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 2 with an overview of the particular requirements imposed by real-time collaboration and the key concepts underlying OT. We then describe in Section 3 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 4, we evaluate the performance of Public Record. Finally, in Section 5, 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.

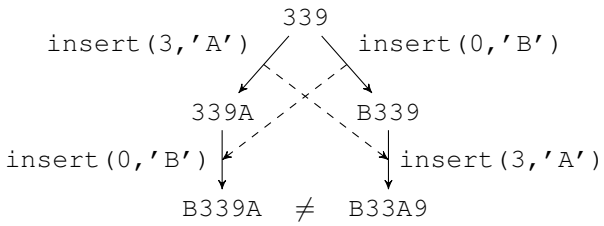

Figure 1: No convergence due to noncommutativity.

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 1 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.

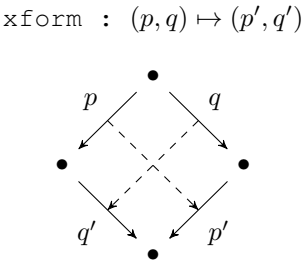

Figure 2: The `xform` function.

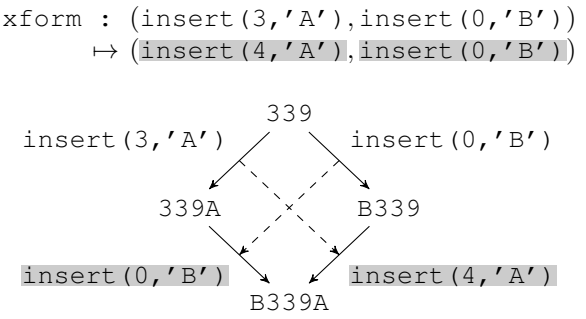

Figure 3: Convergence via OT.

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 2. 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 3. 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.

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 1); which final state is chosen for convergence, however, is not important.

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 upon the receipt of p, it may simply transform p against each of the q_i in succession. The transformed result p''' is applied to the document, while the results q_1', q_2', q_3' form the new buffer. See Figure 4 for a depicition of this procedure. This is equivalent to the behavior of Alice or Bob in the example in Section 2.

The server, on the other hand, must track concurrency between the operations received from many clients. The server may simply apply operations in the order it receives; however, it must determine against which already applied operations a newly received operation should be transformed. To do this, the server maintains a Lamport clock that corresponds to the number of operations it has applied to the master document. The server also maintains a history of all applied operations (see Section 5). It attaches the current clock value to each broadcasted operation; then, when the client sends an operation q to the server, it attaches the last received clock value from the server incremented by one. The timestamps on the operations received by the server indicate from what point in its history it must begin transforming q before applying it.

For example, consider the diagram in Figure 5, where the indices on the operations denote their timestamps. The server has accumulated a history

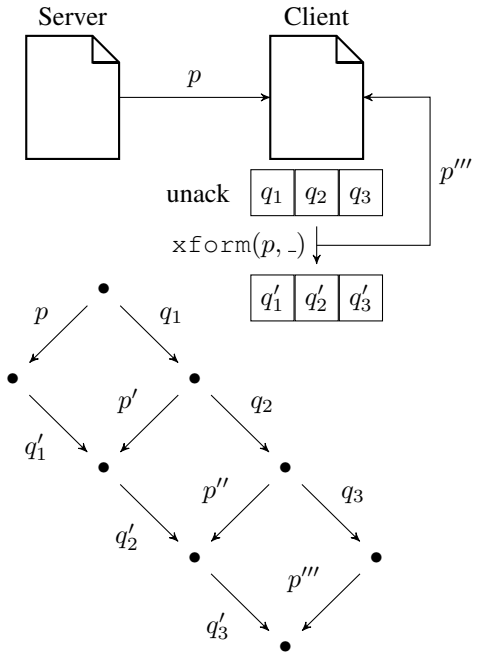

Figure 4: Client-side concurrency tracking.

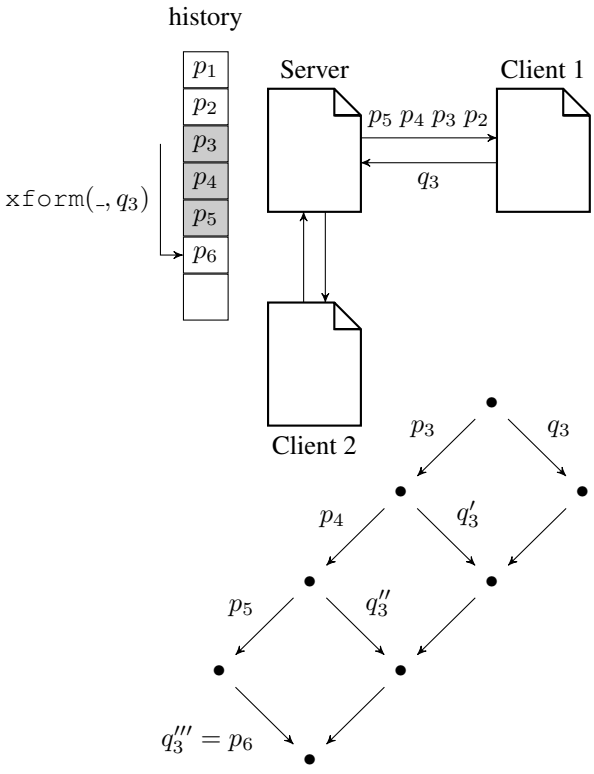

Figure 5: Server-side concurrency tracking. The indices on the operations denote timestamps.

of 6 operations p_1 through p_6. In the meantime, its connection with Client 1 has lagged because of network latency, such that Client 1 has just received operation p_2. Client 1 then sends a new operation q_3. The timestamp 3 indicates that q_3 applies to the same document state as p_3, and that all applied operations since then are concurrent with q_3. Thus, the server knows to transform q_3 against each of the operations p_3, p_4, p_5. The transformed result $q_3''' = p_6$ is then applied to the document and added to the history.

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.

To optimize against this, we borrow a trick from Google [S] and structure operations so that they support *composition*. With operation composition, a client may simply compose together any local operations that occur while it waits for another to be acknowledged. We elaborate on this optimization further below.

To support composition, operations are structured as *traversals* of the whole document. Specifically, an operation is an ordered list containing primitive traversal units of three types:

- `retain(n)` for some integer `n`, which advances the current position within the document by `n` characters;
- `insert(s)` for some string `s`, which inserts the string `s` at the current position, then advances the current position by the length of `s`;
- `delete(n)` for some integer `n`, which deletes the next `n` characters from the current position.

In this form, an insertion that changes the document from `abc` to `abxc` would be represented as `retain(2),insert(x),retain(1).`

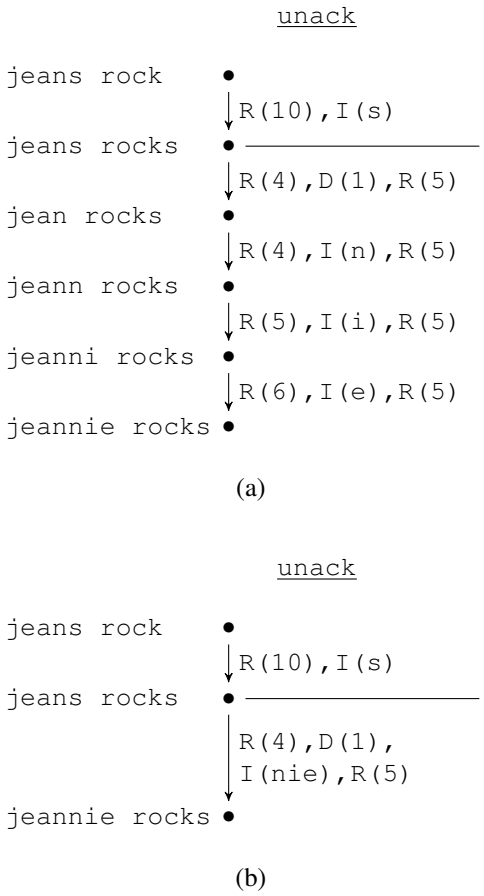

Figure 6: A client's buffer of unacknowledged operations (a) without composition and (b) with composition.

Now consider Figure 6, which depicts a client's buffer of unacknowledged operations. Figure 6a shows the buffer at the granularity of individual character insertions and deletions. While the client waits for acknowledgment of the first operation `ret(10),ins(s)`, however, it may compose the remaining operations in its buffer to a single operation as shown in Figure 6b. In so doing, the client minimizes the number of operations it must send to the server.

4 Evaluation

Evaluating the performance of a real-time collaborator like Public Record comes with a unique set of challenges. Most testing frameworks like `ab` stress test applications by sending many http re-

quests. However, when evaluating Public Record, we are not interested in the server's ability to server http requests. Rather, our goal is to test our implementation of the OT algorithm itself. This means that that our testing framework must simulate actual clients and client-server interactions.

Our plan to accomplish this was to run client bots deployed over many Amazon EC2 micro-instances. Each client bot would connect to our server as if it were a real user and run various test procedures. However, this was not so straightforward.

One particular challenge lay in designing the client bots so that they could run independently of a browser window. Because our client is implemented in JavaScript and originally relied heavily upon the input received through browser-specific callback functions (e.g. `oninput`, `onkeydown`, etc.), we refactored the client so that it can also construct operations programmatically. This way, the client could be run both within a browser as well as in the shell environment of an Amazon EC2 instance.

Another challenge lay in simply distributing and running our client code on the EC2 instances. In order to test Public Record's ability to handle many simultaneous users, we needed to be able to start the client code on all of our EC2 instances at once. Manually doing this one by one via SSH was evidently not an option. Originally, we tried to use a script called Bees with Machine Guns [Be]; however, this turned out to be out-of-date with Amazon's latest updates to its EC2 instances. We then discovered that there is a new option in the AWS console for sending commands to many EC2 instances at once; this required deleting and recreating our instances under a new AMI with the correct permissions, along with installing the Amazon Run Command agent on every instance.

Our plan was to measure the average server response time as we varied the number of requests made simultaneously. We wrote two client bots—one which sends operations as quickly as possible to the server, and another which sends operations at the rate of an average typist. While the former test may seem like a more effective way to stress our server, the reality is that producing operations at such quick rate means many operations are composed together before they are sent. For this reason, the latter test gives us more realistic results.

Unfortunately, after all this work, we were unable to perform any evaluations. The optimization in operation structure that allows for composition makes it difficult for the client bots to construct operations programmatically. Using this structure, each operation has a source and target length and can only be applied to strings of the source length; similarly, two operations can only be composed if the target length of one matches the source length of the other. This meant that the client bots could simply send a blind stream of operations, but rather had to keep track of the server operation history and only send operations that could applied to the server's current state. This is difficult to control when the client bots are sending operations at a fixed rate, regardless of server acknowledgment. We have some ideas about how to address this issue; however, we realized that this was the source of error very late in the process, and we decided that it was time to let things go.

5 Future Work

There are many obvious directions for future work on Public Record. One glaring implementation flaw is our lack of maintenance of the server's queue that stores the history of all applied operations. This could easily be managed by keeping track of last-received operation timestamps for each client, and removing from the queue any operations with timestamps smaller than any client. Without this fix, Public Record is certain to run out of memory if it runs for long enough. A related improvement would be to replicate the master document to ensure that it truly persists.

Another source of inefficiency is parsing user input. Because we implemented Public Record using JavaScript, our parsing is restricted by what is available through browser callback functions. Because these functions do not provide the exact location of user input within the textbox interface, we are forced to take a diff of the original state and the new state upon every user input. Although we do not need to run a full diff algorithm due to the fact that a user can only apply changes at her cursor at every point, parsing input still grows linearly with the length of the document. To get around this issue, we would have to hack into the depths of JavaScript.

References

[B] T. Baumann. "What is Operational Transformation". Available https://operational-transformation.github.io/what-is-ot.html [Accessed: 20 May 2016]

[Be] Bees with Machine Guns. Available https://github.com/newsapps/beeswithmachineguns [Accessed: 20 May 2016]

[EG] C.A. Ellis & S.J. Gibbs. "Concurrency Control in Groupware Systems". SIGMOD (1989).

[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]