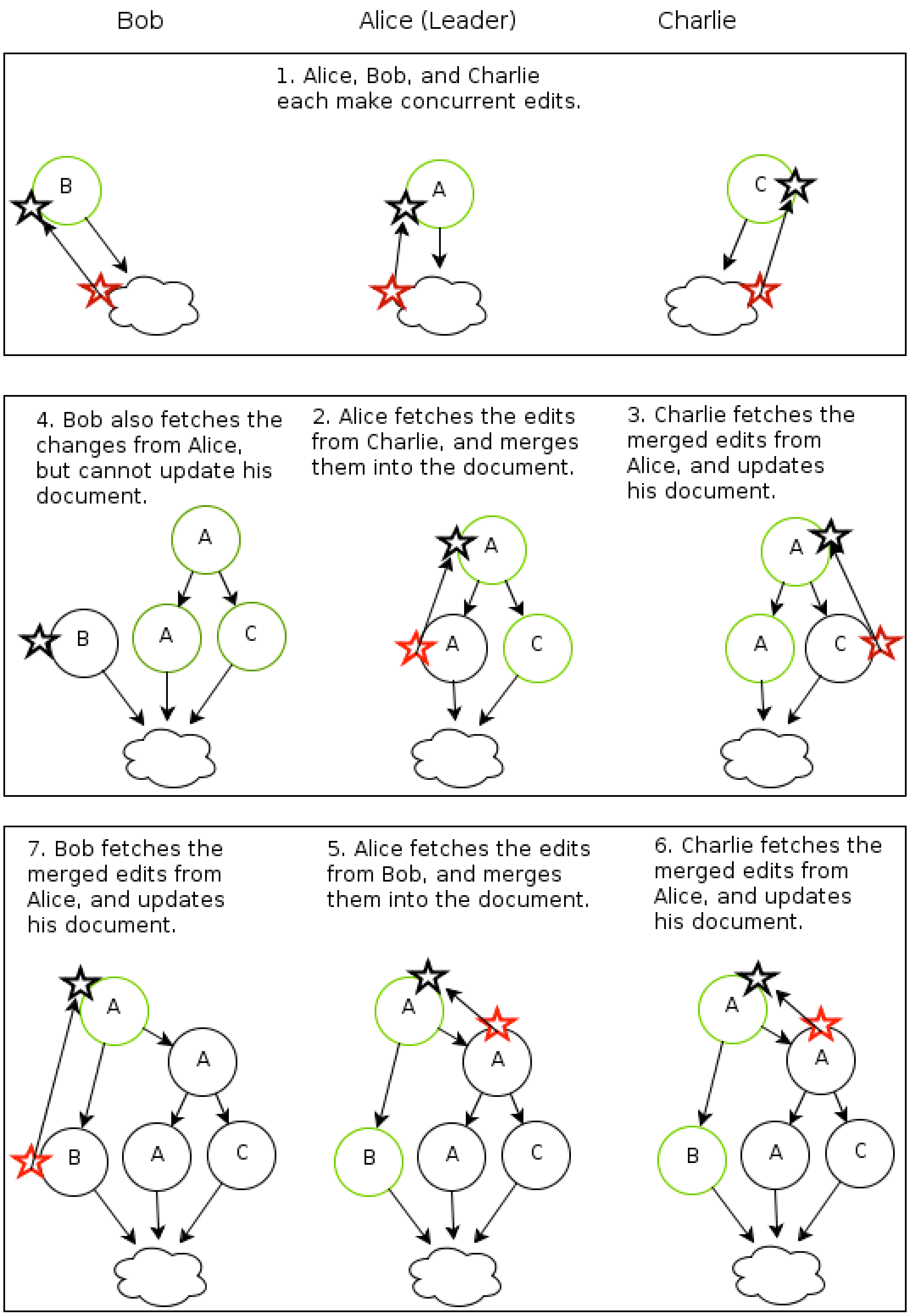
Using the git protocol, the leader fetches changes from all other online users, one at a time. Every time the leader pulls a new commit, the leader runs a ReWordi diff, merges the changes on his or her computer, then commits the merge to his or her git repository, allowing the updated document to be fetched by all other users.

When ReWordi runs a git fetch to retrieve changes from another user (the remote user), the only change is adding git commit objects and tag objects to the local git repository (no changes are made to the working document). There are two categories of commits:

The first category is a commit that is a descendent of the current working commit. This means only the remote user has made changes to the document since the last reconciliation. In this case, ReWordi can safely "fast-forward" the local user's working document to the newly fetched commits.

The second category is a commit that has diverged from the current working commit. This means that the two commits have happened concurrently since the last reconciliation. In this case, the newly fetched commit must be merged with the working commit before the fetched commit's changes will show up in the working tree. Only the leader performs merges. The merge results in a commit that is a descendent of both commits, so when other users fetch from the leader, their working trees can be fast-forwarded to the merged commit.

This reconciliation scheme works when any subset of users is online either through the Internet or with direct connectivity. The scheme also works when a large group of users subdivide into small groups that work together separately, and later reconvene into a large connected group. Take, for example, a situation in which user A has reconciled with user B, and user C has reconciled with user D in separate networks (letter order reflects chain of command). When all users join the same network, user A will pull changes from either user C or user D and create a tree that contains all users' edits. Users B, C, and D can then pull this updated tree from user A, and all users will then be reconciled. See Figure 2 for an example of change propagation.



Bob
Alice (Leader)
Charlie
1. Alice, Bob, and Charlie each make concurrent edits.
B
A
C
4. Bob also fetches the changes from Alice, but cannot update his document.
2. Alice fetches the edits from Charlie, and merges them into the document.
3. Charlie fetches the merged edits from Alice, and updates his document.
7. Bob fetches the merged edits from Alice, and updates his document.
5. Alice fetches the edits from Bob, and merges them into the document.
6. Charlie fetches the merged edits from Alice, and updates his document.

Figure 2. *An example of ReWordi's change propagation protocol in action.* All three users make concurrent edits. Then Alice fetches and merges the edits from Charlie, since Alice is the leader. Then, Charlie fetches from Alice. When Bob fetches from Alice, he can't update yet. First, Alice must fetch from Bob, and

merge. Then, Charlie and Bob can both fetch from Alice and update. Note that the star symbol indicates the current HEAD of the user’s repository.

Commits

The purpose of a commit is to mark the current version of the document as a “final” version after ensuring that all group members agree on that specific version of the document. A commit is represented by a git tag containing a user-defined name pointing to the specific git commit. ReWordi ensures that tags can only be added to a commit if there are neither unsynchronized edits nor existing commits with the same name.

Our design uses a two-phase commit. First, one user machine (the initiator) initiates a commit on its latest version, “head,” and notifies the other machines about the operation with the name of the commit and git hash of the current version. Once a commit is initiated, the initiator's current version is frozen until the two-phase commit ends. For each other machine, if its latest version is the same as “head” (the git hashes are the same) and there does not exist an existing commit by the same name, the machine sends back a “yes”; else “no”. All machines must agree on “yes” for the operation to proceed; otherwise the commit is aborted. This means all users must be online. Second, as soon as the initiator receives a “yes” from every machine, ReWordi creates a git tag on “head.” A git tag present in the repository indicates a successful ReWordi commit. Successful ReWordi commits are eventually propagated to all other users during reconciliation, because tags are included with the changes retrieved with a git fetch.

If all users don’t reply with an appropriate “yes” or “no” within a timeout period, the commit fails. The initiator’s document is frozen until a commit finishes, so the timeout should be short enough that it wouldn’t significantly disrupt the user. The timeout should be long enough to contact users with a long round trip packet time. We estimate 15 seconds will be appropriate.

Analysis

Scenarios

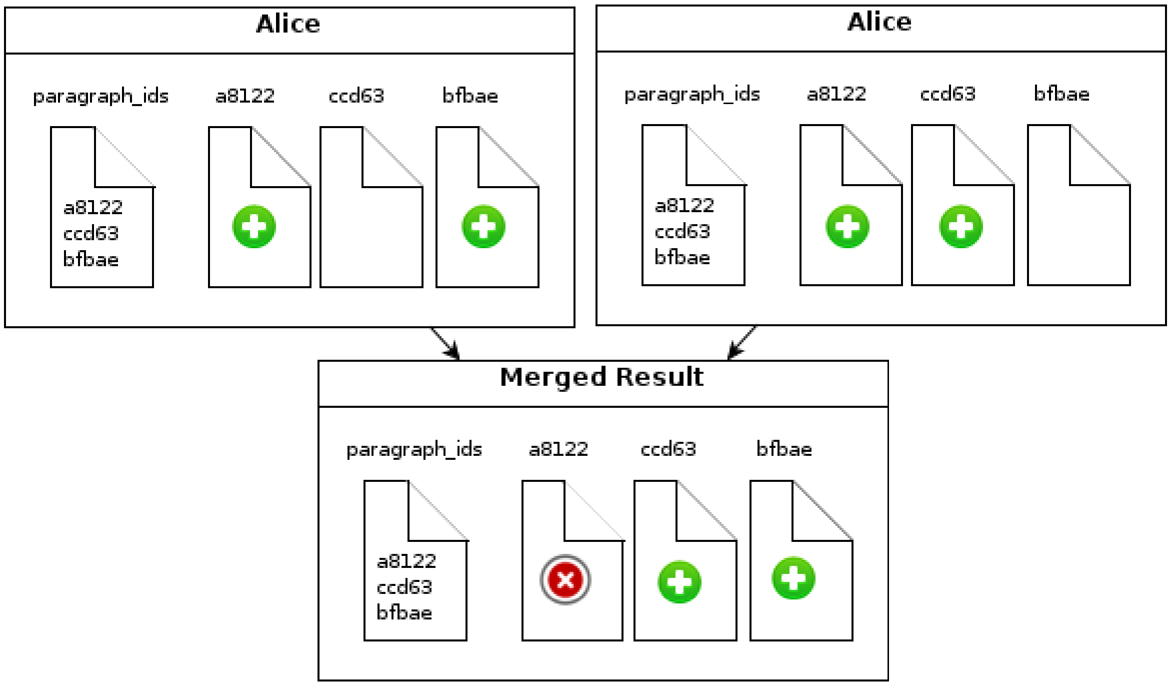


Figure 3. Illustration of the first scenario. The red X indicates that a conflict must be resolved manually.

In the first scenario, two users add text to different paragraphs, but they also both change the same sentence in the first paragraph. Since our design tracks different paragraphs separately, the first part, adding text to different paragraphs, can be automatically merged. Each of those edits were in different files. Then for the second part, the diff detects that two users have tried to modify the same sentence and requests a manual conflict resolution. See Figure 3.

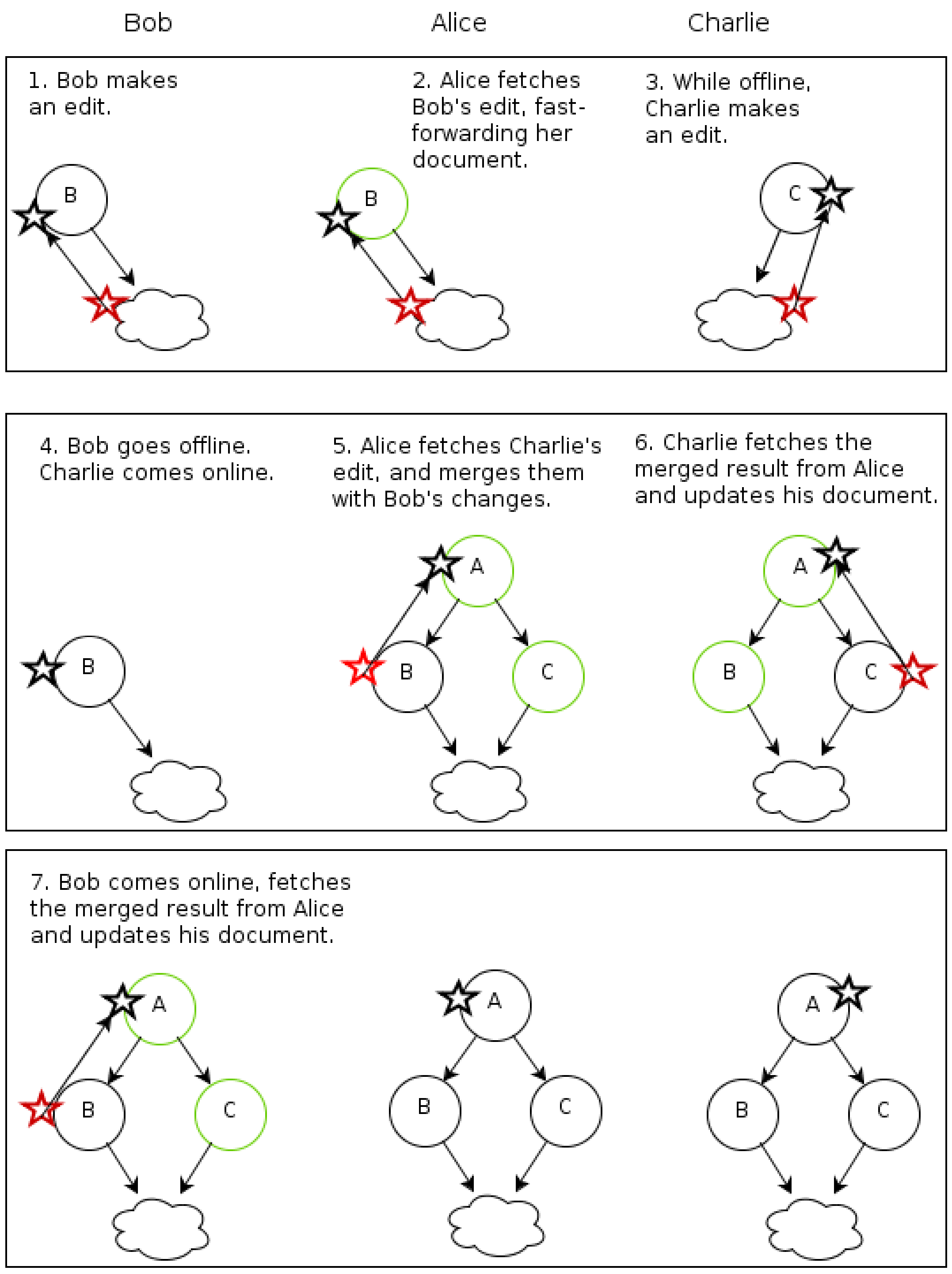

Bob
Alice
Charlie
1. Bob makes an edit.
2. Alice fetches Bob's edit, fast-forwarding her document.
3. While offline, Charlie makes an edit.
4. Bob goes offline. Charlie comes online.
5. Alice fetches Charlie's edit, and merges them with Bob's changes.
6. Charlie fetches the merged result from Alice and updates his document.
7. Bob comes online, fetches the merged result from Alice and updates his document.
A
B
C

Figure 4. Illustration of the second scenario.

In the second scenario, an online user Bob and an offline user Charlie both change the same sentence. Another user, Alice, synchronizes with Bob so that she has Bob’s changes. Later, Bob goes offline and Charlie comes online. Alice fetches Charlie's edit and merges them with

Bob's changes. Then, Charlie can fetch the merged result from Alice to update his document. Finally, when Bob comes back online, he can update his document by fetching from Alice. See Figure 4. Note that two conflict resolutions might be required in the case where Charlie disconnects from Alice after Alice has merged the changes from Charlie but before Charlie fetches the merged result, then Charlie meets up with Bob, though this is a slightly different scenario.

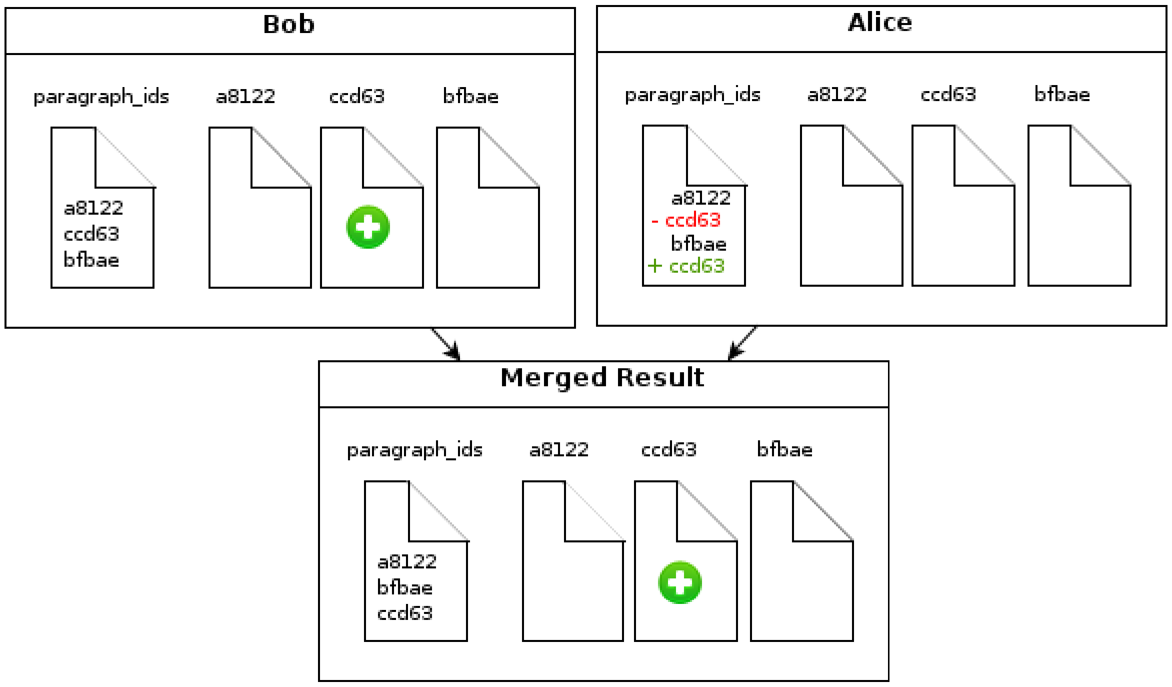


Figure 5. Illustration of the third scenario.

In the third scenario, an online user rearranges some paragraphs, and an offline user edits a sentence in one of the paragraphs. When syncing, there should not be any conflicts. In our design, rearranging the paragraphs would simply be changing the order of the IDs in the paragraphs file. Changing a sentence would update one of the paragraph files. Neither of these updates conflict, so all changes would merge automatically. See Figure 5.

In the fourth scenario, two offline users independently fix the spelling of one word in a sentence so that the two versions are the same. Our system will detect that the versions are the same because “diff” compares actual sentences and would not produce a conflict. The versions will then be merged together as usual with a git commit.

Update Reliability

Git commits and fetches are atomic, ensuring that ReWordi’s repository is kept consistent even after failures. Git fetches can only access data that has been git committed on disk and both git operations happen frequently; these traits guarantee that ReWordi can only silently drop changes if the user’s machine crashes before the user saves. In addition, even though leaders and peers perform actions asynchronously, the order that users interact is serializable because all actions involve interacting with the leader's repository, which is updated with before-or-after atomicity. For cases with reliable connections and a stable network, these two features are enough to provide change propagation reliability.

When the network is less stable or connections are less reliable, there are cases in which a conflict may need to be resolved more than once. One situation in which redundant conflict resolutions can occur is if the network is partitioned so that conflicting changes exist in two sets of users, and these two sets of users meet up in random, disorganized smaller groups. In this case, each smaller group will independently merge the changes and resolve conflicts.

One such example is if the users split into two groups. One group worked on an ad-hoc network in the park, the other group worked on an ad-hoc network in the subway. Then, without connecting to the internet, the groups randomly shuffled. One subset of the users worked on an ad-hoc network on a cruise ship, and the other subset of users worked on an ad-hoc network in a space shuttle. It is unreasonable and counter-productive that users would work so often in small, offline groups.

Another case that may cause redundant conflict resolutions is when the leader pulls a conflicting change from a user and merges the change, but the user goes offline (or changes leaders) before fetching the merge from the leader. In this case, that user may synchronize with another group of users, triggering the same conflict resolution. This is rare because the time between a merge and a user fetching that merge is small.

Each of the cases of redundant conflict resolution arise from rare use cases and failures. Fortunately, when users with redundant conflict resolutions synchronize, identical conflict resolutions are merged without conflict, and differing conflict resolutions trigger yet another conflict resolution. Essentially, these are seen as more independent edits. At worst, they require slightly more user intervention. Any discrepancies will always be resolved by commit time.

Commit Reliability

The use of the two-phase commit ensures that the documents across users stay consistent even with failures. Once a git tag is created, the commit is permanent. Until then, a failure on the initiator's machine simply results in the commit failing (but does not leave any side effects). If any other user's machines fail (or disconnect) before a "yes" is sent, the commit fails. If any other user's machines fail after a "yes" is sent, the commit will still be successful. Once they come online and fetch changes from another user, they will be aware of the commit.

Two commits with the same name but on different versions of the document can never succeed, because in order for a commit to succeed, all users must be (1) online, (2) on the same version of the document, and (3) no user can have a record of a commit by the same name.

Concurrent commits are not a problem. There may be a case in which two users initiate a commit under the same name at the same time and still succeed. However, if both commits succeed, there is still no inconsistency because both of the commits are ensured to be on the same version of the document. In git, two identical tags on the same version would be represented by the same object and will not cause conflicts when updating.

Scalability

ReWordi’s design prioritizes correctness over performance. Because there is one designated leader for each group of connected users, ReWordi does not scale well in terms of performance as the number of connected users increases. Leaders are not only responsible for all merges, but also behave as a hub from which all other machines fetch. However, this design sacrifice can be easily remedied for propagating changes because ReWordi supports subsets of connected users. In the future, a new layer could be written to create trees of users to virtualize the connects that ReWordi sees, yielding O(log(n)) propagation performance. This feature is left out for implementation simplicity.

Unfortunately, while change propagation could plausibly be scaled, commits cannot. ReWordi’s commits require all users to come online and remain running for the entire commit process. As the number of users increases, the rate of failures increases asymptotically exponentially. While ReWordi’s commit scheme is adequate for a small group of friends, successful commits will take more time with more users.

Summary

ReWordi allows multiple people to collaboratively edit a single document both online and offline, as well as to decide on a specific version of a file to commit. It (1) processes the document into paragraphs and then each paragraph into sentences and (2) has a flexible change propagation protocol to effectively use git. Overall, ReWordi presents a lightweight and reliable system to collaborate on a peer-to-peer network. Although scalability remains a future problem to be addressed, ReWordi works effectively for the small user groups.

Acknowledgements

Thank you to Travis for reviewing our preliminary design and finding error scenarios that we did not address.

Word count

3355