ORGanized Interactions: a tree-based collaborative editor

Andrew Xia [*]
Massachusetts Institute of Technology
axia@mit.edu

Slava Kim [*]
Massachusetts Institute of Technology
imslavko@mit.edu

Kevin Lu [*]
Massachusetts Institute of Technology
kezilu@mit.edu

ABSTRACT

In this paper, we discuss our implementation of Operational Transforms using a tree-based structure. We have successfully implemented a command line interface text editor, such that multiple clients can collaborate on a single document and attain eventual consistency in the document through the use of operational transforms.

1. INTRODUCTION

Collaborative editors like Etherpad tend to implement Operational Transforms to allow multiple users make concurrent changes to data. So far people have been successful in making flat-text editing collaborative but the space of the hierarchical text editing is yet to see a collaborative version. In this project, we first build a flat collaborative editor that uses operational transforms. Building on top of this flat structure, we also explore a version of a collaborative editor with a tree-like structure containing text.

In our operational transform system, allow operations for work on objects forming a tree shape including: editing fields, appending nodes, deleting nodes, reordering nodes on the same level, moving node to a different subtree. We would draft inspiration from OT-related papers and ShareDB implementation.

Working on trees as opposed to text is interesting, because structured text tends to be represented as a tree of data (e.g. HTML DOM).

In this project, we first build a collaborative editing system that would only allow text to be inside the shared document, much like etherpad. Following this, we would also like to incorporate a more complex tree-like structure to the text, which would allow for editing nodes. Future work on further optimizing our collaborative editor include exploring ideas such as sharding the data for better scaling, or using a distributed consensus algorithm such as Raft to ensure fault-tolerance on the server side.

[*]Department Electrical Engineering and Computer Science

2. RELATED WORK

Many collaborative text editors exist, but we were most influenced by Google Wave [1]. Google Wave takes the 1995 Jupiter paper [2] and implements a variety of performance optimizations. We have not implemented some but not all of the ideas from the Google Wave whitepaper.

Literature such as [3] and [4] suggest methods to modify Operational Transforms to include tree-like primitives. However, both papers have certain drawbacks (for example, does not support node splitting). Thus, after some consideration, we chose to more closely follow the Wave implementation.

3. COMMAND LINE INTERFACE

Since every lab of the class used Golang as the primary implementation language, we decided to stick to Golang and write our editor entirely in the same language. The benefit we gain is the ease of code sharing between the client and the server implementations. Also it was the language everyone on our team knew decently well after the semester of use for this class.

We implemented a simple text editor basing our implementation on simple pattern of redrawing the screen after each user or network event. We use Golang bindings for working with terminal modes and capturing user input that uses curses under the hood.

4. OPERATIONAL TRANSFORMS

Consider the situation where Alice and Bob are editing a shared document with the text, "1234". Alice sends the operation "Insert A at index 1" to the server and applies it locally, while Bob does the same with the operation "Insert B at index 3." The server then sends Alice's operation to Bob and vice versa. If Alice and Bob just blindly apply these operations, Alice will get the string "1A2B34" while Bob will get the string "1A23B4."

The solution for this is Operational Transforms, which enforces eventual consistency. When the client or server receives a message from the server that could potentially cause a conflict, we would modify it first. This is done through a function named Xform, which is central to operational transforms. Xform takes as input two operations c and s, and returns two operations c' and s' such that performing c followed by s' results in the same state as performing s followed by c'.

To make Xform concrete, consider figure 1, a typical OT diagram. Each point represents a state of the document

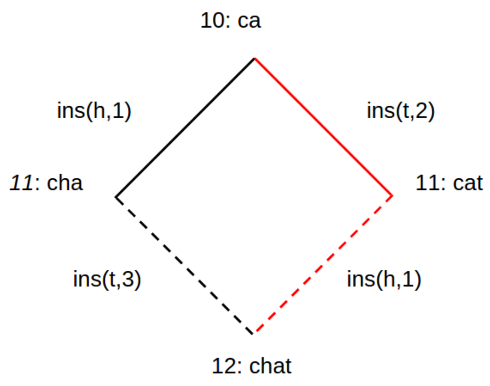

Figure 1: A simple Operational Transformation Example

as seen by the client or server. The black lines represent the local operations of a client, and the red lines represent the operations of the server. The diagram is meant to be read from top to bottom; lines slanting towards the bottom left are changes initiated from the client, and lines slanting towards the right are changes initiated from the server.

The client and server agree that at version 10 the document contains the text "ca." The client inserts 'h' at index 1 while the server inserts 't' at position 2. This is a potential conflict, so Xform(c, s) is called on $c =$ins(h, 1) and $s =$ins(t, 2), and the pair returned is $(c', s') =$(ins(h, 1), ins(t, 3)). Note that at version number 12, the states of the client and server are both the string "chat," but the operations that led them there are different.

Out implementation also includes version numbers and logs on both the server and client side. For the the server, the version number starts at 1. Each time the server receives a operation from a client, the client may potentially transforms the operation, writes the (potentially transformed) operation to log, and increments its version by 1. For the client, the version number is defined as the last server version number where the server and client are known to be consistent. Thus, consider the state 11:cha in figure 1; when the client is in that state, its version number is still 10. Furthermore, the client also logs all of its changes in its own log.

Other implementations such as [2] use two version numbers in its operational transforms. The first version number represents the client's version number, while the second number represents the other party (server)'s version number Instead, figure 1 would have versions $10, 10$ for "ca", versions $11, 10$ for "cha" on the client side (since the client is one operation ahead of the server), versions $10, 11$ for "cat" on the server side (since the server is one operation ahead of the client), and version $11, 11$ for "chat" when both the client and the server have converged again. However, we realized that having version numbers for each client and server added unnecessary complexity, as we defined our server to be the always-correct state, so our implementation only uses a single global version number.

4.1 Op.Op

op.Op is our data structure for all operations; see table 1 for all of its fields. We support insert character, delete

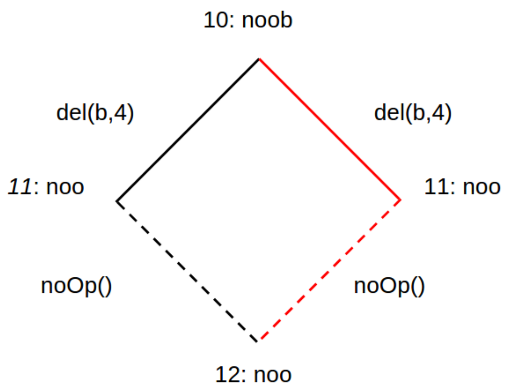

Figure 2: When two deletions have been recorded as operations, the transformation coud potentially be a NoOp. We do not want to end up at state *no* at version 12

Table 1: Operation Structure

Line	Type	Description
OpType	STRING	Type of Operation
Position	INT	Location of Operation
Version	INT	The client version
VersionS	INT	The server version
Uid	INT64	Unique ID of the client
Payload	STRING	Value associated with the operation
Path	STRING	Path of the operated file

character, and no-op. Resolving transformations for most pairs of operations is pretty straightforward; we list some of the more interesting ones here.

Multiple inserts are the same index: We need a way to determine which insert comes first and which one comes second. Since each operation has a Uid, we arbitrarily chose that the operation with the smaller Uid goes first.

Two deletes at the same index: This means the client and server are both trying to delete the same character. Thus, the Xform of 2 deletes at the same index is (noOp, noOp).

4.2 XForm: transformations

In the function `XForm` we resolve all possible conflicts between three types of supported operations: `ins`, `del` and `move`. Insert and delete commands operate on files with a certain path and edit at a certain position. Move operations just specify two paths, the operation is similar to UNIX's `rename`.

Inserts and deletes are resolved with correctly adjusting the position of the operation.

In case of moves, the path is adjusted. If one file had a path `/foo/bar` and there was an edit `(ins, /foo/bar, 3, text)` but in between server received a move `(move, /foo, /qux)`, the edit would be transformed accordingly with the path of the file containing the new name of the folder: `(ins, /qux/bar, 3, text)`.

5. CLIENT SIDE IMPLEMENTATION

Our goal in the design of the client is similar to the motivation of the Google Wave design. We want to make the client compute as much as the operations as possible, while lessening the workload on the server end.

The client communicates with the server through two RPC calls, `sendOp` and `pull`, both of which will be described below.

When we complete an operation, be it an insert or delete, on the CLI, we can apply the operation to the client's log directly. The client's log may diverge from the server's logs, but it should accurately reflect the sequence of operations that is seen from the client's perspective. Next, we add the operation to a buffer on the client. The buffer is essentially a first in first out queue, such that SendOp will look through the buffer and send the first operation in the buffer. An operation remains in the buffer until the sendOp RPC call returns that the client is up to date with the server. If the sendOp RPC call returns that the client is not up to date, then the client will have to process the outstanding operations that it will receive from the server, and only after all of the operations have been processed and applied can the originally operation sent by the client be cleared from the buffer.

We maintain the following invariants:

- Every operation that the client sends must be an operation that is consistent with the last version at which the client is in sync with the server.
- The version number of operations in the log differ by exactly 1 for consecutive operations.

For the first point, in order to ensure that the server's workload is reduced, we allow only operation to be sent to the server at a time. We can send an operation to the server through an sendOp RPC call over a TCP/IP connection. After sending the operation, if the server responds with an acknowledgment that the client is in agreement with the server, then the client can continue sending operations. However, if the server responds that the client is not up to date —that is, maybe a separate client has added operations to the server and our client would need to process those logs —then, the client cannot send operations to the server until it has pulled the server's outstanding logs, and computed the proper operational transforms against the client's buffer.

Figure 3 describes the scenario. Say at first that all clients and the server are at an agreed version 10. Let us imagine a scenario when the client 0 has completed two operations A and B on it's command line interface, while a separate client 1 has also completed two operations C and D, and sent them to the server. The server has processed and applied the operations and incremented to version 12, so when the server receives operation A from client 0 on version 0, it will complete operation transforms, apply operation A', and increment to version 13. Client 0 will then know that it is out of date, and receive operations A and B from the server. At this point, the client will compute A' and B' against operations C and D, which are stored in the client's buffer, and applies A' and B' to its log. Through the operation transforms of A' and B', we will also have computed D'. Only after applying A' and B' to the log, can we clear C from the buffer, and then send D' to the server.

However, figure 4 describes a more complicated scenario. In this situation, by the time we have received the response for OpC, the server has already applied Ops A, B, C, and E. Thus, when processing OpA and OpB, we transform everything in the buffer as we have done previously and apply

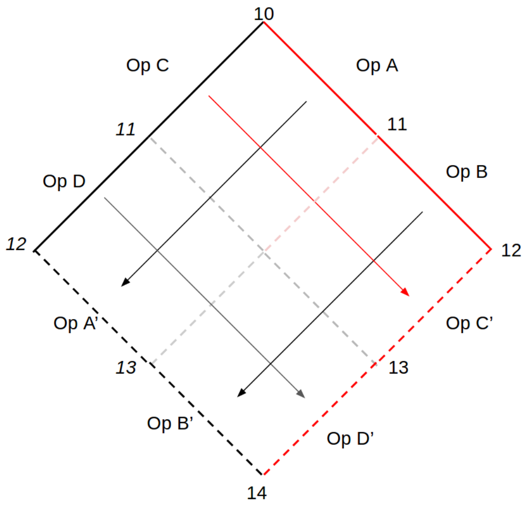

Figure 3: In this scenario, the client and the server have diverged by two operations. The light red is the intermediate transformation done by the server and grey is the intermediate operation done by the client. We try to make the client do as much of the operation transfoms as possible, such that the client will send *Op D'* to the server but it will record *Op D* in its own logs.

OpA' and OpB' (the transformed versions of OpA and B) locally. When we get to OpC', which will have the same Uid as the client. When the client receives a message with the same Uid as itself, it must be a transformed version of its own message. Thus, we pop the first element of the buffer such that the buffer consists of only OpD. For the remainder of the logs, we transform them as normal.

Due to RPC constraints in golang, only one way RPC calls can exist. That is, while the client can send RPC calls to the server, the server cannot send RPC calls to the client. We attempted to bypass the issue at first by using third party libraries such as RPC2 [5], but using these libraries introduced further issues.

Thus, we implemented a `pull` functionality on the client to act as a figurative "push" from the server's end, in case the server has outstanding logs for the client. In the client's "pull" function, it will periodically send an empty request to the server, containing its version number. If the server's version number is ahead than the client, it will send the client all the logs that it needs to catch up to the server and achieve a consistent state. This pull RPC call can also be accelerated in the case when a sendOp RPC call returns a out of date situation, which would happen if the client is attempting to send an operation but the server has previously received other operations from other clients.

6. SERVER SIDE IMPLEMENTATION

On the server side, we maintain a server version number as well as a version number for each client. When a message is received from a client, the server sends that client all of the logs from the server's version onwards and updates its own state. We define state of the string in the server to be the correct string, such that if clients diverge from the server's string representation, it is up to the client to correct

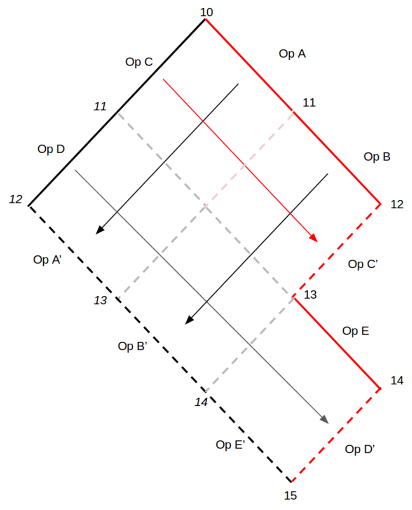

Figure 4: Similar to figure 3, except the client receives a log containing a transformed version of one of its own operations. When processing that log, the client pops one element of its buffer and then proceeds as normal.

itself.

Whenever a new client joins, we send that client a snapshot such that they are also up to date. Snapshots contain a the state of our strings and the version number of the server, but it does not include the logs of the server. For example, if at version 10 the server is displaying string "hello" and a new client joins, we can simply send "hello" and version 10 to the new client without having to include the previous logs.

We keep a lock on the state whenever we do a modification, so we do not get race conditions. If a server receives a operation from the client such that the version number of the operation is behind the current version number of the server, then the server will apply the appropriate operational transforms on the received operation. Note that the server does not need to keep a buffer like the client, since the server itself does not create any operations itself.

Similar to the client, the server also maintains a log record that represents its path of operations to reach its current state. For example in figure 3, the server's logs follow the red path and would contain Op A, Op B, the transformed Op C', and the transformed Op D'.

7. CONCLUSION

We have successfully implemented a collaborative text editor that includes basic text editing and snapshotting. We also implemented the functionality for moving files concurrent with the edits in a tree-like file system but didn't have enough time to build and debug the appropriate user interface for these operations. In terms of our future use, we are considering the following:

Batching messages that are sent to the server. Google Wave implements "compound operations," which is one of the main optimizations that allows it to scale to hundreds of people. This would greatly improve performance.

Essentially implementing snapshotting. When a client reaches a state that is known to be consistent with the server at some state, it can discard all its logs. The server can discard all logs before a version x if all clients have version number greater or equal to x. This would prevent the log from growing indefinitely.

8. REFERENCES

[1] Wang, Ma, Lassen, "Google Wave Operational Transform", July 2010.

[2] David A. Nichols, Pavel Curtis, Michael Dixon, and John Lamping. 1995. High-latency, low-bandwidth windowing in the Jupiter collaboration system. In Proceedings of the 8th annual ACM symposium on User interface and software technology (UIST '95). ACM, New York, NY, USA, 111-120. DOI=http://dx.doi.org/10.1145/215585.215706

[3] C. Ignat, and M.C. Norrie: "Customizable collaborative editor relying on treeOPT algorithm," Proc. of the European Conf. on Computer Supported Cooperative Work, pp. 315 - 334, Sep. 2003.

[4] A. Davis, C. Sun and J. Lu: "Generalizing operational transformation to the standard general markup language," Proc. of ACM Conf. on Computer Supported Cooperative Work, pp. 58 - 67. Nov.16 - 22, 2002.

[5] Cenk Alti. RPC2: Bi-directional RPC in Go. https://github.com/cenkalti/rpc2, Nov 2016. Accessed May 2017.

Appendix

Our code can be accessed through this Github repository: https://github.com/qandrew/6.824-fp

Acknowledgments

We would like to acknowledge Prof. M. Frans Kaashoek, Prof. Robert Morris, and the 6.824 Distributed Systems Teaching Assistants for their support in this project.