Collabrador: a collaborative peer-to-peer text editor

Jacob Hurwitz
jhurwitz@mit.edu

Colleen Josephson
cjoseph@mit.edu

David Lawrence
dlaw@mit.edu

May 10, 2012
R07 and R09 with Nir Shavit

1 Introduction

Collabrador is a peer-to-peer text editor that lets multiple users collaborate on a document. Users can edit a document while online or offline, although they cannot view each other's changes until they join a (possibly ad-hoc) network and synchronize. In addition to storing the text of a document, Collabrador logs each user's edits as individual insert, delete, and move operations. When users make distinct changes to a document, the branches can be merged by replaying the changes made by one user atop the changes made by another user. This technique is known as the operational transform and is the basis for all major collaborative editors.

2 Design

2.1 System architecture

On each computer, Collabrador consists of a *text editor* and a *checkpoint database*. In addition to storing the text, Collabrador's text editor saves a description of the *operations* performed by the user. These operations are inserting a character at a given position (*insert*), deleting a character at a given position (*delete*), and using cut/paste to move a block of text from one position to another (*move*); collectively, these operations form an *operational transformation* describing how the user modified the initial document. When the user clicks the "save" button, Collabrador creates a *checkpoint* containing this operational transformation and saves it into the local *checkpoint database* [2].

To be more specific, Collabrador stores a checkpoint database consisting of *checkpoint objects*. A checkpoint object is a data structure encapsulating the details of the operational transformation performed by that edit, the hashes of the edit's parent or parents, the hashes of the edit's children, the hash of the unique checkpoint to be used as the base of the operational transformation, and the *user visibility list*, which is just the list of users that have locally stored this checkpoint. Checkpoint objects are keyed by the SHA-1 hash of the object itself. For full implementation details, see Section 2.1.1. We will assume that SHA-1 has no collisions, so two checkpoints will have the same hash if and only if they perform the same changes to the same base from the same parents.

When the system consists of a single computer, the checkpoint database is just a linear progression of checkpoints. Adding in multiple computers, a hypothetical central server keeping track of edits on all computers would model the checkpoint database as a directed tree, with a branching node occurring whenever two computers make distinct edits. If two computers are allowed to synchronize with each other, then branches of this tree can join together (creating nodes with two parents instead of just one) and so the checkpoint database becomes a directed acyclic graph (DAG) instead, where the nodes are checkpoint objects, and the edges are OTs. (For this reason, this paper uses the terms *checkpoint database* and *DAG* interchangeably.) The challenge with Collabrador is to maintain this DAG in a distributed fashion because there is no central server. Collabrador's solution is a synchronization process in the background that lets users constantly communicate with each other to share and exchange parts of the graph. Collabrador operates on top of TCP/IP to ensure reliable and in-order packet delivery.

The checkpoint database can be in two states. At most times, a user's checkpoint database will be a graph with only a single leaf node, representing the most recent checkpoint created by the user. A database with just one leaf node is in the *merged* state. When synchronizing with another user, that user's checkpoint objects will be copied into the local checkpoint database, causing the graph to have two leaf nodes. A database with two leaf nodes is in the *unmerged* state. Then, as described in Section 2.3, the Collabrador merge algorithm will find the least common ancestor of these leaves, merge the two edits into a single checkpoint, and then save this checkpoint into the database as a node with two parents and an operational transformation relative to the least common ancestor.

2.1.1 Storage of the DAG

The system used by clients to store the DAG of document versions must meet performance and reliability requirements.

The performance requirements are easily addressed by having each node store pointers to its children, parents, and the lowest common ancestor of its parents. Since nodes are named by hash, we also maintain an index table containing pointers to nodes by hash. Finally, we keep pointers to all current leaf nodes (usually one, but sometimes two), and the original common ancestor for the document. Using this structure, we may perform all relevant graph traversals in constant time per node.

To meet our reliability (particularly atomicity) requirements, we store the DAG in a database that is designed specifically to provide fault-tolerance guarantees. One such candidate is PostgreSQL [3].

2.1.2 Named Checkpoints

In order to support named checkpoints, known as *commits*, Collabrador needs one additional data structure. Each user should store a hash table mapping user-defined names to checkpoint hashes. With this data structure, the algorithm for returning a specific named version of a document is as easy as looking up the hash associated with the name, and then looking up the checkpoint object associated with that hash.

One key difference between a checkpoint and a commit is that eventually, all users in the user visibility list have the committed version and associated commit name stored locally. Additionally, the initiator's user visibility list must necessarily contain all members of the group (even ones who dynamically joined) because the synchronization process added all users' user visibility lists to the initiator's. In other words, all users must explicitly agree on which version of the document that a particular name corresponds to. The specifications of the Collabrador system ensure that a successful run of the commit algorithm guarantees correctness of the process. The only possible failure mode is if there is a new user who has never joined the same network as a pre-existing user, but this new user can be safely disregarded because it has never had the chance to obtain any version of the document, much less modify it.

The process for creating a commit is mostly the same as the process for creating an unnamed checkpoint. The only difference is that, during the synchronization process, users also exchange the name of the checkpoint and its associated hash. If the initiator successfully synchronizes with all users on the network, the initiator's leaf node is the same hash as at the start of the process, and the leaf node's user visibility list exactly equals the computer's known user list, then the commit is successful. If any of these conditions are not met within a pre-set number of passes through the synchronization round-robin, the commit fails.

Finally, it is worth noting what happens in the case of a failed commit. Imagine trying to commit a version named "final" and it fails, and then trying again. If a user in the system is asked for the version "final" before the second commit finishes, there is no guarantee as to what version of the document it will return. To avoid this problem, Collabrador requires that no two commit attempts use the same name. Because some users may not know the names of all failed commits, this restriction is enforced in practice, not in software. In other words, the human user needs to know that trying the same name twice can lead to undesirable results, but there are no software safeguards to prevent this situation from occurring.

2.2 Sync algorithm

Synchronization is a process which causes two computers with unique checkpoint databases to converge to a single checkpoint database. Each machine must begin and end with a checkpoint database in the merged state. The computer that initiates the synchronization process is called the *initiator*, and the other computer is called the *provider*. At a high level, the synchronization process performs three steps:

1. The provider sends its checkpoint database to the initiator.
2. The initiator, which is now unmerged, locally performs the merge algorithm described in Section 2.3.
3. The initiator sends its (now merged) checkpoint database to the provider.

One possible implementation of the synchronization process would be to have the provider send its entire checkpoint database to the initiator in the first step. However, it is usually the case that most of the provider's database is already known to the initiator. To minimize the amount of communication required, the provider only needs to send the checkpoint objects it has that the initiator does not have. The process for discovering which objects the initiator does not have is described below.

Let I be the leaf node of the initiator's database, let P be the leaf node of the provider's database, and let C be any common ancestor of I and P. There is guaranteed to be at least one common ancestor because, at the very least, the object representing the initial blank document is necessarily an ancestor of all checkpoint objects. Thus, the provider only needs to send the nodes "between" C and P. To further minimize the amount of the database transmitted, it's ideal for C to be the *least* common ancestor of I and P.

To find the least common ancestor in a distributed fashion, the provider performs the following algorithm: It sends P's hash to the initiator and asks if this object was already in the initiator's checkpoint database. If it was, then this step of the synchronization process is complete. However, Collabrador also needs to update P's user visibility list to include both the initiator and the provider (if they weren't already included). If P was not already in the initiator's database, then the initiator asks for the full checkpoint object corresponding to P, and then the provider sends both this and recusively sends the hashes of P's parents using the same process. This process will terminate upon reaching C. At this point, the initiator has the provider's entire checkpoint database.

Next, the initiator performs the merge algorithm using this most recently transmitted object C as the common ancestor, and sets the user visibility list of the resulting leaf node to contain only the provider and the initiator. Finally, the initiator sends its entire checkpoint database to the provider using the aforementioned process.

When there are more than two computers, Collabrador has them synchronize in a pairwise fashion. The implementation of Collabrador assumes that each user can query for the set of all users currently on the network. Every computer has a background process that, at random intervals, sends a request to another computer asking if it is willing to synchronize. The two conditions under which a computer will refuse a request are if it is waiting for a response from any computer about synchronizing (up to some time-out), or if it is currently synchronizing. Syncronization also ocurrs when a user saves or initiates a named checkpoint. When two computers agree to synchronize, the computer that initiated the request will play the role of initiator and the other computer will be the provider. When this process finishes, the initiator will wait a random interval and then send a request to the user currently on the network with the next-lowest IP address in a round-robin process. After a quadratic number of pairwise synchronizations, all computers in the network will converge on the same checkpoint database.

Additionally, whenever two computers synchronize, they also exchange their *known user lists*. All users, whether online or offline, are included. The purpose of this exchange is so that all users collectively can determine the exact group membership, even if users are dynamically joining the group. Collabrador does not, however, support users dynamically leaving the group.

2.3 Merge algorithm

The merge algorithm is based on the operational transform ("OT"), which is the conflict resolution technique used by all major collaborative software (notably including Google Docs and SubEthaEdit). We model a single OT as a sequence of non-conflicting "insert", "delete", and "move" operations. Note that every OT defines a transformation on character indices (although this transformation need not be injective or surjective), and all OTs are invertible. A full formal description of OTs is given in [1].

OTs can interact in several ways, which are shown in figure 1:

- Any number of OTs that are generated sequentially may be composed into a single OT representing the combined effect. We can thereby express an OT that represents the transformation between any commit and its parent, regardless of how many steps the user took to effect this transformation.

- We may use an OT, viewed as a transformation on character indices, to transform the indices of another OT. This process will fail if and only if there are conflicting changes in the two OTs.

- When two OTs are performed simultaneously, we may use one OT to transform the character indices of the second OT, and compose the result with the first OT. This process is commutative when there are no conflicts.

When the merge algorithm takes over, the sync algorithm has copied nodes between two merged DAGs, resulting in an unmerged DAG. The sync algorithm has also identified the lowest common ancestor of the two leaves in the unmerged DAG. We must merge these into a single leaf—automatically if possible, but with user intervention if necessary. Formally, given two OTs that take the same parent to different children, we wish to generate a single OT that applies both sets of changes to the parent in a logical way.

We simply follow the approach mentioned above: use one OT to transform the character indices of the second OT, and compose the result with the first OT. Conflicting portions of the second OT are ignored and flagged for manual user resolution. (Following the New Jersey design approach, we prohibit incorrect automatic merges, but allow unnecessary manual merges if it makes the implementation substantially simpler.) When conflicting changes are identified, they are flagged for manual resolution, but the algorithm continues to resolve subsequent non-conflicting changes. An example is shown in figure 2.

OT primitive operations

An OT is defined as a composition of the following primitive operations:

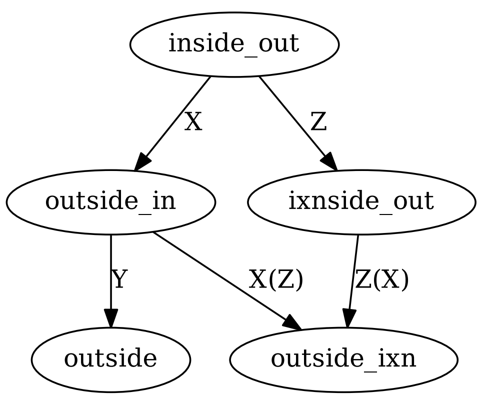
$$\begin{gathered} \mathit{insert}(\mathit{substring}, \mathit{index}) \\ \mathit{delete}(\mathit{start\_index}, \mathit{end\_index}) \\ \mathit{move}(\mathit{start\_index}, \mathit{end\_index}, \mathit{new\_loc}) \end{gathered}$$

Example strings

$$\begin{aligned} A &= {}^{0}\text{i}^{1}\text{n}^{2}\text{s}^{3}\text{i}^{4}\text{d}^{5}\text{e}^{6}\text{\_}^{7}\text{o}^{8}\text{u}^{9}\text{t}^{10} \\ B &= {}^{0}\text{o}^{1}\text{u}^{2}\text{t}^{3}\text{s}^{4}\text{i}^{5}\text{d}^{6}\text{e}^{7}\text{\_}^{8}\text{i}^{9}\text{n}^{10} \\ C &= {}^{0}\text{o}^{1}\text{u}^{2}\text{t}^{3}\text{s}^{4}\text{i}^{5}\text{d}^{6}\text{e}^{7} \\ D &= {}^{0}\text{i}^{1}\text{x}^{2}\text{n}^{3}\text{s}^{4}\text{i}^{5}\text{d}^{6}\text{e}^{7}\text{\_}^{8}\text{o}^{9}\text{u}^{10}\text{t}^{11} \end{aligned}$$

(Superscripts denote character indices.)

String/OT relationships



Example OTs

$$\begin{aligned} X &= \mathit{move}(5,8,0) \circ \mathit{move}(0,2,10) \\ Y &= \mathit{delete}(7,10) \\ Z &= \mathit{insert}(1, \text{x}) \end{aligned}$$

We have $X(A) = B$, $Y(B) = C$, and $Z(A) = D$. The complete diagram of relations is given in figure

Composition

The composition $Y \circ X$ is performed in the obvious way, and we have $(Y \circ X)(A) = C$.

Parallel composition

When two users perform diverging modifications (for example, X and Z to A), we merge them by transforming the transform. For example:

$$\begin{aligned} X(Z) &= \mathit{insert}(9, \text{x}) \\ Z(X) &= \mathit{move}(5,8,0) \circ \mathit{move}(0,3,11) \end{aligned}$$

We would merge the users' changes as follows:

$$(X(Z) \circ X)(A) = (Z(X) \circ Z)(A) = \text{outside\_ixn}.$$

This process is formally described in [2].

Merge conflict

The quantities $(Y \circ X)(Z)$ and $Z(Y \circ X)$ are not defined. Both transformations of indices fail because there are conflicting changes.

Figure 1: Operational transform examples.

3 Analysis

3.1 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, your design must not require resolving conflicts for the changes to different paragraphs.

Alice and Bob will not have to resolve conflicts when the change the document in different paragraphs. For example, if transform $U = \mathit{insert}(12, \text{‘a whole bunch of text’})$ represents Bob adding text to paragraph one, and $V = \mathit{insert}(422, \text{‘more text here!’})$ represents Alice adding text to paragraph two, the resulting merge would be

$$U(V) \circ U = \mathit{insert}(434, \text{‘more text here!’}) \circ \mathit{insert}(12, \text{‘a whole bunch of text’}), \text{ or}$$

The scenario

Start with the document "abc", and consider the following OTs:

$$U = insert(3, \mathrm{z}) \circ insert(1, \mathrm{x})$$
$$V = insert(1, \mathrm{y}) \circ insert(2, \mathrm{z})$$

So $U(\mathrm{abc}) = \mathrm{axbzc}$ and $V(\mathrm{abc}) = \mathrm{aybzc}$. The user must manually resolve the"x" vs "y" merge conflict, but "z" should be merged automatically.

Merge difficulties

Both users have made the same change (inserting "z"), so a merge conflict should not be generated for that character. The users also made a conflicting change ("x" vs "y"), but one did so before inserting "z" and one did so after inserting "z".

The merge

First we compute $U(V)$. $insert(1, \mathrm{x})$ transforms $insert(2, \mathrm{z})$ to $insert(3, \mathrm{z})$. We attempt to apply the coordinate transformation of $insert(1, \mathrm{x})$ to $insert(1, \mathrm{y})$ but this fails with a merge conflict. We then construct U' and V' with the conflicting primitives removed, using the inverse transform to determine that $U' = V' = insert(2, \mathrm{z})$. Now we can compute the coordinate transformation $U'(V')$: we transform $insert(2, \mathrm{z})$ by $insert(2, \mathrm{z})$. Although the insertion coordinates conflict, since they insert the same string, we merge them into $insert(2, \mathrm{z})$. We apply this to "abc", resulting in "abzc". Finally, we present the user with "abzc" along with the conflicting OTs (which have been transformed to the new coordinate space): $insert(1, \mathrm{x})$ and $insert(1, \mathrm{y})$.

Figure 2: An especially difficult merge conflict.

$$V(U) \circ V = insert(12, \text{'a whole bunch of text'}) \circ insert(422, \text{'more text here!'}).$$

When Alice and Bob modify the same sentence, with Alice's changes represented as X and Bob's changes represented as Y, their changes will require conflict resolution if $X(Y)$ tries to compose two OTs that operate on the same index (see Figure 2). Otherwise, the conflict will be automatically resolved as before.

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.

After Alice and Charlie meet, Charlie's graph contains the conflict resolution. When Charlie and Bob meet, the merge algorithm detects that Bob's DAG is a sub-graph of Charlie's, so Bob's document and DAG can be updated to be identical to Charlie's. See Image.

TODO Image: Conflict resolution between Alice, Charlie and Bob.

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 will be required. Alice's OTs are $X = move(12, 422, 1022)$, etc. The OT of Bob's changes is $Y = insert(222, \text{"spurious text generation"})$. When Bob meets Alice, his Collabrador client will synchronize with her, and merge their text. The resulting transform will be

$$Y(X) \circ X = insert(822, \text{"spurious text generation"}) \circ move(12, 422, 1022)$$

The index 822 is determined as follows: index 1022 becomes index 612 because $1022 - (422 - 12) = 612$. The original insert, at location 222, is 210 indices away from what is now index 612, and $612 + 210 = 822$. An equivalent result can be achieved by

$$X(Y) \circ X = move(12, 448, 1048) \circ insert(222, \text{“spurious text generation”})$$

where the *new_loc* and *end_index* have been incremented by 24 (the number of characters in the inserted string).

Two users, while not connected to each other, find a spelling mistake and correct the same word in the same way.

Since the two versions have the same parent, change log and content, the hashes will be the same, and they will be recognized as the same version, making conflict resolution unnecessary.

3.2 Failures

Each Collabrador client keeps the list of users associated with a document, and tracks whether or not they are reachable. A commit cannot successfully begin unless all users are reachable. If any user is disconnected during the commit process, the commit fails. The commit may have created a node on some of the user' computers if it partially completed. To avoid conflicts, Collabrador requires each attempted commit to have a unique name. This ensures that no user will ever have a commit with the same name in their DAG. If the commit process notices that the commit name is already contained in a user's DAG, the commit aborts immediately.

Synchronization can occur when a user's Collabrador client sees one or more other users online, and it is a much more flexible process than committing. When user A decides to sync, a checkpoint is recorded to the local graph. If user A's machine crashes while saving the checkpoint to the local graph, the checkpoint is lost (all-or-nothing). If user A's machine crashes after saving the checkpoint locally, but before initiating the sync process, then Collabrador client will send out a sync request when user A comes back online. If user B crashes afterbeginning to send sync data to A, user A will terminate the sync process due to timeout and discard the partial data. If any user goes offline after he finishes syncing with at least one machine, but before all pairwise syncs have completed, the sync process will go on amongst all other machines. The booted user will re-synchronize during the integration process when he comes back online.

We use TCP for reliable and accurate packet delivery, so we do not have to worry about data corruption at the application layer.

3.3 Performance

3.3.1 Performance of the sync algorithm

3.3.2 Performance of the merge algorithm

The time taken by the merge algorithm grows linearly with the number of changes between the two leaves and their lowest common ancestor. Assuming that it takes about a thousand CPU cycles to process a single OT primitive, the process of merging two encyclopedias would take less than a millisecond. Thus, we are not especially concerned with performance of the OT merge algorithm. (Moreover, the time taken by the merge algorithm is negligible compared to the time taken by a user to manually resolve a conflict.)

Using the naive approach, opening a merged DAG for editing takes linear time in the size of the document history, since the complete set of OTs must be layered upon the root of the DAG. This can be mitigated by caching file contents for each node in the DAG, and selectively reordering OTs to minimize the number of primitives required by an operation.

3.4 Design Tradeoffs and Flaws

Having no centralized server makes Collabrador a more reliable document editing system, because there is no single point of failure. It also allows for increased flexibility, as users can edit while connected to the internet, ad-hoc, or offline. However, because Collabrador is distributed, synchronizations are pairwise, making the sync algorithm quadratic with respect to the number of users. A centralized server would have asymptotically linear synchronization.

Some collaborative editors use version vectors: a vector containing the last known version number for each user. For example, $V = (1, 2, 2)$ means that the first user has version 1 of the document, and the other two have the second version. This is a compact and nifty way to represent version data. V_1 is newer than V_2 if all of V_1's elements are greater than or equal to V_2's. If one vector is not clearly older than another, e.g. $V_1 = (1, 2, 3)$ and $V_2 = (3, 1, 4)$, then there is a conflict. A DAG is more visually informative than a version vector and provides an intuitive way for users to visually track the document history. The DAG allows us to perform OTs and use a highly autonomous merge algorithm. A system using version vectors would require the user to merge conflicts more frequently, since less information is available. The cost, however, is that the DAG grows linearly with edit history. The more frequently a document has been committed, the more overhead there is. Version vector overhead only increases as the number of users editing a document increases.

We support only named checkpoints or dynamic membership, not both. This is because the ability to delete users greatly complicates things. For example, if user A is deleted, and broadcasts while only user B is available, but then user B is deleted before he can pass on the word about user A. This presents problems for commits, because all users are required to be online. Also, if a user were allowed to join while a commit was underway, the initiator might receive a visibility list that does not include the new user, and think that it is correct because his client was unaware of the new user. There are ways around these problems, but this is a challenge we have left for later releases. The feature was not included in the first release because static groups greatly simplify the design, and allows Collabrador to have stronger guarantees about correctness.

Finally, it is worth noting that Collabrador does not have an authentication system, so it is very simple for a user to send false changes.

4 Conclusion

Collabrador is a sophisticated collaborative document editing system that gives users flexibility in connectivity. It uses industry-standard operational transforms to merge changes, and has a failure-resistant synchronization process. Problems that remain to be solved include dynamic membership and adding an authentication system.

References

[1] D. Wang, A. Mah, and S. Lassen. (2010, July) Google wave operational transformation. Google. [Online]. Available: http://www.waveprotocol.org/whitepapers/operational-transform

[2] M. Ressel, D. Nitsche-Ruhland, and R. Gunzenhäuser, "An integrating, transformation-oriented approach to concurrency control and undo in group editors," *Proceedings of the 1996 ACM conference on computer supported cooperative work*, 1996.

[3] M. Stonebraker and L. Rowe, "The design of postgres," *Proceedings of the 1986 ACM SIGMOD international conference on management of data*, 1986.

4622 words