Pad

https://github.com/jdhenke/pad

MIT 6.824 Spring 2014

Joe Henke, Marcel Polanco, Evan Wang

The Problem We Solved

For our final project, we created Pad, a web based, plain text, collaborative document editing system. Think plain text google docs. The focus of our system was to gracefully handle concurrent, conflicting updates to a document in real time, guaranteeing eventual consistency. We also achieved some fault tolerance, in that multiple servers serve our website and communicate via paxos to remain consistent, so if a server crashes, a client can connect to any of the other servers. Our servers persist state to disk and can guarantee after all servers are shut down and started up that they have consistent state. Lastly, we created unit, integration and latency tests.

System Overview

Pad is comprised of a collection of servers, each serving a website which provides a unique *document* at each URL prefixed with `/docs/`. Pad's documents are just plain text. If two users concurrently edit the same document, what is displayed on each of their screens may temporarily diverge; this is unavoidable. To remedy this, Pad propagates the edits of each user to all other users currently viewing the same document. Pad guarantees eventual consistency; if all users stop typing, each user's interface will eventually show the same text.

Conflict Resolution

Our Git-Like Approach

To be more specific, Pad models a client's edits as commits, as in git. When a client makes an edit, a commit is created and sent to a server. The server then communicates this commit to all other servers through paxos, so commits are always handled in the same order by all servers. Once received through paxos at each server, this commit must be rebased to the head of the document (explained next) then propagated to all listening clients. Listening clients must then update their local state to reflect this commit (also explained next).

Rebasing

As just described, rebasing is the core operation of our conflict resolution scheme. The basic premise is that two clients have made concurrent edits to a document. In our scheme these are two commits, C1, C2 with the same parent, CP. Both commits are sent to Pad's backend and are serialized in Paxos, so WLOG one is handled first, say C1. For each server, handling C1 is not problematic as its parent, CP, is the current head of the document. The problem is handling C2, whose parent is no longer the head of the document.

First, we explain why this is problematic. A commit essentially contains instructions for how to modify the state of the document. Imagine C2's instructions were to insert "a" at index 3. What if C1 contained instructions to delete the entire document? It is no longer a valid operation to blindly apply C2 after C1. C2 must be modified to be sensibly applied to whatever state results after applying C1. This is rebasing.

We define rebasing just as described: given two commits, C1, C2 with the same parent CP, we define C2' := rebase(C1, C2) st. we can apply C1 then C2' to advance the document to a new state which, as best as we can, includes updates from C1 and C2. Note that it is not always possible to preserve every operation in C2. For example, if C2 inserted a word into a paragraph and C1 deleted that paragraph, what should be left? It's up to interpretation. We decided that the result should be same if C1 or C2 was received first, so in this sense the paragraph is simply removed along with the inserted word within it. The details of rebasing are tedious, however we encourage the reader to look at our unit testing to see examples of its expected behavior.

To bring the discussion back to Pad as a system, when a pad server receives a new commit from paxos, it is rebased to the head of the document with the following pseudocode:

```
function receiveCommitFromPaxos(c) {
  // rebase over all commits which have happened since c's parent
  for (var i = c.parent + 1; i < commits.length; i += 1) {
    c = rebase(commits[i], c);
  }
  // reassign its parent now that it has been rebased
  c.parent = commits.length;
  // add it to the history of commits
  commits.push(c);
  // propagate c to all waiting clients...
}
```

In summary, we have used rebasing with automatic conflict resolution to serialize commits so that they may be applied one after the other to correctly advance the state of the document.

Applying Updates on the Client

Even when rebasing on the server resolves conflicting commits there are still two subtle points which must be addressed about applying updates on each client.

The first is that when a client receives a commit which originated from itself, it ignores it. Its local state already reflects the changes contained within that commit.

The second is that, as previously shown, blindly applying commits can be bad. What if a user has made changes since applying the last commit? Does it still make sense to apply the current one? The answer is no. Again, imagine if the user has just deleted the entire document since applying the last commit and now the new commit is supposed to insert "a" at index 3. We again run into the same problem, but that's fine because we can again apply the same solution: rebasing, only on the **client** this time.

Whenever a client receives a new commit from the server, it creates a temporary commit which stores the local changes since the last received commit. Then it undoes the local changes from the interface so the new commit can be applied without issue. Finally, it rebases its temporary commit over the new commit and applies it to the local state. In this way, it correctly applies new updates while preserving local changes.

Additionally, note that this client side rebasing of local edits is critical in maintaining consistent state between clients. If a client ignores its own commit but that commit was rebased on the server, that clients local state must also be â€œrebasedâ€ in the same way as that commit was rebased on the server. We can see that we've achieved this.

Implementation

Server(s)

Pad's backend is made up of a group of Go servers. Each server serves the files for the website and communicates with clients' web browsers. This group of servers also acts as a paxos group used to serialize the order in which commits are processed. Each Go server has a companion node server which is used to effectively perform RPC calls to run any git operations. This allows the same code to be used by the client and server for git functionality.

Each document is written to disk containing the current text of the document, the history of commits, and the time the document was last written. Our system persists all documents to disk every five seconds and each document is written concurrently so as to not inhibit the write time of the other documents.

In our system, persistence guarantees that all servers will initialize with the same state after an entire group shut down. Our system does not guarantee consistency in the instance that a single server shuts down and boots back up; we just let that server fail. We considered instead making Paxos persistent, but ultimately did not do this as it is more complicated and would interfere with the critical path of operations.

In order to startup and shutdown servers with ease, we created a script which when given a configuration can start all necessary processes for Pad. It works both locally and remotely on AWS. On startup, the pad servers undergo a **sync** operation, in which the servers agree on the persistent states of existing documents, agreeing to use the states with the latest timestamps should they differ.

Client

On the client side, we created a Javascript Client which encapsulates all of aforementioned conflict resolution and communication with the server, making it extremely easy to hook up to a text area. We used Web Workers to perform heavy computations off the UI thread and long polling to push updates from the server to the client.

Evaluation

Pad provides an end-to-end integration test suite, unit tests for the git-like system we implemented for conflict resolution, and latency tests

Unit Testing

We provide tests for the three core functions that comprise our git functionality -- getDiff, applyDiff, and rebase. We define `getDiff(a, b)` to be the set of operations that should be applied to `a` to transform it to `b`, which we refer to as a **diff**. Running `applyDiff(a, getDiff(a, b))` should return `b`. Therefore, we test getDiff and applyDiff in conjunction. Our tests consist of both basic test cases and randomized inputs of longer lengths.

Rebase is more difficult to test. The result of `rebase(diff1, diff2)` is `diff2prime`, which as described in

conflict resolution is `diff2` modified to account for `diff1`. More succinctly, the result of `applyDiff(applyDiff(c, diff1), diff2prime)` is what we expect `c` to look like if `diff1` and `diff2` were somehow combined and applied to `c`. This mentality is what we used to write our test suite. Rebase should behave differently depending on whether an action in a diff is an insert or a delete, and also depending on the relative indices at which these actions start and end. Therefore, we enumerated all the different combinations of actions and relative indices and created a test for each, ensuring rebase operates correctly on all edge cases.

Integration Testing

Pad uses PhantomJS to simulate client web browsers interacting with our system. Due to the overhead of using PhantomJS, we couldn't use this for latency testing, however we were able to test for correctness of the Pad Javascript Client under single, concurrent but not conflicting and conflicting writer workloads. It was necessary to use PhantomJS because the Pad Javascript Client uses Web Workers, which are only supported by web browsers and PhantomJS.

Latency testing

In order to test the end-to-end latency of the system, we designed a test which sends fake commits to each server in a configuration, recording when they were sent. Then, it listens for them to be propagated through each server, recording the latency for that commit from each server. We can then measure the average latency. In practice, we found the average commit propagation latency to be 100ms locally and 500ms on AWS, which we contend is fairly reasonable.