Design for a Collaborative, Distributed Editor

Mason Glidden, mglidden@mit.edu
Ted Hilk, thilk@mit.edu
Vlad Kontsevoi, vok@mit.edu
TA: Peter Szolovits
2:00 pm section

Introduction

Current collaborative text editors lack support for offline editing modes and require a central server. They rely on a constant Internet connection to synchronize with the server, which maintains a master copy of the document. We propose a new text editor, Synchronicity, which solves these issues by allowing distributed clouds of peers to collaborate on a document. No central server is needed, and users can make edits without an Internet connection and merge with their peers later on.

Synchronicity stores a given text document as an ordered B-tree. The root node of the tree represents the document, with paragraphs as children and sentences as grandchildren. The order of child nodes signifies the order of the elements in the document. Synchronicity allows for the addition and deletion of nodes, as well as their movement or modification. It defines atomic operations over the tree to perform each of these actions. It writes the operations to logs associated with the document as they occur, which simplifies the merge process.

When merging two versions, Synchronicity users share a modified version vector over TCP to determine the most recent common ancestor and perform a three-way merge using the logged atomic operations. The common ancestor can be determined by taking the pair-wise minimum of the two respective version vectors. Synchronicity then replays all changes since this ancestor, allowing the users to resolve any conflicts that cannot be handled automatically.

Major design decisions with trade-offs included defining sentences as the most fundamental entities in the document representation, storing snapshots of the document at the time of a given merge operation, and storing logs of the operations conducted on a given document. Defining sentences as the lowest-level units of the document reduces the size of the document tree and eliminates the possibility of a specific type of merge error described below, but gives up some granularity in the logs. Storing document snapshots on merge reduces the amount of time necessary to start Synchronicity or to perform a merge operation, but marginally increases the amount of space required to store a Synchronicity document on disk. Finally, keeping logs in the first place allows us to simplify the automated merge operations considerably, at the cost of a modest increase in space requirements. Given the high priority that the project description places on functionality, we consider these trade-offs justified.

Design

Assumptions

We assume users can only edit plain text files with Synchronicity. Per the project description, we assume each user knows the IP addresses of the other users. We also assume that no machines permanently crash without notification from either that user or from another user, since it is otherwise impossible to prevent subsequent commit attempts by any user from permanently hanging. We assume there are no malicious clients, and hence the design does not include any specific security considerations -- note also that none were required in the project description. Finally, we assume that updates conducted at a later point in time than a commit initiation, but that occurred before the updating user received the request, should still cause the commit to fail -- we believe this is in the spirit of the commit operation as described.

We consider each of these assumptions to have been warranted directly by the project description.

Document Granularity

We define sentences as the most fundamental units of documents. Users can perform atomic operations over sentences or paragraphs, but not the contents of sentences. Therefore, if a user changes a single word, the entire sentence is marked as modified. Consequently, Synchronicity cannot automatically merge two changes within a single sentence that result in a different final state; a user must must handle them manually.

We consider this superior to the alternative of tracking operations at the word or character level. Consider a situation in which Alice and Bob each remove a repeated word in a given sentence (e.g. “the the”): Alice removes the first instance, but Bob removes the second. They then merge their respective versions. If Synchronicity handled changes at the word level, the merged document would contain no instances of the word at all. Conversely, if changes were merged only at the sentence level as we propose, the merge process would observe that both Alice’s and Bob’s versions of the sentence are identical and do nothing, which is clearly the correct response. This also has the benefit of reducing the size of the document tree and the lengths of the logs. It merely sacrifices some logging granularity.

Document Representation

Synchronicity uses an ordered B-tree to store the current document state. As shown in Figure 1, the ordered children of the root node represent paragraphs. The first paragraph is the first child node, the second paragraph is the second child, and so on. We define paragraphs as text sections separated by line breaks. The document representation implicitly defines line breaks by the fact that multiple paragraph nodes can exist. The ordered child nodes of the paragraphs represent sentences. Each sentence node contains the text of that sentence. We define sentences as text separated by the types of punctuation marks that can end sentences: i.e. periods, exclamation marks, and question marks, possibly with trailing quotation marks. However, we consider any white space following a period to be part of the next sentence, and it is stored explicitly in that sentence node. Refer to Figure 1 for a graphical depiction of the document representation in Synchronicity.

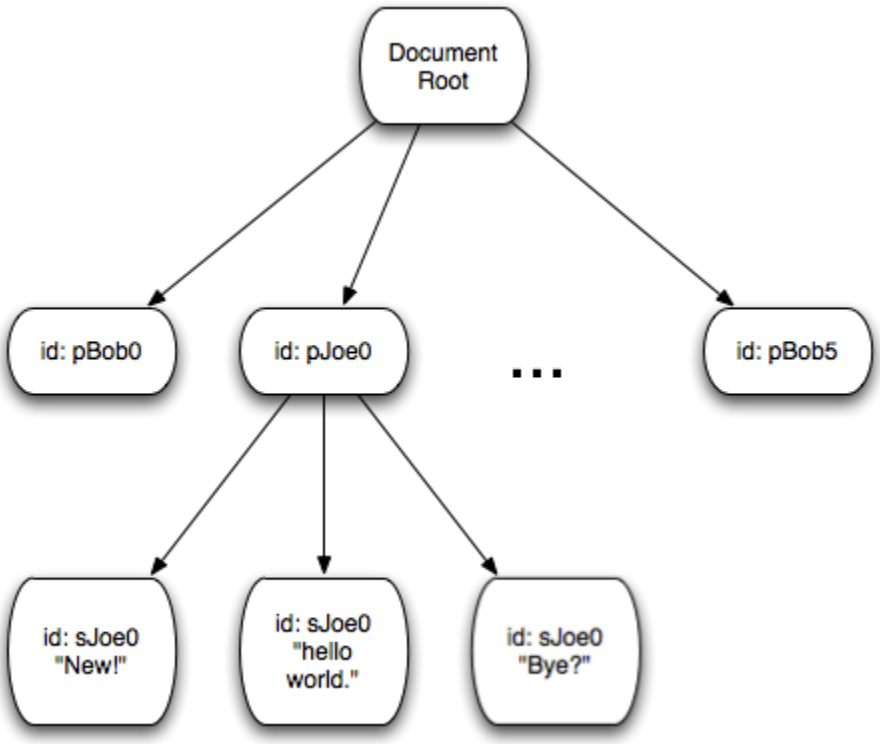

Figure 1: Example of the document tree structure.
Documents in Synchronicity are based on a tree structure. At the top is the root node. Its children are paragraphs, which are defined by line breaks. Their children, in turn, are sentences, defined by periods, question marks, or exclamation marks and any following quotation marks. For clarity, the usernames in this diagram omit the appended MAC addresses; we discuss these in more detail in the "Dynamic Groups" section.

Each node in the tree has a unique identifier. The identifier consists of the user identifier of the creator, the node type (i.e. 'p' or 's'), and an integer chosen at creation time. Since every user has a unique identifier, the identifiers are globally unique if the integer is a local, monotonically increasing number, so this is how Synchronicity chooses them.

Synchronicity stores the document tree in memory and flushes it to a file on disk a few seconds after every atomic operation, if no subsequent operation has yet taken place. It does not immediately flush a given change to disk, since that could result in an excessive number of disk accesses (e.g. every time the user types a character). When saving to disk, the node for a paragraph contains a list of sentence identifiers corresponding to child nodes. The order of these nodes represents the order of the sentences in the document. Similarly, a list of paragraph identifiers at the start of the file signifies the order of the paragraphs. When restarted, Synchronicity can trivially reconstruct the textual representation by parsing the tree.

For actions, Synchronicity uses a limited set of atomic operations on the document and stores records of these actions in a log file. All logged actions contain, at minimum, the identifier of the node acted upon. Synchronicity creates *add* and *delete* log entries when nodes are added or

removed respectively. An *add* action also stores its parent and the position relative to its siblings. *Move* operations are recorded when the user moves a paragraph or sentence from one location to another and contains the start and end locations. If a user changes part of a sentence, Synchronicity creates a *modify* record, which contains the final content of the sentence. (The initial content can be located by finding the most recent previous *modify* record).

Synchronicity compacts its logs before initiating or responding to a merge request. It does this by first enumerating all actions in the log, assigning monotonically-increasing integer labels to each successive action. It then uses a hashmap to associate each paragraph/sentence node ID with a list of the successive actions performed on that respective node. Successive *delete-add* chains involving the same content with no intervening *modify* entries are then combined into a single *move* action, and successive *move* actions with no intervening *modify* entries are combined into a single line. Consecutive *modify* actions are combined into a single *modify* action containing the last change that occurred. Lastly, Synchronicity enumerates the still-remaining records in ascending order by label to produce the final, compacted log.

Synchronicity tracks paragraph merges and splits as a group of *add*, *delete*, and *modify* operations. If a user adds an end-of-line character to the middle of a paragraph, the sentences before it remain the same and are not modified. A new paragraph is *added* to store content after the break, and the respective sentences are *moved* to this new paragraph. If an enter was removed, the second paragraph would be *deleted* and the sentences *moved* back. Similar actions are taken for sentence merges and deletions, using *modify* instead of *move*.

When a node is deleted, Synchronicity keeps track of it to ensure that a copy-paste combination keeps the same node structure. We maintain a list of all past elements that the user deleted. When the user pastes text into the document, Synchronicity checks to see if it matches any of the removed elements. If so, the text retains the same identifier as the removed node, creating an *add* event log. The log-compaction step will turn this *delete-add* combination into a single *move* operation. Synchronicity saves this list of past elements in a log as well, but clears it whenever a checkpoint occurs since the information is only relevant between merge operations. This log prevents an issue whereby a user could cut some text, close Synchronicity, re-open Synchronicity, and then paste the text, resulting in a new sentence/paragraph ID and thus potentially requiring manual merges later if some other user changed the original paragraph.

A checkpoint includes the current version associative array (discussed below) and a full enumeration of the contents of the tree. Checkpoints occur whenever two users perform a merge operation. Users can disable enumeration on individual client machines if merge operations occur frequently enough that such consolidation would create excessively large files. The trade-off is a longer load time for the document, since Synchronicity must then rebuild it from scratch.

State Tracking for Merge Operations

We use a version associative array (VAA) to track versions through multiple merges. A VAA is essentially a version vector that tracks users by username rather than by an index. It can be implemented as a hashmap, a binary tree, or any other associative array data structure. This

permits users to dynamically join or leave the system, as discussed below. Overall, the VAA allows users to make and merge changes in a fully decentralized manner.

Each key in the VAA represents a Synchronicity user; the values represent the number of edits that each user has made to the *local* version of the document. A VAA whose keys are all zeros indicates a blank document with no changes made. If user A edits the document, he increments his own entry in the VAA. He does not increment any other entries. After two users merge, they set each element in their respective VAAs to the maximum of that element across both VAAs, resulting in identical copies.

Dynamic Groups

Group membership is dynamic: members can join or leave at any time. New members join by choosing a name and syncing with any current member. To eliminate username conflicts, each user's MAC address is appended to their chosen name to form their username. This makes every username unique, preventing multiple users from asynchronously choosing the same name. Figure 2 illustrates this process. The username generation process only takes place once per user, after which that user's username is stored on disk. Therefore, even if a user has multiple MAC addresses (e.g. one for a wired Ethernet NIC and a different one for an 802.11 wireless NIC), his or her username will remain consistent regardless of how he or she subsequently accesses the Internet. Members can leave by telling another member of the group, who marks that user as being permanently deactivated. The deactivated user is no longer required to vote on commit requests. Once a member leaves, they must choose a new name to rejoin the group.

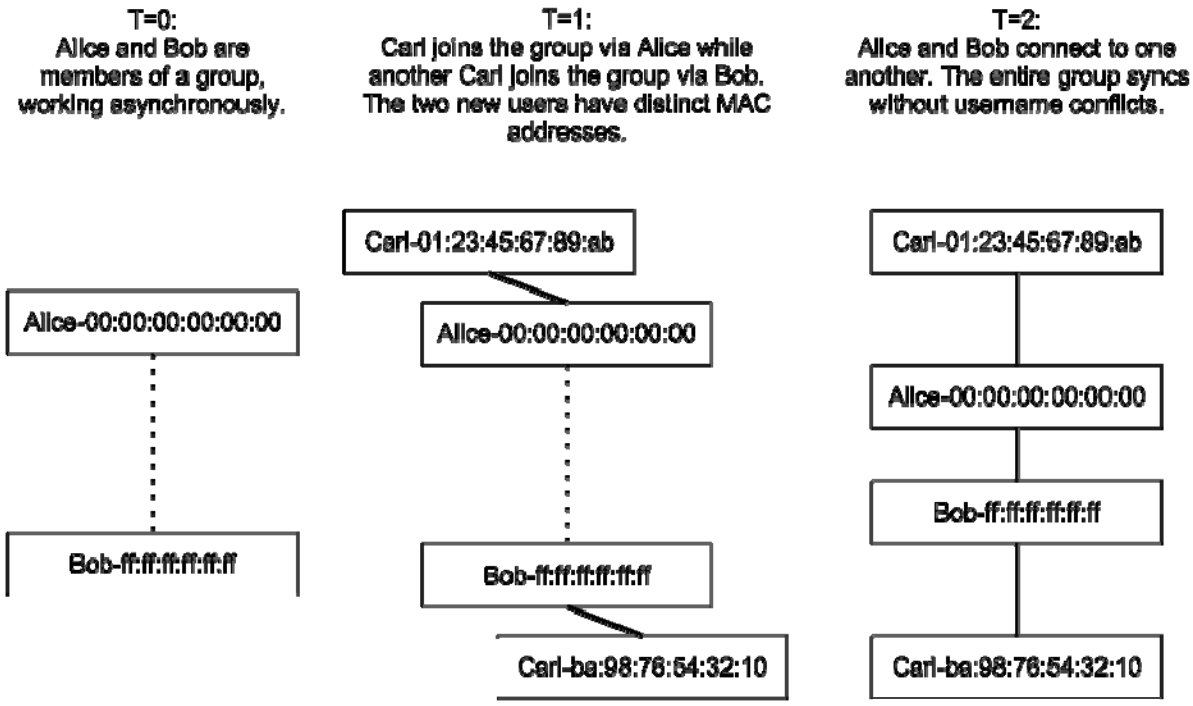

Figure 2: Two users join the network with the same username.
As described, using the MAC address is usernames prevents asynchronous name conflicts.

Synchronicity updates group membership whenever two users merge. Each of the two users adds any users that they haven't seen before to their respective VAAs, and removes users that the other client has marked. This propagates all user changes. Users must sync their membership table before leaving the group to propagate any new additions or deletions.

Protocol and Algorithm for Merge Operations

Suppose user A wishes to merge a document with user B. A sends his VAA with a merge request to B. B replies by sending her VAA to A. A and B first iterate over the VAA keys of the other, updating their VAAs to include information about joining or leaving users. A then compares his VAA to B's, computing the VAA of their most recent common ancestor by taking the pairwise minimum of the values associated with each key in the VAAs.

The merge itself is conducted by replaying the actions of the two users from their respective logs. Conflicts that do not eventually end with the same state require manual resolution. Upon establishing the common ancestor, A and B can determine the changes that their partner is missing. A gathers all log statements that have occurred since the common ancestor and sends them to B. B re-constructs the version of the document present at the common ancestor and collects all log statements that occurred after. B then associates log statements with the node that they modify. If a node only has one change or changes that do not conflict, they will be automatically propagated to the final version. Conflicting changes will be handled by the user. Conflicting changes cause the VAA for both users to be incremented. This prevents repeated manual conflict resolution upon subsequent merges with other users.

For example, suppose users A, B, and C each start with an original document <0,0,0>. They each make edits, after which A has vector <1,0,0>, B has <0,1,0>, and C has <0,0,1>. A then merges with C (common ancestor is <0,0,0>), after which both A and C have <1,0,1>. B then merges with C (common ancestor is <0,0,0>), so B and C both have <1,1,1>. Finally, A merges with B (common ancestor is <1,0,1>, i.e. A's current version), after which A, B, and C all have <1,1,1>. This chain of merges is demonstrated in Figure 3.

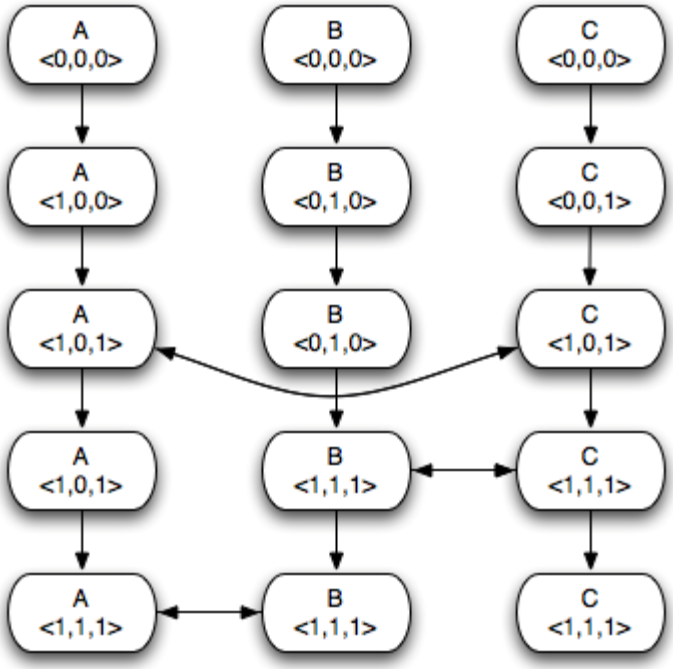

Figure 3: A sample merge sequence.
Single-sided arrows represent the progression of time. Double-sided arrows represent merges.

Synchronicity automatically infers when manual conflict resolution for a given change will not be needed, and skips it. This occurs when the structures being edited end up at the same state, even though there was a previous conflict. For instance, suppose A changes a sentence stating “This is a cat” to “This is a hat”, and B changes “This is a cat” to “This is not a cat”. They then both change the sentence to “This is a furry hat.” Since this latter state is the same in both versions of the document, Synchronicity does not prompt B to resolve the conflict between the initial edits. If both users had deleted the conflicting sentence, Synchronicity would likewise merge the changes automatically.

After combining the log files on her side, B can trivially determine which log statements transformed A’s version of the document to the merged version, so she includes the missing transformations in the merged document that will be sent to A. This effectively merges the ancestry tree of both clients, allowing the clients to use the others ancestors as common ancestors in future merges, which is critical to the operation of merge functionality.

Once B generates the merged version of the document, she generates the new VAA by taking the pairwise maximum of the values associated with each key in her VAA and A’s VAA. After storing the document and these new values locally, B sends the new document back to A, including the new VAA. A then sends an acknowledgement to B stating that the new version has been successfully stored.

If either user crashes during the merge operation, before writing the updated VAA, then Synchronicity simply deletes whatever happened after the last successfully-written VAA update. This effectively rolls back the partial merge.

If A crashes before it receives the updated document, then B will have the merged document, while A will still have its original local state. Importantly, though, state will be consistent on both sides. If A and B were to attempt another merge immediately after A reboots, without having made edits, the new common ancestor will be A, as B's VAA is strictly newer than that of A. The merge operation thus reduces to replacing A's current document with B's.

If A crashes after receiving the merged document but before sending the acknowledgment to B, then B can again retry the merge if desired. They will have identical VAAs, and nothing will need to be merged.

N-Way Merges

Supporting two-way merges permits conducting n-way merges by using 2n-3 two-way merges [1]. Suppose there are four users: A, B, C, and D, each with different versions of the document. To perform a 4-way merge, A can merge with B, who then merges with C, who then merges with D. C and D then have the final version of the document. D then merges with A and then with B, at which point all users have the final version. This required a total of five merge operations.

Commit Points

Synchronicity allows users to create named commit points, based on a specific version of the document. Commits are created through a two-phase commit process. Once a user initiates a commit with a particular commit name, Synchronicity first attempts to connect to all known members, broadcasting the document version to be committed, the corresponding VAA, and the commit name with the originator's username appended (for example, "['Draft 2a', 'Bob-ff:ff:ff:ff:ff:ff']"). If anyone has un-merged changes (seen by comparing the respective VAAs) or disagrees with the version name, this information is propagated back to the originator, aborting the commit. The commit returns after it can contact all current members. Figure 4 demonstrates a successful asynchronous commit.

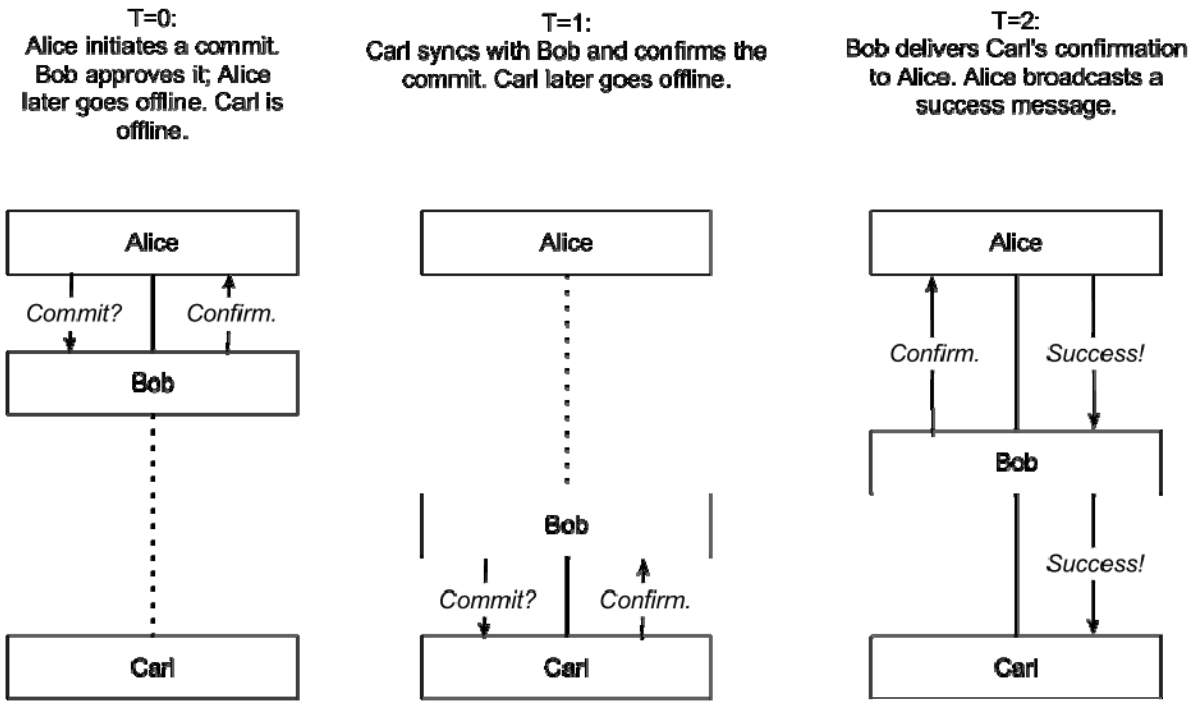

Figure 4: The two-phase commit process.
Commits can work asynchronously, just as merges do. The success message causes the commit to be stored in the users' logs. For clarity, the usernames in this diagram omit the appended MAC addresses; we discuss these in more detail in the "Dynamic Groups" section.

Once Synchronicity receives confirmation from every group member, it logs the document state, the version associative array, and the commit name. Then it broadcasts the commits success and naming to all known members of Synchronicity. Upon receiving this success, each user logs the document state, the VAA, and the commit name in their logs.

Upon a successful commit, each user will have a copy of the committed version, the associated VAA, and the associated commit name in their logs. When a new user joins the network, they will get the full log state of a user already in the network, so they will have the commit listed in their new logs. Thus, any user can retrieve a committed state on demand from their logs.

Synchronicity will remain in a consistent state if multiple users successfully commit with the same commit name. Since the originator's username is appended to the commit names, commits will not have duplicate identifiers. Users will simply have multiple entries in their commit logs, with the same commit name but different originator usernames.

If a commit is initiated by user A but user B permanently crashes before confirming it, the commit will essentially have failed: A will never get B's approval. This is unavoidable, as it is impossible to distinguish an extended period of offline editing from a permanently downed user.

If the commit is successful, but the originator, user A, crashes before logging receipt of all necessary confirmation messages, then the commit will have failed. User A cannot tell whether it received all confirmation messages. If user A crashes after logging receipt of all necessary confirmation messages but before sending the success message, then user A will send a success message upon rebooting. This could lead to sending the success message multiple times – however, this assures at-least-once delivery.

If the commit is successful but user B crashes before receiving the success message, then that user will be in an inconsistent state in regards to the success of that commit. To remedy this problem, Synchronicity merges users' commit logs whenever users merge. In this manner, successful commits are eventually propagated across the network.

Analysis

Design Challenge Use Cases

Synchronicity allows for disconnected operation. Two or more users can make changes to different objects in the document, then merge later when they reconnect (through the Internet or any other direct link). If they change the same object in a way that leaves its state inconsistent, Synchronicity asks the users to resolve the conflict. Once it has been resolved, Synchronicity increments the relevant VAA elements, preventing other users from having to resolve the same conflict repeatedly since the result of the first manual merge overwrites the older versions.

Synchronicity also permits editor use by two users who are directly connected to one another, but not to the Internet. Synchronicity supports this trivially, since the application is strictly peer-to-peer in the first place. The two users simply need to create an ad-hoc network, for instance by plugging a crossover Ethernet cable into their respective wired NICs, or back creating a wireless ad hoc network. Their respective instances of Synchronicity can then communicate with each other over this link, allowing them to synchronize with each other. Once they reconnect to the Internet, they can send their changes to other users.

Synchronicity allows a given user to initiate a commit request on a given document, specifying a particular name for the commit point. The request then goes out to the other users and they confirm that their own versions match it, at which point they write commit points to their respective logs and return a result of success. Once the commit points have been written to the logs, their respective users can agree on the document corresponding to the name of the commit point, which is in persistent storage so it is not lost in the event of a crash. If the versions of the document are inconsistent, then the commit fails. This ensures that a committed document correctly includes the changes from each user at the time of the commit operation.

Group membership in Synchronicity is dynamic. By using a version associative array (VAA), users can join or leave at any time. New users sync with an existing user, and departing users must notify someone else in the group.

Conflict Resolution Use Cases

Two users, Alice and Bob, add lots of text to the document in different paragraphs, and also make different changes to a single sentence in the introduction. Once Alice and Bob connect to each other and Alice sends a merge request to Bob, Synchronicity parses the changes in terms of the atomic operations conducted by Alice and Bob respectively, triggering a manual merge for the conflicting *modify* actions involving the first sentence. Since there are no conflicting

operations for the *add* actions involving the two paragraphs, no manual merge is triggered for these actions.

Two users, Alice and Bob, are connected to each other, and Bob makes a change to a sentence. Concurrently, an offline user, Charlie, changes that same sentence in a different way. Alice goes offline but later meets Charlie, at which point they synchronize, detect the conflict, and Alice resolves the conflict. At a later point, Charlie meets Bob and synchronizes with him. Bob does not have to resolve a conflict between his change and Charlie's change, because when Alice resolved this conflict it triggered an increase in the VAA elements for Alice and Charlie. The result of the manual change just overrides Bob’s outdated version silently.

One user, Alice, moves several paragraphs from one section of the paper to another, but does not change the contents of those paragraphs. Concurrently, another user, Bob, who is offline, edits a sentence in one of those paragraphs. When Alice and Bob meet, no conflict resolution is required, since the *modify* action Bob created has a reference to the ID of the specific paragraph in question, which is not changed by Alice’s *move* action.

Two users, while not connected to each other, find a spelling mistake and correct the same word in the same way. When they re-connect, no conflict resolution is required, since Synchronicity checks whether later states are identical before requiring a manual merge for a given object.

Performance Analysis

The overhead from Synchronicity is fairly small -- only a small constant factor under typical workloads. Indeed, every sentence in any given document is typically added, deleted, modified, or moved a small constant number of times. This means that the space usage of the log representation of a document is only a constant factor more than the space usage of a text representation of the same document. Commit points and VAA states use additional space, but this space is relatively small under typical workloads.

Suppose that a typical document is comprised of approximately 6,000 words, or about 300 sentences. Suppose that three users independently write separate sections of this document, synchronize their changes and create a commit point named “First Draft”, make edits, and create a commit point named “Final Version”.

Estimating that each word takes up about 7 bytes, a text representation of this document would take up roughly 42,000 bytes. Each of the users’ contributions would take up roughly 14,000 bytes. Once two users synchronize with one another, their logs will have about 28,000 bytes each. Once the third user synchronizes, every user’s logs will have about 42,000 bytes. If we suppose the each VAA takes up 1,000 bytes (an overestimate), then the VAA overhead will be about 3,000 bytes. The overhead from the “First Draft” commit point will also be about 1,000 bytes, as all users will already have the corresponding document version logged.

Suppose that each user edits 50 sentences, or about 1,000 words each -- approximately 7,000 bytes. Once these edits are synchronized and conflicts are resolved, each user’s logs will have 21,000 additional bytes, as well as 3,000 bytes from storing the corresponding VAAs. The “Final Version” commit point will take up an additional 1,000 bytes, as all users will already have the corresponding document version logged.

Synchronicity uses about 71,000 bytes total to store the document's raw text, VAAs, and commit points in the log. There is an additional overhead associated with logging atomic operations and with storing tree node identifiers. If we assume that it is 100 bytes per sentence, this is an additional 45,000 bytes, as we assumed that the users performed 450 *add*, *modify*, *delete*, and *move* operations on the sentences.

Thus, Synchronicity's representation of the final document uses about 116,000 bytes, which is comparable to the 42,000 bytes required to store a text version of the document. The bandwidth overhead is similarly modest.

Conclusion

Synchronicity is a distributed collaborative text editor that uses a tree-based document representation, logging, and version associative arrays to support both on-line and off-line collaborative editing. It performs well under common use cases and supports dynamic group membership, automatic and manual merges, as well as asynchronous creation of commit points.

Synchronicity's strengths include its support for dynamic group membership as well as its powerful merge operation. Furthermore, while some logging overhead is necessary to enable a simple merge operation, the overhead is relatively minimal.

Future work

Synchronicity could be improved by focusing on security. As is, Synchronicity contains potential security flaws. A malicious user could pretend to be a co-worker and make unwanted changes. Additionally, an attacker could pretend to have the higher version number, forcing a merge to his state. However, these attacks are mitigated somewhat by the extensive logs, which save the entire history and rollback state.

Currently, if two users can only communicate indirectly (e.g. through a third user), then they will not be able to commit documents. Synchronicity could be updated to propagate commit requests to all known neighbors, but this is left as future work.

References

Acknowledgments

Thank you to Katherine Fang, who provided feedback on our design proposal and clarifications on the project requirements. Thanks also to Prof. Peter Szolovits, who provided a great deal of the relevant technical background in 6.033 recitation.

Citations

[1] T. Jim et. al., "How to Build a File Synchronizer," Unpublished, 2002.

[2] M. Rosenblum and J. K. Ousterhout, “The Design and Implementation of a Log-Structured File System,” Berkeley, 1991.

[3] J. H. Saltzer and M. F. Kaashoek, Principles of Computer System Design. Burlington, MA: Morgan Kaufmann, 2009.

(Word count: 4964)