Raph Levien Follow
Raph Levien is a software engineer on the Fuchsia team, and creator of xi-editor. Opinions here are his ...
Jul 6, 2016 · 10 min read

Towards a unified theory of Operational Transformation and CRDT

Update (2 Mar 2017): there is now working code for the ideas in this post, with additional optimizations.

Lately I've been diving deep into the literature on collaborative text editing. There are basically two opposing camps: Operational Transformation, which dates back to the 1989 Grove system and is the basis of most collaborative editors today, and the much newer Conflict-free Replicated Data Types (CRDT) approach. However, I've come to realize that there is a common theory which unifies at least a subclass of OT and CRDT's. I believe the two camps have potentially much to learn from each other.

The so-called Transaction Property 2 (TP2) is something of a faultline running through the OT literature. A landmark 1996 paper by Ressel et al introduced the property, proved that it was necessary and sufficient for correctness of their system, then proceeded to propose a set of operations for editing strings that, in hindsight, did not satisfy TP2. Oops. For a decade or so, TP2 was something of a philosopher's stone, with several alchemists of OT claiming that they had found a set of transforms satisfying TP2, only for counterexamples to emerge later. Only with the 2006 publication of Tombstone Transformation Functions was actual gold struck, but at a cost; instead of doing insert and delete operations directly on the string as in the original OT vision, the document state is augmented with *tombstones* that record the locations where deletions would happen. A separate *read function* is required to erase the tombstones and yield the user-visible string.

Incidentally, the authors of the tombstone paper were, at the same time, working on WOOT ("WithOut Operational Transformation"), which is now seen as an early instance of CRDT. The CRDT approach uses a simple merge function, basically a generalization of set union (more formally, monotonic operations over a semi-lattice, which has a commutative, associative, and idempotent *join* operator), and an even more complex read function to reconstruct the string from the document state. Even so, it is appealing because the simple merge function means that it is very easy to reason about convergence.

Almost all practical implementations of OT forego TP2, and solve the problem by limiting concurrency in some way, generally requiring a central server to decide on a canonical ordering of the operations (but still using transformations to let clients apply operations out-of-order just a little bit). In fact, practical OT today is based firmly on the 1995 Jupiter system, with approximately the last 21 years of academic work having essentially no impact. I should mention Google Wave here, which restricts concurrency even more (only allowing a single operation from each client in flight). This approach simplifies the construction of servers, but still has basically the same concurrency and operation model as Jupiter. Jupiter and Wave are based on a strict serialization concurrency model, while both CRDT's and OT with TP2 are eventually consistent. A good paper is *Specification and Complexity of Collaborative Text Editing,* which also characterizes the former as satisfying the *weak list specification,* while the latter satisfies the *strong* counterpart.

Adopting TP2 eliminates the need for the central server, allows peer-to-peer configurations, and can reduce latency (including eliminating the need for Wave-style "batching" of operations). It's possible to implement OT with TP2 reasonably simply and efficiently, but this fact doesn't seem to be much appreciated in the oral tradition of people actually building OT systems. I'd like to try to change that, a bit.

An OT system satisfying TP2 has properties very similar to CRDT-based text editing. I'll argue that this is no coincidence, that a unified theory underlies both.

I'll try to explain an idealized OT system, closely based on GOTO (Generic Operational Transformation Optimized) and SOCT2 (Sérialisation des Opérations Concurrentes par Transposition). This will necessarily be a high-level overview, and for a full understanding readers are directed to those papers (with considerable sympathy from me, the OT literature as a whole is difficult).

This OT system represents the document state as a *history log,* which is at heart a sequence of operations. The key to OT is that each operation has a *context,* basically the set of operations that precede it in the log, and that contexts always match up. When merging an operation from another node's document state, the context in general won't match up, so the operation needs to be *transformed.* The standard notation for this is IT(Oa, Ob), where Oa and Ob have the same context, yielding a transformed O'a. This transformed operation additionally has Ob in its context, but does the same thing as the

original Oa. In this notation, IT stands for “Inclusion Transformation”. Early OT systems used just IT, but this approach requires keeping the original transformations, some explicit representation of context, and a lot of juggling.

Transformation of operations changes their contexts, but it also makes sense to track their *original context* (that is to say, not affected by the transformation). When one operation appears in the original context of a second, we say that the second is *causally dependent* on the first. When neither appears in the other, we say they are *concurrent.*

SOCT2 and GOTO use a second transformation, often written ET (Exclusion Transform. It’s basically an inverse of IT, and *removes* one operation from an operation’s context. I prefer transpose(Oa, Ob), where the context of Ob is that of Oa plus Oa itself, yielding O’b and O’a in which the context relation is reversed. The context of O’a is that of O’b plus O’b itself. You can write transpose in terms of ET: transpose(Oa, Ob) = [ET(Ob, Oa), IT(Oa, ET(Ob, Oa))].

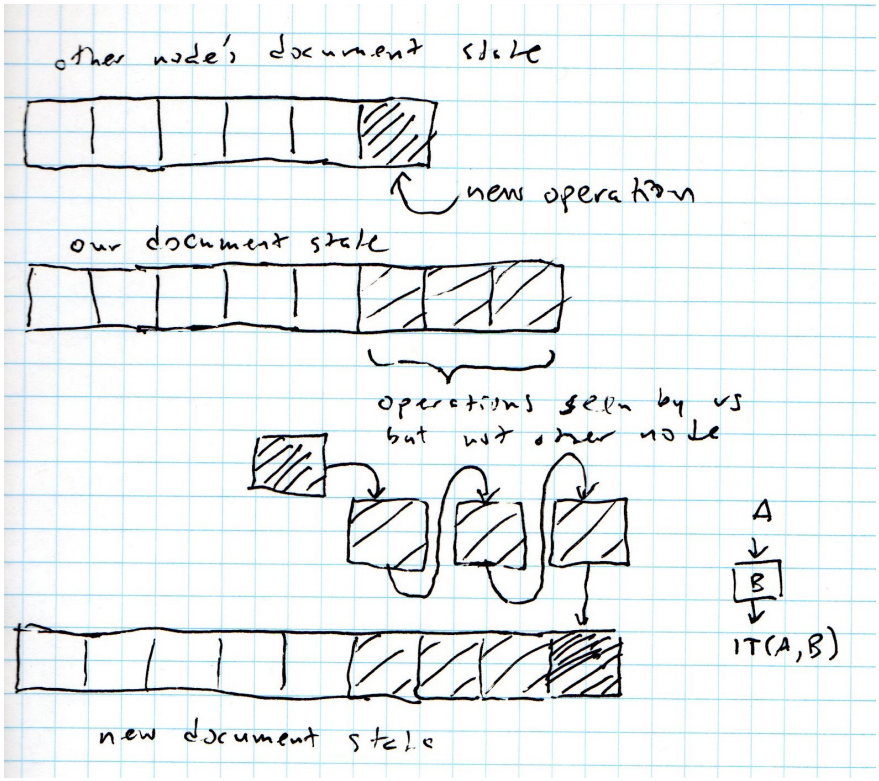
The idea of these two algorithms is that doing the transpose doesn’t affect the semantics of the history log as a whole. Also, you don’t need the original operations, so it can be done more simply and efficiently. The correctness of these algorithms relies on the fact that you only do the transpose on concurrent operations, that is to say all the permutation respects the partial order of original context. Of course, proving such a thing is not easy. Frankly, I didn’t really understand the proofs in those papers, and didn’t really trust that it was true until I did automated randomized testing on my own implementation.

It’s also important to note that the TP2 property is required for all this to work. Otherwise, you’ll find that while the transposes don’t affect the document, they do affect how an operation is transformed through a sequence of other operations. TP2 gives you exactly the guarantee you need: as long as the two operations are concurrent with respect to their original context, the transpose basically doesn’t change anything; there’s no way to observe the difference.

I’ll dive into a little more detail. Consider a peer-to-peer model where every node has its own copy of a document state. Each peer is continually sending updates to all its neighbors (the topology can be anything from a star to a spanning tree to a complete graph). The communication is assumed to be in-order, but of course is subject to unpredictable delays (essentially the same communication model of

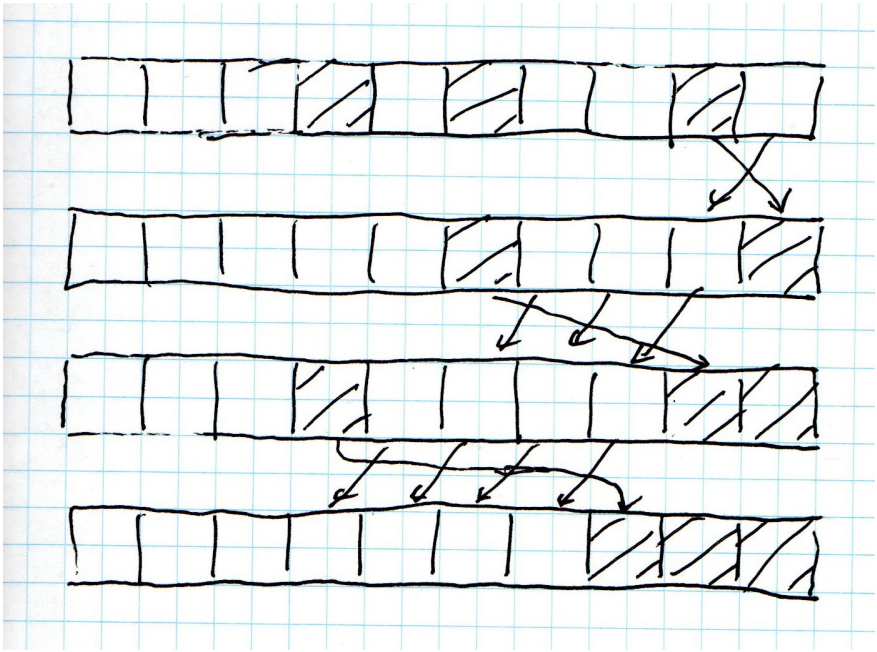
TCP/IP and WebSockets). To simplify, an update is considered appending one operation to the node's document model.

When a node receives an update from another node, it's already received all preceding updates (thanks to the in-order nature of the communication channel), and has merged these into its own document state. Its document state contains all operations of the other node's document state, plus additional operations that the other node had not (yet) seen at the time it sent the update. If, in a lucky coincidence, these operations are clustered so all those in the second set come after those in the first, doing the transformations is easy, just do an IT transformation of the new operation through each of those in sequence. This figure shows it graphically:



Here, the white squares represent operations common to both document states. The three light shaded squares are additional operations we have that the other node hasn't seen. The dark shaded square is the new operation from the remote node that we want to integrate. As stated above, we just run this new operation through an IT transform of the three operations we have that the other node doesn't. If you're following along with the GOTO paper, the sequential application of IT transforms is the LIT function, and the above picture corresponds to case 2 in their Algorithm 2.

The above is the case where these light-shaded operations are clustered together. Note that this can be enforced when there are only two concurrent users (basically the case that the original Grove system handled correctly). Things get more complex when there are three or more. The central idea of GOTO and SOCT2 is to *permute* the history log, using transpose operations as defined above, to establish that clustering. Without presenting the algorithm in detail (see the papers), the following figure should give a good idea what's going on.



Here, each point where two arrows cross is a transpose operation. If you're following along with the GOTO paper, each line in the above figure corresponds to one invocation of their LTranspose function, and the above picture corresponds to case 3a in their Algorithm 2.

The concept of transposing and permutation is also, I think, the key to analyzing these OT algorithms in the CRDT framework. Define an *equivalence relation:* two document states are equivalent iff one is a permutation of the other, and all swap operations comprising the permutation respect the partial order induced by original context. Under this equivalence relation, the merge operation on two document states satisfies the semilattice property required by CRDT; merge is commutative, associative, and idempotent. Proving that formally is an exercise for the reader, but it basically follows from the correctness proofs of GOTO and SOCT2 (again, the reader has my deepest sympathies).

There's one small sleight-of-hand here. The equivalence relation above is defined on mathematical objects in which each operation contains its original context. Let's say you've got two document states in which one user has typed "a" and the other user has typed "b". If these operations happen concurrently, then the document states [Oa, O'b] and [Ob, O'a] are considered equivalent. However, if the second user waits for the first operation, then they're not. The GOTO and SOCT2 algorithms don't carry this extra metadata around in the document state. Even though that means they're missing information needed to evaluate the equivalence relation, from a practical perspective I consider that a good thing. In fact, I'd say that one of the strongest criticisms of CRDT is that it generally requires a lot of metadata to build a tree or graph structure to make merging easier, which is then erased in the read function. Keeping that around requires more RAM and more communications bandwidth, compared to more OT-flavored approaches.

You could hand-wave and say that a theoretical model of GOTO/SOCT2 that includes the context metadata is a CRDT, and that erasing this data is merely an implementation detail, but to me that's unsatisfying. If the definition of the equivalence relation includes the original context, then the document state of a GOTO/SOCT2 algorithm is not a CRDT under that definition.

There is an alternative, which I personally find more appealing. Instead of saying that two operations can only be transposed if they were in fact concurrent, loosen the definition. Basically, if the history can be explained by two concurrent operations, allow the swap. In the example above, let's say that the tie-breaking rules place edits from user A before those of user B on concurrent insert. So "a" and "b" entered concurrently result in "ab". Allow "ab" to be swapped whether the operations were in fact concurrent or not, as there is no difference in the observable state. There are no two *concurrent* operations that can yield "ba", so that pair would still not be swappable.

More formally, consider the sequence [Oa, Ob] to be transposable iff IT(Oa, O'b) = O'a, and IT(O'b, O'a) = Ob, where [O'a, O'b] = transpose(Oa, Ob). In the "ba" example above, this would fail, as Oa and O'b would be concurrent inserts of "a" and "b" at the same location, and the tie-breaking rule in IT would resolve those to "ab", not a match.

Since this new equivalence relation is defined entirely on information actually contained in the document state, the GOTO/SOCT2 merge

operation can then rightfully be considered a CRDT (assuming no correctness mistakes, of course).

Further, I'd say that the looser equivalence relation is actually useful, not just a mathematical trick. For example, you can permute the document state so that it compresses better. If two users are concurrently typing in distant parts of the same document, then the original context induces nearly a total order (actually a total order if latency is fast enough), but in reality the edits of one user can be transposed with the edits of the other without harm. The looser equivalence relation lets you do that.

What of non-TP2 operational transformation? I've thought about ways of jamming that into the CRDT framework. In particular, you can attach a timestamp to each operation, then use a read function to play the operations back in timestamp order. In practice, while this might be a CRDT in some technical sense, it's a really bad one. If you integrate an operation with an early timestamp relative to the rest of your buffer, then it can force replay of a lot of operations, which is both slow and can make weird changes to the document. In particular, situations like TP2 puzzles (see Fig. 3 of the tombstone paper for a great explanation) can cause the reordering of big chunks of text, which would usually be a terrible experience. If these are your transformation functions, it's better to rely on a central server to actually limit concurrency so that doesn't happen.

Thus, I assert that the TP2-conforming subclass of OT can be considered an instance of CRDT, and so the OT and CRDT camps aren't as far apart as maybe they first seemed. The OT camp offers efficient implementation techniques, in particular teaching that retaining graph-structure metadata is not absolutely required. In turn, the CRDT camp offers a sounder basis for mathematical reasoning (the track record of the OT literature certainly demonstrates that it is very difficult to reason about the correctness of OT transformations) and a potentially rich set of techniques for composing and combining CRDTs.

I hope this blog post suggests ways for the OT and CRDT communities to work together. I'd also encourage designers of collaborative editing systems (and academic researchers) hoping for the benefits of CRDT (in particular, peer-to-peer operation rather than requiring a centralized server) to take another look at OT. Just because there are so many flawed proposals in the literature is not a good reason to

write off the entire field in favor of the seemingly greener pastures of CRDT.