DecentralizedDocs : A Peer-to-Peer Text Editor

 Michael Plasmeier

*theplaz@mit.edu*

Nandi Bugg

*nbugg@mit.edu*

Rahul Rajagopalan

*rahulraj@mit.edu*

 Rudolph

April 26, 2012

**Introduction**

Users would like to collaborate on a text document without using a central server or needing to be online all the time. We propose the design of DecentralizedDocs, a peer-to-peer text editor, which fulfills this demand. DecentralizedDocs allows users to edit text offline and then reconcile the text with their teammates. It supports both written text and code.

DecentralizedDocs requires that each machine has a unique machine name.

# Data Structure

The basic unit of the document is a line. In code this is simply one line, but in a written text document, one line is equivalent to a paragraph when word wrap is enabled. Lines are broken up by *\n* characters. Line structures contain a reference to the text in the line, an index showing the line's position in the document, and a unique ID. This line ID is the hash of the current timestamp and machine name, making it effectively random. Lines have the following code:

struct Line {

 long id;

 char\* text;

 int position;

 VersionVector text\_version\_vector;

 VersionVector position\_version\_vector;

}

Each line has two version vectors associated with it; one for text and the other for position. A version vector contains a line revision numbers for each user. Version numbers start at 0 for a new line. When either text or position is changed, the corresponding version vector component is incremented by 1. Version vectors have this code:

struct VersionVector {

 int version\_counters[N]; // N is the number of collaborators

 // Position n corresponds with collaborator n

}

DecentralizedDocs stores documents in memory as linked lists of lines. To save documents on disk, it serializes the linked lists. Figure 1 shows the data structure of a document.



Figure 1. The data structure. The user sees a continuous block of text, but lines are stored in their own structures internally.

## Reconciliation

DecentralizedDocs system only supports pair-wise reconciliation. In larger networks, pairs will reconcile individually until the entire network reaches equilibrium.

We define an operation called “pull.” When Alice pulls from Bob, Bob sends Alice his linked list of line data structures. Alice adds to her document all line data structures that were newly created in Bob's version. Alice then compares all of her version vectors with Bob's. If one version vector is completely larger than the other (all of its components are greater than or equal to the other vector's components), then that vector's corresponding value will be used in the merged document. If one vector is not completely equal or greater than, the version vectors are concurrent. If the vectors are concurrent, DecentralizedDocs compares the two changed variables. If the changes are both the same (both users made the same change), it automatically accepts that change; otherwise, it asks Alice to resolve the conflict. At the end of the pull, Alice's version vectors are updated to contain the maximum version number between her vectors and Bob’s for each component.

The following listing shows an example of version vectors syncing:

Before sync:

A’s VV = <5,7>

B’s VV = <5,9>

B’s text is chosen

After sync:

A’s VV = <5,9>

B’s VV = <5,9>

A “sync” can be implemented as two pulls wrapped in a transaction: first Alice pulls from Bob (possibly with manual resolution), then Bob pulls from Alice (this is completely automatic; after the first pull, all of Alice's version vectors are strictly newer than Bob's). After the sync, Alice and Bob have identical documents and version vectors. If at any point during the sync the connection or one host fails, the sync is aborted and all changes are rolled back.

If Alice edits sentence *x* of a Line while Bob edits sentence *y* (two changes to the same *text* variable), DecentralizedDocs considers this to be a conflict requiring manual resolution. While it is possible to create a merged line containing Alice's sentence *x* and Bob's sentence *y*, doing so may create a paragraph that is semantically invalid. Merging in this case could be especially harmful in languages other than English. We think silently writing such a paragraph to the document is not user-friendly, and choose to alert our users instead.

# Scenarios

Our design for a Peer-to-Peer text editor supports various scenarios.

If two users each make changes to different paragraphs, the system will take the latest version of each paragraph without conflict because they involve different variables.

If two users both change the same line differently, we recognize the possibility that the two changes together produces incorrect semantics, and ask for conflict resolution. If both users make the same change, those changes will be processed without conflict. The resolved line will have its version vector incremented at the index for the user who made the pull, preventing double-reconciliation.

If a user moves a line, and another user edits that line, those changes will be reconciled without conflict because they involve different variables.

# Text Editor UI

DecentralizedDocs provides a text editor with a special user interface.

The document does not auto reconcile or display the cursor of other users. Instead, the document reconciles when the user clicks the save/refresh button. If there is a conflict, a special user interface will launch. A user may also attempt to “commit” the document in order to make it as final. All of the users need to agree to commit the document in order for the document to commit.

## Conflict Resolution UI

When the system encounters concurrent version vectors, it is not able to reconcile the changes by itself, so it will present a conflict resolution user interface dialog. Once conflicts are resolved between users, those who share a version of the files involved with the conflict will be updated when they sync with someone who has a resolved version (and thus newer version vectors).

# Committing

The commit system is implemented as a two-phase commit system. The commit initiator serves as a coordinator by contacting all of the users in the system individually. The coordinator also maintains a transaction log for error recovery.

When a pair of users attempts to commit, the system will check that both users are connected over a network. If the users are not connected, the message will be stored so that it appears once the users become connected.

The process will first check to make sure that all users are up to date (i.e. all version vectors match among all users). If this condition is not met, the commit will be aborted. Next, are asked if they want to commit through a user interface. All users must agree to the commit before the commit occurs. If any user disagrees, the process is aborted. After a user agrees to commit, their local file is locked.

If all users agree, the coordinator decides to commit. A checkpoint is recorded in the log, along with the specific version being committed. The coordinator then tells all nodes to actually commit. The local node marks that version as committed and unlocks the local copy.

If something goes wrong before the coordinator decides to commit, the coordinator can tell the local nodes to release their locks. If something goes wrong after the coordinator decides to commit, the commit will be processed once the nodes come back online.

For the sake of simplicity, only one commit at a time can be initiated.

Word Count: 1208 (Not including cover page)