DP2 Report: A collaborative text editor

Eleftherios Ioannidis, elefthei@mit.edu
Tal Tchwella, tchwella@mit.edu
Larry Rudolph, R01
May 25, 2012

1 Introduction

This paper describes the architecture of a collaborative text editor and the decisions that informed its design. At a basic level, each document is shared by a group, and the text editor allows multiple users, who are part of the same group, to work concurrently on the same document. In the proposed design, users can edit documents whether online or offline, and can join and leave groups at will. The system supports pairwise merging of documents, done through several possible connectivity scenarios. The merge process is automated and minimizes the need for users to explicitly resolve conflicts between document versions. Users can also name and commit a specific version of a document. Redundancy and use of a write-ahead log provide fault tolerance for the aforementioned features.

Keen attention to modularity played a large part in the design: the use of interfaces not only shields the user from the inner workings of the text editor, but separates the text editor's implementation from the rest of the system, reducing complexity. Some reasonable constraints and assumptions led to further sweeping simplifications in the design without significantly reducing usability. The end result is a collaborative text editor that is both simple to use and relatively simple to implement.

2 Design

The design description outlines (1) interfaces, (2) data structures, (3) logging, (4) merging, (5) commits, and (6) dynamic group membership.

2.1 Interfaces

The text editor has access to two interfaces - one for document storage, the other for communication with other users in the same editing group. Communication to both interfaces is transparent to the text editor.

2.1.1 Network Interface

The design provides a network interface, which will handle *read()*, *write()* and *get_online()* operations. All operations are handled using RPC calls and return appropriate error codes when they terminate. In case of direct connectivity, all functions work the same way assuming there is a working RPC socket established between two users.

The text editor will implement two possible network connectivity scenarios: group connections, and direct connections. In a group connection, any group member connected to the Internet is *online* and also connected to other online group members. On the other hand, a direct connection only requires a physical connection between two members of a document group, and does not require Internet connectivity.

The design assumes each user has a unique IP address. Thus, each user is identified by a UUID (Unique User ID), which is the user's IP address.

2.1.2 read(UUID, file, file_cursor, buffer)

Performs a remote *read()* on the designated *file* belonging to the user with the given UUID. The read starts at the *file_cursor* and returns read data in *buffer*. The number of read bytes is equal to the size of the buffer. Figure 3.1.2 shows all possible error codes.

2.1.3 write(UUID, file, file_cursor, buffer)

Performs a remote *write()* on the designated *file* belonging to the user with the given UUID. The write starts at the *file_cursor* and writes all data in *buffer* in the file. Again, figure 3.1.2 shows all possible error codes.

Return	Meaning
0	Returned successfully
1	User unreachable
2	Transaction incomplete
3	Network down
4	File not found

Figure 1: Possible return values of read() and write(), and their significance.

2.1.4 get_online(UUIDS[])

Sends a ping to everyone in the given list of group members (UUIDS[]) and returns the IP addresses of the ones available, via the network interface used. The network interface makes no assumptions about the state of UUIDS[]; instead, group membership is stored in each document, and passed whenever this operation is invoked.

2.1.5 Storage Interface

The design also provides several storage interfaces, allowing users to save their documents in different locations, such as local storage or on "cloud services." Implementation of the local storage interface is considered sufficient. In the local storage implementation, users store their own version of each shared document and its associated log on local disk. Each user's shared documents and logs are placed in a directory that serves as a chroot jail, to prevent both unauthorized accesses to the user's file system and unauthorized changes to the shared documents by outsiders.

2.2 Data Structures

The data structures that make up the document are organized hierarchically. The text editor contains one or more open documents. Each document contains an ordered list of paragraphs, and each paragraph contains an ordered list of words. These data structures are derived from a common abstract class. Figure 2 shows these different classes.

Document contents consist only of plaintext. Paragraphs are separated by new-line characters, and each word is the concatenation of contiguous non-whitespace and contiguous whitespace characters. Paragraphs always contain an empty word which marks the end of the paragraph. Words never contain new-line characters.

The text editor treats the plaintext and data structure representations of the document as equivalent. In particular, data structures are updated to accurately represent the document during a save or merge.

All data structures share a globally unique identifier (GUID), a checksum, and a modification flag. The document itself stores a hash table of version numbers (i.e. a version vector), while each paragraph and word stores an index indicating its position in the document.

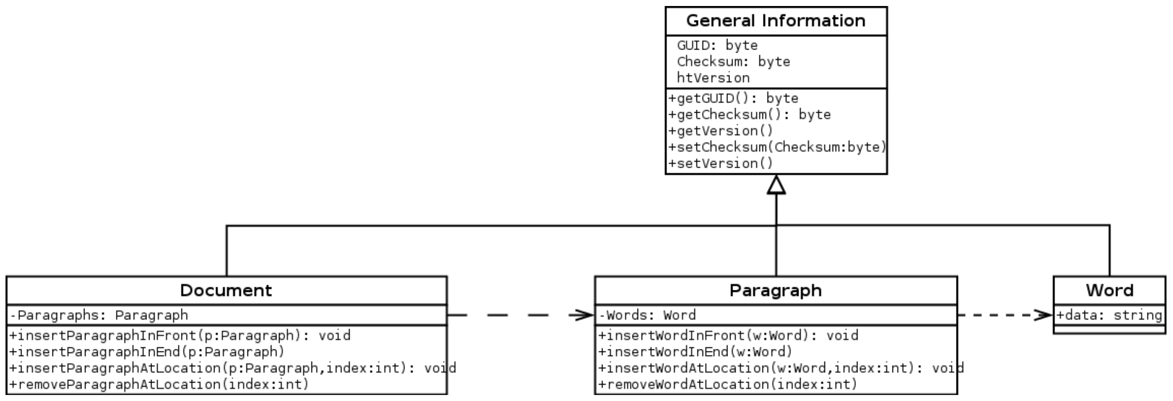

General Information
GUID: byte
Checksum: byte
htVersion
+getGUID(): byte
+getChecksum(): byte
+getVersion()
+setChecksum(Checksum:byte)
+setVersion()
Document
-Paragraphs: Paragraph
+insertParagraphInFront(p:Paragraph): void
+insertParagraphInEnd(p:Paragraph)
+insertParagraphAtLocation(p:Paragraph,index:int): void
+removeParagraphAtLocation(index:int)
Paragraph
-Words: Word
+insertWordInFront(w:Word): void
+insertWordInEnd(w:Word)
+insertWordAtLocation(w:Word,index:int): void
+removeWordAtLocation(index:int)
Word
+data: string

Figure 2: UML diagram of the document, paragraph and word classes and the abstract class they all inherit from.

2.2.1 GUID

Each GUID is a concatenation of the owner's UUID with the GUIDs of all parent classes as well as a unique label associated with the data structure's type. The document label is randomly generated and immutable once a document is created. The paragraph and word labels are generated by two separate integer counters local to each user and initialized to zero; the counters increment by one whenever a new paragraph or word is inserted. Note that a paragraph's GUID is *distinct* from its index, and likewise for words. Figure 3 shows how inheritance rules provide a GUID for each object.

2.2.2 Checksum

Every data structure corresponds to a contiguous subset of plaintext characters in the document. Checksums are calculated simply by taking an MD5 hash of the plaintext that defines a given data structure. A word's checksum is equivalent to the checksum of the string of characters that make up that word.

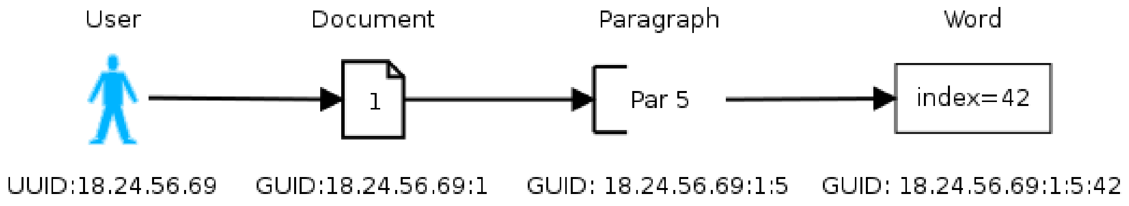


Figure 3: An object's GUID is the concatenation of the owner's UUID, parents' GUIDs and a unique object label.

2.2.3 Version Vector

The document contains a hash table whose keys correspond to the UUID of each user currently sharing the document. There are as many keys as group members.

2.2.4 Modification Flag

Every data structure contains a boolean flag whose value is **true** if the data structure was modified since the last merge. This flag is initialized to **false**.

2.3 Logging

Each user's text editor maintains a separate write-ahead log (WAL) for each shared document. The WAL is crucial for completing merges and commits as well as improving the system's robustness.

2.3.1 Operations

The WAL tracks what changes are made to the document. In particular, the following actions are recognized:

1. Inserting a new paragraph at a known index.
2. Deleting a paragraph.
3. Inserting a sequence of one or more words at a known index.
4. Deleting a sequence of one or more words from a paragraph at a known index.
5. Moving a paragraph from one index to another.

In turn, these operations define the log entry format. A log entry consists of the type of operation performed, the GUID of the paragraph being modified/added/deleted, the paragraph/word index where the operation is performed, number of words inserted/deleted if applicable, the resulting paragraph checksum and a timestamp. If multiple paragraphs are instantly added or deleted, the log will consider this as if one paragraph at a time was added or deleted.
Given the difficulty of distinguishing paragraph moves from deletes and inserts, the system defines a move as deletion of a paragraph and insertion of the *exact same contents* at a different paragraph index. When this occurs, the inserted paragraph will have the same GUID as the deleted paragraph.

2.3.2 Logs and Saving

For each open document, the text editor maintains a separate temporary log. Changes made by a user are automatically recorded in memory. These records are flushed to the temp log in local storage at some configurable, fixed time interval.
During a save, the user's text editor is blocked until the save completes or fails. Several steps are needed when the user saves a document:

1. Any unsaved changes in memory are flushed to the temp log. A checkpoint entry is appended to the log to indicate that all changes are recorded in local storage.
2. A shadow copy of the document is created, and the contents of the temp log are appended to the shadow copy's document log.
3. The text editor scans the log entries and sets the modification flag to true for each new or modified data structure and its parents.
4. Checksums are recalculated across the board, and appended to the shadow copy's document log. Another checkpoint entry is appended to the temp log; this is the save's commit point.
5. Finally the old document is deleted, and replaced by the shadow copy. A final checkpoint is written to the new document's log, and then the temp log is cleared.

The above scheme tolerates system failure at multiple points:

- If the system crashes before the user saves, some, if not all progress can be recovered by fast-forwarding through the temp log. This is also the case if the system crashes before step 1 of the save completes.
- If the system crashes during a save, but before the commit point, the save process can be restarted.
- If the system crashes after the commit point, recovery is trivial.

2.4 Merging

Any two users who are connected to each other can merge their local changes to produce a single unified document version. Merges must start manually: one user sends a merge request to another user, who must accept before the users merge. Only two users can merge at a time; both users must also save their document before the merge can begin. A user automatically rejects other merge requests if the user is already in the middle of a merge.

Once a merge begins, both users' text editors are blocked until the merge terminates, successfully or otherwise. Although slightly inconvenient, it greatly reduces the complexity of the merge implementation.

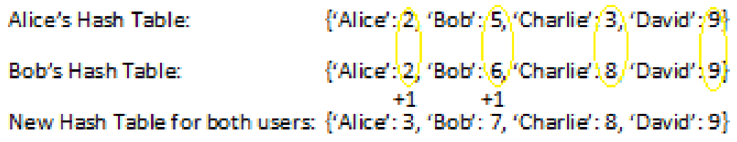
When a merge starts, each user's client creates an empty merge log. Once filled, the merge log will contain all changes that need to be applied to the document in order to successfully merge.

A merge outcome is determined by comparing the version vectors in the users' documents. Figure 4 illustrates one possible scenario. Define $version_A(U) = n$, such that user A has version n of user U's changes to the document, and U is some key in A's version table. Before comparing versions, if U's document's modification flag is set to **true**, the value of $version_U(U)$ is temporarily incremented by one during the merge. If a merge is successful, the new value of $version_U(U)$ is made permanent and the modification flag is set back to **false**.

Depending on what states the documents are in, a specific merge outcome is decided. Possible outcomes are described below:

2.4.1 Case 1

Suppose without loss of generality that $version_A(A) = version_B(A)$. If $version_A(B) = version_B(B)$, then the documents are equivalent and a merge need not take place. Otherwise, $version_A(B) < version_B(B)$ and therefore B's document version should overwrite A's version. Then, the contents of B's document are transferred in their entirety to A, whose client writes the document to a shadow copy. User A then compares the checksums and version vectors of the shadow copy and B's document. If both agree, A writes a checkpoint in the old document's log, sends an acknowledgement of success back to B, and replaces the old document with the shadow copy. A final merge checkpoint, i.e. the merge's commit point, along with the new version vector are written in both document logs and finally, both clients clear their merge logs and unblock the text editor.



```
Alice's Hash Table:              {'Alice': 2, 'Bob': 5, 'Charlie': 3, 'David': 9}

Bob's Hash Table:                {'Alice': 2, 'Bob': 6, 'Charlie': 8, 'David': 9}
                                          +1         +1
New Hash Table for both users:   {'Alice': 3, 'Bob': 7, 'Charlie': 8, 'David': 9}
```

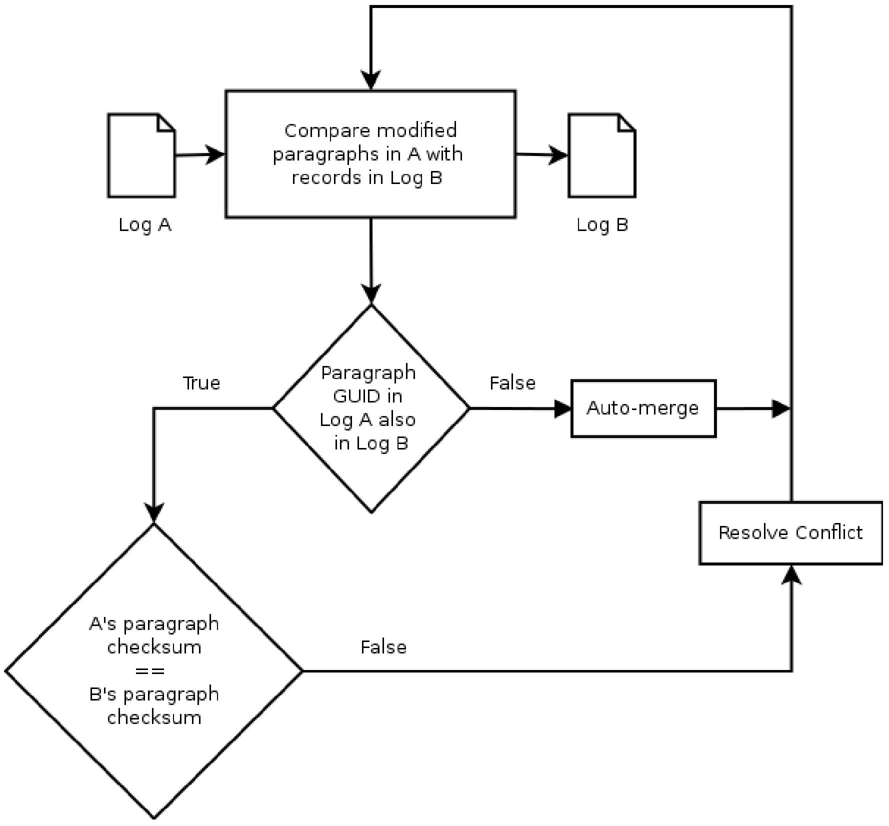
Figure 4: Instead of using IP addresses as keys, names are used in the example to better illustrate the scenario of updating version numbers. The version numbers are maxed, to produce a new hash table for both users.

If the checksums do not agree, the overwrite process retries up to three consecutive times. If the checksums still never match, the merge process aborts and both users are alerted of the error. The merge is aborted, both users are unblocked, and all merge logs and any shadow copies are deleted.

2.4.2 Case 2

Another possibility is that $version_A(A) > version_B(A)$, while $version_A(B) < version_B(B)$. In this case, the document checksums are compared next. If the checksums match, the changes in A's version and B's version are equivalent. The new version vector should be the max of each of A and B's version numbers.

Otherwise, A and B must backtrack through their logs to find the latest merge checkpoint in which $version_A(U) = version_B(U)$ for some key U in the version vector. From that point, the merge proceeds by rolling forward from each checkpoint to the end of the associated log. Figure 5 shows the pseudo code for this merging process.



Compare modified paragraphs in A with records in Log B
Log A
Log B
Paragraph GUID in Log A also in Log B
True
False
Auto-merge
Resolve Conflict
A's paragraph checksum == B's paragraph checksum
False

Figure 5: Flowchart of the merge process. Conditions are rhomboids and actions are rectangles. Arrows indicate code execution flow.

This particular merge process has the following steps:

1. Each user creates his own shadow copy of the checkpoint document version. Merged changes are applied to the shadow copies.
2. Both logs are scanned for move records. If the same paragraph was moved by both users, that paragraph is added to the conflict list. The conflict list contains sets of two or more changes that affect the same GUID.

   All other move records are merged automatically, and are ignored in future steps.
3. A's modifications are compared against B's modifications. If user A modified a paragraph whose GUID appears in any of B's log records, all matching log records are in conflict, and are added to the conflict list.
4. Step 2 is repeated for any records in B's log that were not added to the conflict list.
5. After all possible conflicts are found, each set in the conflict list is checked. If the checksums agree, there is no conflict. If none of A's inserts have the same index as any of B's inserts or are overlapped by B's deletes, none of A's deletes overlap any of B's inserts or deletes, and vice versa for B and A, these changes can be merged without conflict. Otherwise, a conflict is signaled and the merge starter must resolve this manually. After a manual conflict resolution, a checkpoint is written to each merge log.
6. All remaining changes not included in the conflict list are automatically merged.
7. Finally, the version numbers of both users are matched as shown in figure 4. The checksums of both shadow documents are compared and the merge process is finalized as in case 1. This time, both users send success messages to each other.

2.4.3 Unsuccessful Merges

Since the merge process uses shadow copies, both users can abort the merge at any time while keeping their files unchanged.
Merges can also fail for unexpected reasons storage failure, network failure, log failure, checksum errors, and incomplete transfers. In these cases, all merge logs and shadow copies must be deleted. If one of the users crashes during the merge, then the merge timeouts. The other user's merge log is erased, and his text editor is unblocked.

One problematic scenario occurs when one of the users crashes between receiving the checksum verification, and saving the actual shadow file as the primary file. To address this, the pre-merge document's log tracks the progress of the merge. If the user received a verification, the client should be able to see that the commit was successful or not, and then perform the necessary steps to recover. Either the shadow copy is saved as the primary copy, or the shadow copy is deleted.

2.5 Commits

The text editor allows a group's users to specify commit points, or canonical versions of the group's document.

Commits are uniquely named, immutable, and readable on local storage by all group members. For any commit, each user has the exact same copy. Commits are stored in a hash table whose keys are the names of each commit, and whose values are the document contents identified by the associated commit.

The proposed system implements commit points with a standard two-phase commit (2PC) protocol. To simplify the commit process and any reasoning about its correctness, all group members must be online during the entirety of any commit. The authors believe this requirement does not pose an unreasonable burden on usability.

2.5.1 Phase 1: Starting a Commit

Only when all the users in document's group are online, connected to each other, a member of the group can initiate a commit by assigning a name to his local version of the document. The initial committer broadcasts a commit prepare message to every other user; the message's contents include a PREPARE header, the identity of the initial committer, the commit name, and lists of the document's checksums and version vectors, ordered such that a given index refers to the correct pairing of checksum and version vector. The initial committer's document is locked until the entire commit process terminates.

Users (everyone else besides the commit initiator) vote to COMMIT or ABORT upon receiving a commit prepare message. After receiving the prepare message, a user can only vote if it isn't currently blocked (for example, in the middle of merging with another client); otherwise, the user waits for the blocking action to finish before voting.

Voting is automatic. A user votes COMMIT if and only if the commit version and local version have equivalent checksums and version vectors. Else, a user votes ABORT for any of the following reasons:

1. The commit version and local version are not equivalent.
2. A hardware or OS failure prevents the client from processing the PREPARE message.
3. The user timed out because a blocking action took too long to finish.

Upon deciding what to vote, a user records its decision in the log, and then sends an acknowledgement back to the initial committer, containing an ACK header, the user's identity and the value of the vote. If the vote is COMMIT, the user's text editor is blocked for the remainder of the commit process.

If the user does not respond within five minutes, the initiator will automatically assume the user voted an ABORT, due to a failure of timeout. Since all users are supposed to be online at the time of the commit, a failure to receive a response from one of the users means that the user crashed, and therefore the commit cannot take place at that time.

2.5.2 Phase 2: Finishing a Commit

When the original committer receives everyone's vote, the final action is decided as follows: only if everyone votes COMMIT, the final action is a COMMIT; else, the final action is ABORT. If the original committer times out, the final action defaults to ABORT. The final action is broadcasted to each client that voted COMMIT; the broadcast contains a FINAL ACTION header, the identity of the original committer, the commit name, and the actual final action to be taken.

When a committer receives this broadcast, it stores the successful commit in its commit table when the final action is COMMIT. A client does nothing when the final action is ABORT. Since a commit only occurs when the commit and local versions match, a client needs not undo any changes to the document; indeed, the document itself is never changed during a commit.

A successful commit implies that everyone has the same version. Therefore, when a commit succeeds, each user clears his document log.

After a client takes the appropriate action, it records its decision in the log and sends a second acknowledgement back to the original committer. The client's text editor is also unblocked. The original committer's document is blocked until acknowledgements are received from everyone else.

2.5.3 Simultaneous Commits

The proposed system forbids simultaneous commits. However, if one user has started a commit, it is very much possible for another user to start a commit before receiving the first user's PREPARE message.

Unsurprisingly, detection works when one commit initiator receives a commit broadcast from a different commit initiator. When this occurs, the client votes ABORT for the received broadcast and also automatically aborts its own COMMIT once all the votes are received. This occurs regardless if all the votes are COMMIT or not.

2.5.4 Dynamic Group Membership

As noted earlier, the version vectors are really version hash-tables, however, they will still be referred as version vectors. Of course, hash tables support insertions, deletions, and resizing. The initial size of the hash-table is going to be of 10 buckets, to allow 10 different users to work on the same document. Any additional user, above the current size of the hash-table is going to create a new hash-table with double the current size. If the hash-table shrinks to one-fourth the current size, it is going to shrink by a factor of 2.

In order to join a group, a new user must bootstrap himself to the group by accessing one of the group members through one of the network interfaces specified for the text editor. The client thus receives the IP addresses of all the other users, so he can communicate with them online, or by direct connection. Moreover, the new client receives the latest document version of the client he connected to. On all merges, the two clients first compare the checksums of the keys of their version hash-tables. If they match, they proceed to the merge process as described in the merge section. If they do not agree, the clients that are missing from the version vector are automatically added with along their version numbers to the other client.

Removing oneself from the group requires use of the commit process, although a commit isn't stored per se. Since commits can only succeed if all group members are online, everyone can acknowledge the removal of a user from the group without synchronization issues. Before the commit can take place, all version vectors are compared and merged. Afterwards, anyone who wants to remove himself can initiate a commit; removal can be safely done after completion of the 2nd phase.

3 Analysis

3.1 Scenario Specific Analysis

When two users merge, it is trivial to treat every difference between the two documents as a conflict. However, significant performance and usability gains can be realized in certain scenarios where document changes do not affect each other. The merge process described previously handles the scenarios described below.

3.1.1 Scenario 1: Non-Interfering Changes

Any change to a part of the document made only by user A or user B, but not both, is a non-interfering change. Since only one user made a change at that location, it cannot possibly introduce a merge conflict.

During the merge process, after the set of possible conflicting changes is produced, any remaining change must be a non-interfering change. These changes are merged automatically in step 5. However, this only applies at the paragraph level of granularity. The merge process is also able to detect non-interfering changes inside a single paragraph, since the log tracks the location of all document changes.

3.1.2 Scenario 2: Merging Between Three Users

Suppose users A, B, and C each have the same copy of document D. Users A and B connect to each other and make the same change to some paragraph P, producing $D^{'}_{A,B}$. Meanwhile, user C makes a different change to P, producing $D^{'}_{C}$.

Next, A and B disconnect from each other. A comes into contact with C so when the two users connect, a conflict is discovered. This conflict must be resolved manually. Now, A and C's document versions match.

Finally, B and C connect. While one can naively consider B and C's versions of paragraph P to be in conflict, C's versions are greater than or equal to B's versions for all users U in the version vector. This means merge case 1 can be applied, and B's document is rolled forward to C's version automatically. Figure 4 demonstrates this example of version vector matching.

Automatic merging is only possible if B made no further changes to paragraph P after disconnecting from A. Otherwise, when B and C connect, this also constitutes a conflict and must be resolved manually.

A	221 $\rightarrow$ 323
B	221 $\rightarrow$ 221
C	112 $\rightarrow$ 323

Figure 6: Initially, users A and B are merged so they have the same version number 221. Once user A and C merge, their new version number is 323. When users B and C come to merge, all the version numbers in C's version vector are bigger than or equal to B's version numbers, and therefore C's version overwrites B's version.

3.1.3 Scenario 3: Simultaneous Move and Modification

If users A and B move the same paragraph to two different locations, there must be a conflict. However, if user A moves paragraph P while user B modifies the same paragraph, both changes can be merged automatically without any conflict.

These changes are handled in step 1 of the merge process, where the log is scanned for paragraph moves. The design makes this possible because when a paragraph is moved, the GUID is unchanged.

3.1.4 Scenario 4: Equivalent Changes

When two users make the same changes to a paragraph, the resulting checksums are also equivalent for both users. Step 4 of the merge process detects this scenario and treats these changes as if no conflict had ever occurred. This optimization can also apply at the document level.

3.2 Broad Performance Analysis

On this collaborative text editor document reads, writes and merges take polynomial time, as the pairwise merges scan both users logs, and compare each record, one-by-one. Moreover, the performance depends a great deal on the external underlying infrastructure such as network latency and storage read/write speed of the storage interface.

On the other hand, commit operations are a bottleneck in terms of performance, for two reasons: they require all users to be online, and also stall other operations until the commit terminates, since everyone's vote is required. The commit overhead increases with the number of users as it takes more time for everyone to get the prepare message from the initiator and vote on a commit. In the end, the running time of a commit operation depends on when the last user will vote on the commit. A more efficient design would require introducing additional complexity and bridging commits for users connected through direct connectivity. For simplicity's sake the authors decided to implement unbridged commits, with the only disadvantage that they are slow and provide an upper bound on the number of collaborating users.

4 Conclusion

The crux of the design is the modularization of the text editor, network, and storage systems, as well as the interfaces connecting them. Simplicity and transparency were the guiding principles; the overall design achieves both, for users and programmers alike. The design is also robust to system crashes and network disconnections.

While the design can be implemented as is, one should consider these additions and improvements: a basic security system, optimizations in the merging algorithm, a solution to eliminate blocking during merges, and support for asynchronous commits and group membership changes.

5 Word Count

4230.