CS7610: Final Project Report

Due on December 3, 2018

Professor Cristina Nita-Rotaru

Ben Weintraub and Jaison Titus

1 Introduction

For our final project, we have chosen to implement a collaborative document editor. Our implementation follows an algorithm called REDUCE, presented in 1998.

2 Context

The idea of computer supported cooperation work (CSCW) was a popular area of research in the late 1980's and 1990's. Until 1989, two methods were used for sharing documents, 1) locking; and 2) transactions. If locking was used, then when one user went make an edit, a lock was placed on the document to prevent concurrent and possible destructive edits. There are various optimizations for locking, like locking only a part of the document, but in the end, it is not really possible for the system to know when a user is done editing or just pausing and so locking is not a good strategy. Transactions were proposed as a strategy, and can be effective. They have a cost however, because to execute a transaction in a distributed system, there will need to be at least a two phase commit among all participants, and conflicts may need to be resolved by some sort of consensus. While possible, this means that an edit made locally by one user cannot show up in their document until the transaction completes, resulting in unusable latency for document editing software. In Concurrency Control in Groupware Systems (Ellis et al., 1989) an idea is proposed called Operational Transformation. The high level idea is the following. Any edits that are made locally show up immediately in the editor, and then are sent out to the other peers. The other peers perform a transformation on the operation such that the original operation can be applied to a potentially different state on each peer. For instance, imagine two users concurrently delete the first character in a document. If the operation was applied immediately at both sites, and then applied again after receiving the same operation from the other peer, then two characters would have been deleted, when clearly both users only wanted one character deleted. Operational Transformations are one way to solve this problem and guarantee that each peer's view of the document converges to the same state. The same paper proposed an algorithm called dOPT (distributed Operational Transformation), and our goal was to implement this.

3 Our Experience

Our classmates, Connor and Giorgio, who were working on a similar concept, made the discovery early on that the dOPT algorithm is incomplete. What this means is that in certain situations having to do with concurrent edits at different sites, operations can be transformed incorrectly. This leads to divergence. As a result of this major issue, we were forced to look into the research and find an alternative algorithm to implement. Looking to maintain the peer-to-peer aspect, we chose an algorithm called REDUCE (Sun et al., 1998). The REDUCE algorithm introduces two new ideas are essential to a correct Operational Transformation (OT). The first is what the authors call "undo/do/redo". This means that in order to guarantee convergence, an operation must be applied to a document in the same state as the one in which it was generated. So, if two sites have a series of operations that they have done locally *after* a particular operation was generated on another peer (according to a vector clock that is passed with each operation), then all of those local operations must be undone, so the remote operation can be transformed and applied, and the the undone local operations themselves must be transformed and reapplied. The other idea is that there are actually two different types of transformation functions that are called in different contexts, they are "inclusion transformations" and "exclusion transformations". Inclusion transformations are roughly equivalent to the transformation functions in dOPT. They are applied when an operation, O_a, should be transformed against O_b in such a way that O_b's effect on the document is accounted for in O_a. Exclusion transformations are a new kind of transformation that are needed as part of the undo/do/redo scheme. They are applied when an operation, O_a, should be transformed against O_b in such a way that O_b's effect on the document is not

included in O_a. This new type of transformation is necessary because it guarantees "intention preservation". Intention preservation means that after an operation has been applied on a remote node, the same effect should have happened as on the originating node. From the concurrent delete example above, this would mean that only one of the deletes actually changed the document state–the other would be converted into a nop. These two changes in reduce are necessary and sufficient for correctness.

4 Challenges

Having decided to implement REDUCE, we came to discover that the algorithm is quite complicated. Each operation is passed around to the other peers which themselves generate a slew of new operations as part of the undo/do/redo and transformations. This created many difficulties in debugging, because even if we were to extract the current history of operations that were applied by each peer, it would not be sufficient to know that the document would be converging, because each peer will have executed differently transformed operations based on their own unique document state at the time messages were received from other peers. Testing our program also had its complications because of the complexity of the algorithm. REDUCE has many execution paths, and some of them depend on the timing of relative operations (in terms of their vector clocks). Forcing the timing to meet the constraints of a particular execution path was non-trivial. The most effective testing method we found was to randomly generate operations, but that is not targeted and meant that we were unable to reproduce bugs by rerunning the testing environment.

There was also an additional element of uncertainty with the algorithm. No implementation was provided by the authors, and this is significant for an algorithm that took 50 pages to describe, because we have no assurance that it is really correct. This is something that we were already on edge about following our discovery about dOPT being incomplete.

Had we been more aware of the complexities of the REDUCE algorithm and the state of the art, we would have chosen a different approach. In particular, using Conflict-free Replicated Data Types (CRDTs) would have led to a better implementation. A CRDT could be used to track additional metadata for each character in the document. This would greatly simplify the operational transform logic, almost completely removing it from the picture.

5 System Architecture

At a high level our architecture consists of two classes of independent components: servers and clients. Servers are where the main processing is happening, and each of the peers is communicating with each other to perform the operational transforms. The clients are lightweight interfaces that each interact with their local server. In practice, a single node will contain both a client and a server.

5.1 Peer Server Architecture

The peer architecture includes three key components, it has an event handler that both receives and sends client operations, another event handler that does the same for peer operations of other peers, and an operational transform processing module. The event handlers send and receive operations as messages that include the operation type (insert or delete), the character being inserted or deleted, and the position of the character involved. The peer to peer messages also include a vector clock which records the time at the peer when the operation was generated. The operational transformation processing module is a little bit dense to

read. This is because we opted to retain the same abbreviation-laden notation as the REDUCE algorithm as specified in the paper.

5.2 Client Architecture

The client serves two purposes. It updates the document according to the latest document state as provided by the server, and it sends operations generated by the user to the server, which then gets multicast to the other peers. We chose to separate out the clients so that we could support multiple interfaces. The two interfaces we have implemented are an Emacs minor mode and a small javascript browser app.

5.2.1 Emacs Plugin

The Emacs plugin was implemented in a language called Emacs Lisp, which was required. Lisp is an unusual class of language so there was some learning involved here. However, in the end, it is easy to enable this mode for a document, requiring only a short Emacs command.

5.2.2 Javascript App

The Javascript app is a simple web app made using the React framework. It uses Socket.Io to connect to its respective server and communicate with it.

6 Test Cases

We build a series of test cases using docker containers. Using docker compose, we created five peers and a client for each of them. The clients generate random insert and delete operations at random positions, and send those to their respective peers who relay it to the rest of the network. After a configurable number of operations have been generated and applied to each client. The final state of the client can be seen on stdout, and if they are the same, then the documents have converged.

7 Implementation Issues

The implementation of this project turned out to be much more complex than we had anticipated. As a result, there are a handful of edge cases that we were unable to handle. As defined in the paper, the REDUCE algorithm operates on strings–not characters–however we felt that that would incur additional complexity, so we opted to modify the algorithm to handle only character-wise operations. Having diverged from the algorithm, there were certain edge cases for which the algorithm would have to be redefined. As a result, we did not handle those cases. Those cases are rare enough however, that we don't hit them very often.

Another problem we ran into which was not discussed at all in the paper was how to keep the cursor in the correct position. The cursor may need to move if a character is added or deleted earlier in the document. It is not trivial to track this though, because the undo/do/redo algorithm means that a whole series of transformations and document mutations must happen in the background, so the actual changes that the client sees may not be incremental.

8 Conclusion

Though challenging, this project proved to be an excellent learning opportunity. Aside from learning the intricacies of operational transformations, we learned a lot about the Go programming language a well as research in general. Our research learnings basically revolve around our perceptions of trust. We learned that that just because a paper is published, does not necessarily mean it's correct. In searching for a replacement for the dOPT algorithm, we also learned that the state of the art includes many disparate contributions, and it is useful to get a broad understanding of what's out there before choosing a path to walk.

9 Sources

- Ellis, Clarence A., and Simon J. Gibbs. "Concurrency control in groupware systems." Acm Sigmod Record. Vol. 18. No. 2. ACM, 1989.
- Cormack, Gordon V. "A counterexample to the distributed operational transform and a corrected algorithm for point-to-point communication." University of Waterloo Technical Report (1995).
- Sun, Chengzheng, and Clarence Ellis. "Operational transformation in real-time group editors: issues, algorithms, and achievements." Proceedings of the 1998 ACM conference on Computer supported cooperative work. ACM, 1998.
- Sun, Chengzheng, et al. "Achieving convergence, causality preservation, and intention preservation in real-time cooperative editing systems." ACM Transactions on Computer-Human Interaction (TOCHI) 5.1 (1998): 63-108.