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

Users have the ability to edit documents offline. Once the user comes back online Collabrador synchronizes with other online users and merges the changes made while offline. Direct connectivity is also supported, using ad-hoc networking and Bonjour to discover other Collabrador users. When a user joins the internet (or an ad-hoc network), his Collabrador client will integrate him into the network, merging the latest commit into his local copy.

A checkpoint occurs automatically when a user saves, and is named by hash of the document, provenance graph, and change log. Checkpoints happen when a user fully connected, offline, or connected ad-hoc. The hash for a checkpoint will be unique unless users make the same changes in the same way, thus there will always be agreement as to which document version corresponds to a version name. We will also allow user-provided comments e.g. "Ben's final draft", however the user is responsible for ensuring that these are useful.

Commits are a special type of checkpoint that a user manually initiates. The commit process involves merging changes from all users into one agreed-upon document. Merges are autonomous unless there has been two or more edits at the same point, relative to a common ancestor. This is called a conflict. If conflicts occurs, the machine running the merge algorithm will ask the user to resolve them by hand. Once each user's changes have been merged, all users must approve the resulting document before the commit is finalized.

2.1 System architecture

On each computer, Collabrador consists of a **text editor** and a **checkpoint database**. Rather than storing the final 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**. [TODO: cite OT paper] [TODO: think about whether the user should explicitly click "save" or if this should happen in the background]

To be more specific, the checkpoint database is a dynamic perfect hash table of **checkpoint objects**, keyed by the SHA-1 hash of the object. 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 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. 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 instead. The challenge with Collabrador is to maintain this directed acyclic graph in a distributed fashion because there is no central server. Collabrador's solution is for each user to store in its local checkpoint database only the portion of the graph that it knows about while a synchronization process in the background lets users constantly communicate with each other to share and exchange parts of the graph.

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 [TODO: cite the other section], 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.

2.2 Sync algorithm

Synchronization is a process which causes two computers each with a merged checkpoint database to converge to the same merged checkpoint database. 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 initiator.

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.

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. 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 another 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. This 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:

$$insert(substring, index)$$
$$delete(start\_index, end\_index)$$
$$move(start\_index, end\_index, new\_loc)$$

Example strings

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

(Character indices are shown for clarity.)

Example OTs

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

We have $X(A) = B$, $Y(B) = C$, and $Z(A) = D$.

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:

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

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.

2.4 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. provide quick operations and fault tolerance. 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 Post-

The scenario

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

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

So $U(\mathrm{abc})$ = axbzc and $V(\mathrm{abc})$ = 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.

greSQL [3].

2.5 Checkpoints

In order to support named checkpoints, known as **commits**, Collabrador needs one additional data structure. Each user should store a hash table mapping 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.

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.

The specifications of the Collabrador system ensure that a successful run of the commit algorithm guarantees correctness of the process. Evidently, 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. 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.

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.

3 Analysis

3.1 Design requirements

TODO by colleen

3.2 Performance

TODO by dlaw and jhurwitz

3.3 Drawbacks

TODO by colleen

4 Conclusion

TODO by colleen

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.

TODO WORDCOUNT words