Public Record: Implementing a Real-Time Collaborative Editor Using Operational Transformation

Jonas Luebbers David Moon

1 Introduction

This paper presents our implementation of Public Record, a publicly accessible real-time collaborative document editor. Public Record maintains a single, persistent, unbounded document. Users may read and edit this document through a minimal web interface consisting of a single text box, which is accessible by any bored individual with the URL and nothing better to do. The purpose of this project was to learn about and implement a general technique for optimistic concurrency control called Operational Transformation (OT).

We begin in Section **??** with an overview of the particular requirements imposed by real-time collaboration and the key concepts underlying OT; we also review and compare against OT some recently developed, alternative solutions for optimistic concurrency control. We then describe in Section **??** our implementation of OT in Public Record; we also describe in this section a number of other key features, optimizations, and challenges encountered during development. In Section **??**, we evaluate the performance of Public Record. Finally, in Section **??**, we discuss various directions for future work.

2 Background

Real-time collaboration The complexity of real-time collaborative editors (RTCEs) stems from a fundamental tension between (1) the standard of developing a smooth user experience and (2) the reality of network latency. Users should be able to edit the same document simultaneously and at will; in particular, for the user experience to be sensible, each should see her own edits be applied immediately. With network latency, this rules out a simple lock-based approach in which users pass back and forth an "edit document" token while applying edits to a single central document. Immediate updates can only be applied locally, and thus an RTCE must manage multiple user-local copies that are being updated concurrently. As these copies represent the same document, an RTCE must also ensure that they eventually converge.

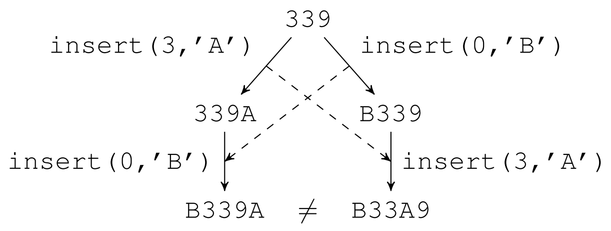

Figure 1: No convergence due to noncommutativity.

The raises a new problem: edits may be noncommutative. Suppose Alice and Bob are collaborating on a document that currently reads `339` on both of their local copies. Alice applies the operation `insert(3,'A')`, which updates her local copy to `339A`. Simultaneously, Bob applies the operation `insert(0,'B')`, updating his local copy to `B339`. Upon receiving each other changes, Alice and Bob cannot both simply apply them verbatim. If they did, Alice's local copy would follow the left branch in Figure **??** and arrive at `B339A`, while Bob's local copy would follow the right branch and arrive at `B33A9`. That is, their local copies would not converge.

$$\texttt{xform : } (p, q) \mapsto (p', q')$$

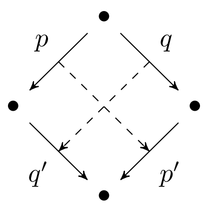

Figure 2: The `xform` function.

Operational Transformation One solution to this issue is Operational Transformation. OT is a general technique for optimistic concurrency control that is widely used today in real-world systems such as Apache Wave and Google Docs. OT incorporates domain-specific knowledge of the state space at hand in order to resolve the issue of non-commutativity such as that encountered by Alice and Bob above.

At the heart of OT is the *transform* function, denoted by `xform`, which takes operations performed in one context and transforms them to apply in another. The `xform` function takes a pair (p, q) of operations concurrently applied to some common `state` and produces a new pair (p', q') of operations such as that

$$\begin{aligned} &\texttt{state.apply}(p)\texttt{.apply}(q') \\ = \; &\texttt{state.apply}(q)\texttt{.apply}(p'). \end{aligned}$$

This is visually captured by the diamond-shaped schematic in Figure **??**. The `xform` function is symmetric in the sense that, if $\texttt{xform}(p, q) = (p', q')$ and $\texttt{xform}(q, p) = (q'', p'')$, then $p' = p''$ and $q' = q''$.

The use of the `xform` function in the case of document editing with Alice and Bob is displayed in Figure **??**. Upon receiving Bob's operation `insert(0,'B')`, Alice transforms it against her locally applied operation `insert(3,'A')` and applies the result `insert(0,'B')`; in this case, the transformed result is the same as the original operation because Alice's local operation occurred later within the document string and therefore has no effect on Bob's operation index. Symmetrically, upon receiving Alice's operation `insert(3,'A')`, Bob transforms it against his locally applied operation `insert(0,'B')` and applies the result `insert(4,'A')`; in this case, the transformed result takes into account the index change imposed by Bob's local operation.

$$\begin{aligned} \texttt{xform : } &(\texttt{insert(3,'A')}, \texttt{insert(0,'B')}) \\ \mapsto \; &(\texttt{insert(4,'A')}, \texttt{insert(0,'B')}) \end{aligned}$$

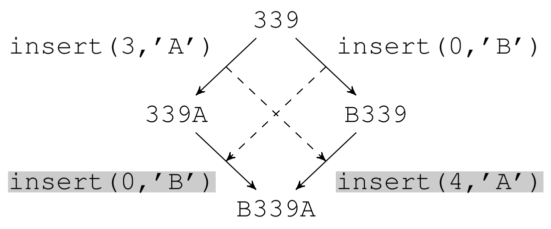

Figure 3: Convergence via OT.

$$\texttt{xform : } (p, q_1 q_2) \mapsto (p'', q_1' q_2')$$

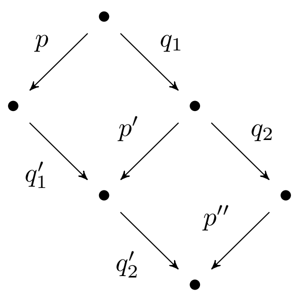

Figure 4: The `xform` function lifted to sequences.

Altogether, this ensures that both Alice and Bob's local copies converge to the state `B339A`. The `xform` function may also have been defined such that their copies converge to `B33A9` (cf. Figure **??**); which final state is chosen for convergence, however, is not important.

Finally, note that the `xform` function may be lifted to sequences of operations in a well-defined manner, as shown in Figure **??**. This allows

3 Architectural Overview

In this section, we describe our implementation of Public Record. We begin with a high-level overview of Public Record's architecture, then describe the implementation details of a few key features.

Client-server model Public Record uses a client-server model in which every user client interacts solely with a central server. Each client maintains a local copy of the server's master document. From the client's perspective, this is equivalent to a two-user peer-to-peer model such as that between Alice and Bob.

The server receives operations from clients, transforms them as necessary against any concurrent operations, applies them to the master document, and broadcasts the applied operations back to all clients. The operations have a field storing the original client author, so broadcasting also serves as an acknowledgment of receipt to the original author.

The client immediately applies any local operations and sends one at a time to the server, waiting until it receives acknowledgment of each operation before it sends the next. Upon receiving a broadcasted operation from the server, the client checks whether it is the original author. If it is, it sends its next unacknowledged operation; otherwise, it transforms the received operation against each of its unacknowledged operations and applies the result to its local copy.

Tracking concurrency The `xform` function solves one problem; knowing when to use it is another. Recall that `xform` applies to pairs of operations that are concurrently applied to a common document state. Both the client and server need to be able to distinguish such pairs.

For the client, this is simple. An operation p received from the server is concurrent with a local operation q if and only if p is received before q is acknowledged. Furthermore, p applies to the same document state as the oldest unacknowledged local operation. This means that, if the client has unacknowledged operations q_1, q_2, q_3, it may simply apply `xform` to (p, q_1) to get...

The server, on the other hand, must keep track of operations received from many clients. To do this, it assigns each

Operation composition Public Record essentially supports operations of two basic types: `insert(i,c)`, which inserts character `c` at index `i`, and `delete(i)`, which deletes the character at index `i`. However, actually representing operations at this level of granularity is not optimal. Recall that the client must wait until each local operation is acknowledged by the server before it sends another. Sending out operations on individual characters in this fashion may cause unacceptably slow rates of convergence between users in the presence of network latency.

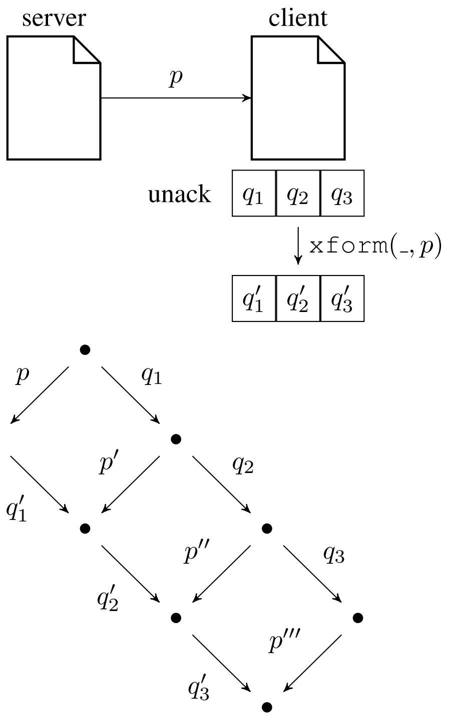

Figure 5: Client-side concurrency tracking.

To optimize against this, we borrow a trick from Google [**?**] and structure operations so that they support *composition*. With operation composition, a client may simply compose together all local operations that are applied between the sending and acknowledgement of an operation. For example, in Figure **??**, the client may compose together the operations q_2, q_3, q_4 into q while waiting for acknowledgment of q_1.

Specifically, operations are structured as lists of elements of three different component types:

- `retain(n)` for some integer `n`, which
- `insert(s)` for some string `s`
- `delete(n)` for

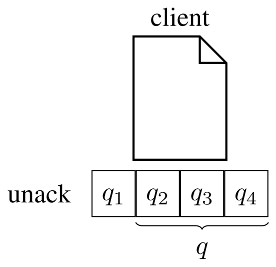

Figure 6: Optimization via operation composition.

Dealing with JavaScript

4 Evaluation

5 Future Work

References

[S] D. Spiewak. "Understanding and Applying Operational Transformation". 17 May 2010. [Blog entry]. Code Commit. Available `http://www.codecommit.com/blog/java/understanding-and-applying-\operational-transformation` [Accessed: 20 May 2016]