Peer Editing with Splice

Somak Das
Joshua Ma

1. Problem

Real-time collaborative (live) *editing* allows users to simultaneously edit a shared document. It has been popularized by client–server services such as Google Docs and Etherpad. Since these services are web-based, they are accessible to and regularly used by tens of millions of users.

One issue that arises is that users don't want to wait for the client to synchronize with the server before their edits appear in their document view. So, the live editor must permit concurrent updates, yet prevent divergent document states. To solve this, *operational transformations* are at the heart of most live editors (providing optimistic concurrency control). Every edit is represented as an operation and sent to the centralized server, which then uses the OT transform mechanism to automatically resolve conflicting operations, while preserving the intent of each.

But another issue is scaling the centralized server. The server must handle millions of edits per second from clients, incorporate them into the document states, and broadcast them out to other clients.

2. Solution

We designed ***Splice***, a ***s***uperb ***p***eer-to-peer ***li***ve ***c***ollaborative ***e***ditor. Splice is a web-based live editor just like Google Docs and Etherpad. But, we ditched the client–server model in favor of peer-to-peer. This distributed solution scales well because a document's operations are only passed between the users accessing that document. Furthermore, it naturally builds in privacy since no centralized server receives the document.

Anatomy of a Peer

Each peer is a member of a group accessing the same document. Since Splice is web-based, the peer is a program running in a user's browser tab. Overall, it is responsible for synchronizing with other peers and managing the user's view of the document. We organize the peer into three components:

#1: The transport layer handles peer connections. It also tracks the active set of reachable peers. Therefore, the transport layer can send or broadcast data to other peers, can notify listeners (i.e. the server) about new data, and can notify listeners about peers joining or leaving. We need this component because browser-to-browser communication is still experimental and requires explicit management.

```
Document:  rtm
Log:  <<Charles: 12:56>, <insert art>>,
      <<Neha: 12:57>, <retain 3, insert m>>,
      <<Neha: 12:59>, <delete 1, retain 3>>
Version Vector:  <Charles: 12:56, Neha: 12:59>
```

Figure 1: Example document state at a peer's server.

#2: The "server" handles the document state. We want to ensure that document states do not diverge, even though each user is making (local) edits to her view of the document and peers communicate with each other in an uncoordinated manner.

Inspired by Bayou, Splice guarantees *eventual consistency*. The server keeps a log of writes, and each local edit is recorded as <version, operation>, where the version is <author's peer ID, timestamp>. A *logical clock* generates the timestamp, taking the user time plus an offset to handle clock skew between peers. Versions enable a universal ordering of writes to the document. The server also keeps a *version vector* that summarizes the contents of its log as a map from peer ID to maximum timestamp seen from that peer (in other words, a set of versions).

At any time, the server on a peer S can transmit a list of writes from its log to R. Then R integrates unseen writes into its own log with the correct order, calling our `merge(myLog, otherLog)` algorithm to do so.

To limit the growth of its state, the server can prune the earliest write w from its log if w has been committed. First, it computes the minimum version vector V_{min} among the peers. Then, it verifies that no peer can generate a write that would precede w, i.e. w.version $\leq v$ for every version v in V_{min}. Splice doesn't need a Bayou primary because it requires all the peers to be live.

#3: The "client" handles the user's view of the document and edits. It listens to the server for changes to the log and updates the user view accordingly. It also detects when the user has edited the document, constructs an operation to describe the edit, and submits the operation to the server.

3. Implementation

We implemented our solution with 1,500 lines of JavaScript, not including libraries. The demo is publicly available at `http://splice.joshma.com/`. Since Splice uses experimental inter-browser communication, it works in Google Chrome and Firefox Nightly as of this writing.

For the transport layer, we included the PeerJS library to transfer bytes of data between peers over the recently drafted WebRTC protocol. It provides reliable, error-checked, but unordered delivery. For the client, we included the CodeMirror editor GUI and subscribe to its `onChange` events. For the operational transformations used in both server and client, our own implementation combines features from `ot.js` and `changesets.js` libraries.

To avoid making the user enter socket addresses when joining, we made a simple BitTorrent-like tracker that has a list of all peers that ever accessed the document. The peer `GET`s this list and establishes connections to the currently reachable peers. Finally, it asks another peer for the document state and, upon receiving it, initializes its server and client.

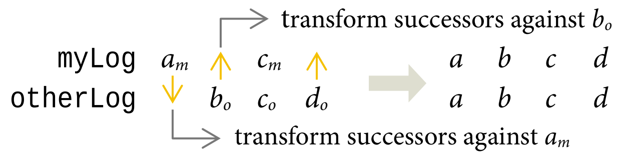
So far we've focused on our main feature: live editing. To provide a better user experience, Splice supports user names (vs. just peer IDs), live chat, real-time cursor presence and highlighting, and undo/redo (which works even in the midst of concurrent updates) too.

4. Technical Challenges

We implemented a distributed system in JavaScript, dealing with the language's asynchronous nature and lack of RPCs. At least its single-threaded event handling allowed us to manage document state without worrying about race conditions.

Our hardest challenge was writing `merge(myLog, otherLog)`. Let's say a, b, c, and d are operations on a document, where $a{\cdot}c{\cdot}d$ happened sequentially, and b happened simultaneously as $c{\cdot}d$. Then $a{\cdot}b{\cdot}c{\cdot}d$ does not make sense because $c{\cdot}d$ is incompatible with b. A centralized server keeps a single history, so it applies the operations as they arrive. In this case, it *transforms* b against $c{\cdot}d$ (i.e. computes a new b' that *can* be applied after $c{\cdot}d$) and returns $a{\cdot}c{\cdot}d{\cdot}b'$. But in Splice, a server often receives writes in different orders from different peers. A naive implementation that processes operations as they arrive would easily result in divergent logs and documents.

The algorithm in Splice merges logs together, with operations ordered by versions. Now, let's say the correct order is in fact $a{\cdot}b{\cdot}c{\cdot}d$, and a server on one peer has `myLog` $a_m{\cdot}c_m$ but receives `otherLog` $b_o{\cdot}c_o{\cdot}d_o$. The subscripts differentiate c_m and c_o, which are the same edit but, due to OT, are represented differently (one comes after a_m, the other after b_o). We want the algorithm to return $a{\cdot}b{\cdot}c{\cdot}d$, appropriately transforming operations so that they can be applied one after another. At a high level, our solution lines up and combines the two logs like so:


transform successors against b_o
myLog a_m c_m
otherLog b_o c_o d_o
a b c d
a b c d
transform successors against a_m

The vertical arrows represent adding an operation to a particular location in the log, and since logs are `Arrays` in JavaScript, this function is called `splice`. This is how Splice gets its name. Upon completion, the server has $a{\cdot}b{\cdot}c{\cdot}d$, but the client still only has $a_m{\cdot}c_m$. The server is only allowed to emit new operations, so it rolls back c_m by emitting $c_m^{-1}{\cdot}b{\cdot}c{\cdot}d$. Now, the client has $a_m{\cdot}c_m{\cdot}c_m^{-1}{\cdot}b{\cdot}c{\cdot}d = a{\cdot}b{\cdot}c{\cdot}d$, as desired.

After verifying basic correctness, we wanted to optimize how responsive Splice is. We observed that the performance bottleneck was network usage, not local procedures; for example, a 10kB log took 3s to reliably transfer over WebRTC but only 10ms to merge. Therefore, to reduce traffic, the sender only transmits writes that the receiver hasn't heard yet, based on the receiver's advertised version vector. The merge algorithm infers the unsent writes from the sender's version vector. We were also able to reduce the size of each write. Since Splice needs an ordering mechanism (not conflict detection), it tags each write with a version, which is cheaper than a full version vector. As a result, we find that nearly all writes propagate to peers in ≤0.5s.