ostracon

Team name: ostracon

- Daniel Kim
- Louis Rassaby
- Scott Jacobson
- Jeremy Max Goldman

Addenda

Though we initially intended to write our library write in Python, we decided that in the interests of learning concurrency, we would switch to Erlang. There are several advantages to Erlang, most notably the lightweight nature of its threads and communication between them versus Python's much heavier threads.

Summary

We aim to create a Erlang server and corresponding front-end framework that implements distributed voting in real time for an infinitely scalable number of clients. Example use cases would include games (what we intend to implement as our deliverable), distributed music composition, and crowd-sourced intelligence.

We were inspired by the recent viral craze, "Twitch Plays Pokemon", where millions of users controlled (and eventually won) a complex game through voting via a text-based interface.

We want to take that idea one step further by creating an Erlang server that can be used for distributed voting through collecting keystrokes and sending them realtime to a server that compiles them and selects the winning keystroke from any number of keystrokes received over a given cycle of time.

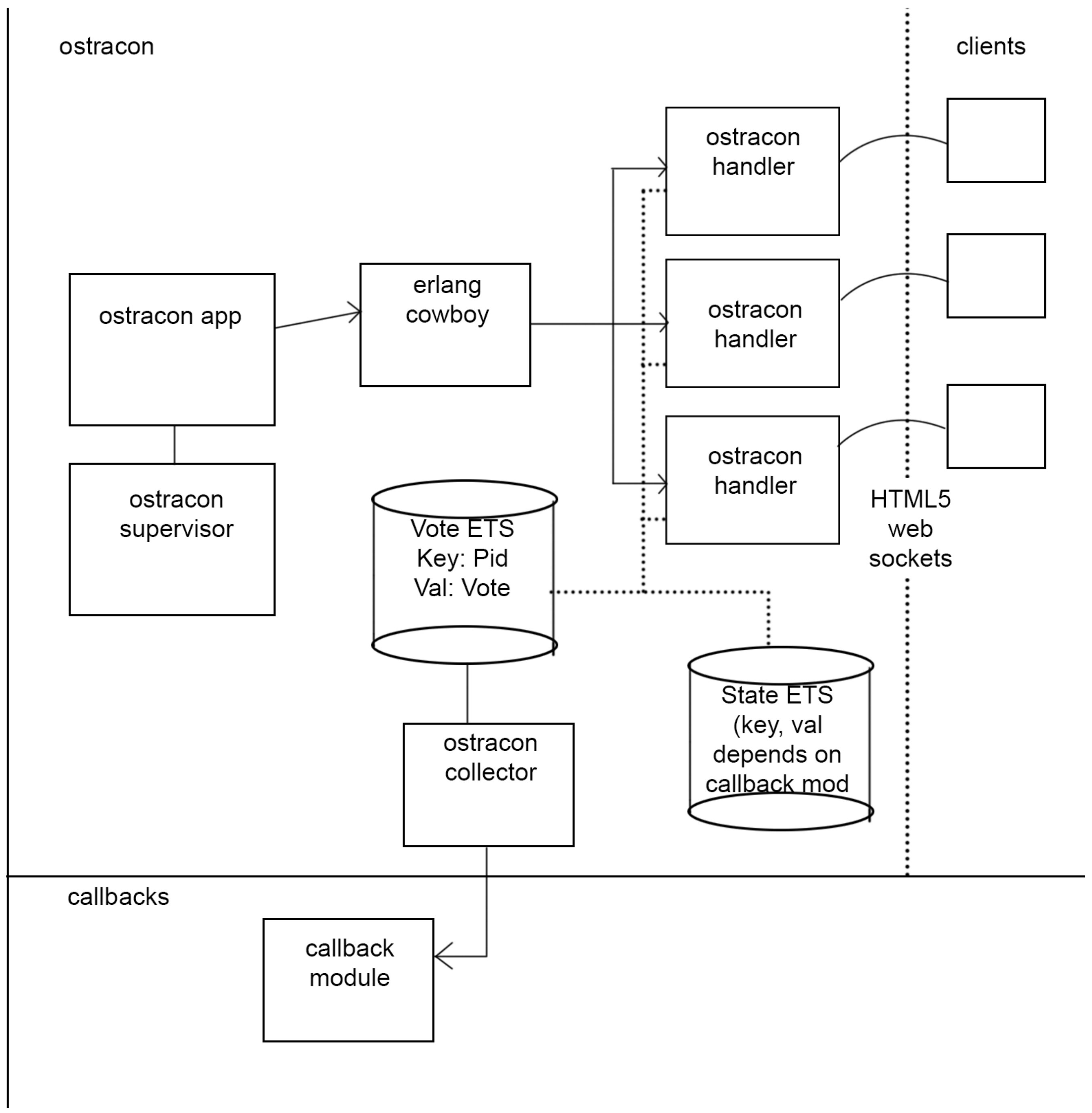
The most basic use of our library would be a maze game with two or more "players" attempting to reach the end of the maze before the other players. Each "player" can be a team of between 1 and n users. The advantage of having multiple players vote is that many minds looking at the same maze may be able to solve the maze faster if they work together.

Another use case is social psycology studies that aim to learn about teamwork behavior between people who don't know each other across the internet.

Deliverables

We have begun to break our overall ostracon server into several components, as described below, user-configurable when specified. The diagram below expresses the updated design, which features a more straightforward socket, voting, and collection process.

Ostracon Server



Ostracon App

This module starts the whole process and activates Erlang Cowboy. Cowboy will spawn the Ostracon Handler processes which each set up an HTML5 WebSocket with a client--one handler per client. The Ostracon Supervisor detects any failure in the app during the entire voting process.

Vote Collector + Handlers

1. The vote collector is the main 'loop' of the program. Its helper recieves requests from the client to

connect via SockJS, and the collector spawns off a socket, AKA ostracon handler process to handle that request and maintain a connection with that particular client.

2. Each ostracon handler process exists for the lifetime of the connection with the client - it begins when the client connects, and it is responsible for fielding votes from the client. Each socket has an Erlang process which we call an ostracon handler (see new diagram). Each one of these processes takes the votes given by the client and writes to the Vote "database" (an Erlang ETS). Each handler also reads from the state ETS to update to each client the state of the voting (and in most applications, the state of a game). Thus, any new handler that is introduced when a new socket is opened up (new client joins) will be able to read from the State ETS to be caught up with the state of the voting. Using an Erlang ETS solves the previous problem of how to get a consistent state that can be shared among all clients.*

3. At the end of a round of voting, the collector reads all the votes from the Vote ETS and computes a histogram of all the votes for that cycle. The histogram is in descending order of most popular vote. Once the histogram is computed, it is sent to the callback module and the Vote ETS is wiped clean for the next cycle.

*In more detail: 1. One problem we wanted to address was how to make sure that all clients using an application of ostracon could see the same consistent state at all times. 2. One way we considered addressing this problem was to set up a SQL database with our server and read from and write to that database. We also wanted to see if we could address this problem by making the state exist only on the front end, since we were having difficulty maintaining a shared state in Erlang. 3. We didn't like the idea of having a SQL database since it unnecessarily added more complexity that we didn't need and may not have been fast enough. We wanted to share the state in Erlang and make JavaScript deal mostly with just the client- side. There was no feasible way to make the state only exist on the front-end; the consistent problem that showed up was maintain consistency among clients. Choosing the Erlang ETS seemed to solve our problems. It fulfills our database needs without making things needlessly more complicated. By keeping the votes on one ETS and the state on another, we have a point of consistency and also have databases that integrate very well with our Erlang design.

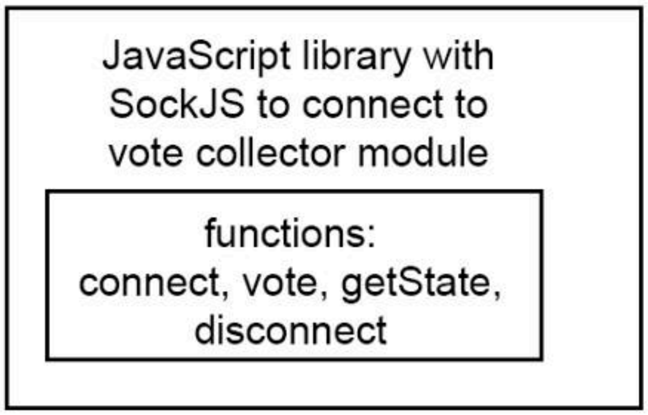
Callback Module

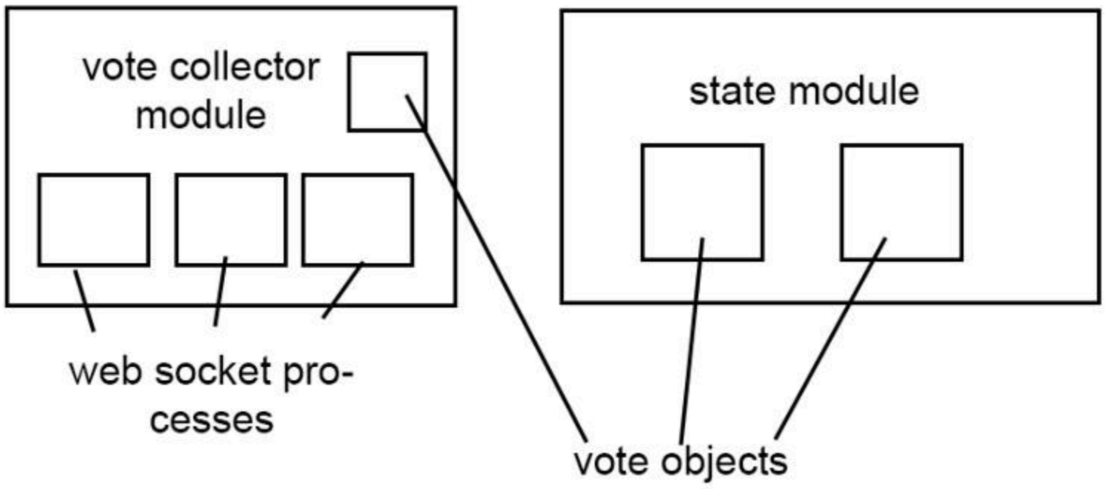
This is a configurable module that is responsible for turning the results of a particular round of voting and its associated histogram into a state that can then be communicated back to the client. This layer of abstraction allows for the voting mechanism to be integrated into full products we have not yet thought of, and attempts to avoid any assumptions about how the votes will be used. It fields messages from the Vote Collector and determines what the ostracon handler will tell the clients. The winning vote by default is simply the first entry in the histogram sent by the collector, since the histogram is computed to be in descending order of popular votes. It is not necessary for the callback module to directly deal with issues of concurrency--the ostracon package is designed to abstract that part away. The callback module specifies the length of each clock cycle for each round of voting.

Packages

Client-To-Server-Side WebSocket Connection:

On the server side, we are running a Cowboy server that handles requests from the client for direct WebSocket connections and hands those connections off to a function in our `ostracon_handler` module called `websocket_handle`. This is a deviation from our original plan, which was going to use SockJS as an abstraction above WebSockets for potential compatibility and performance benefits. It turned out that WebSockets are more widely supported than initially anticipated, and it would have added complexity and bloat to use SockJS. Instead, we are using the native WebSockets built both into pure browser-side JavaScript and the Cowboy package.


JavaScript library with SockJS to connect to vote collector module
functions: connect, vote, getState, disconnect



We have implemented the WebSocket connection on the client side as a pure JS file embedded in our index.html for testing purposes. We still plan on building a thin client-side framework, ostracon-client, over

WebSocket to abstract away the details of communication between our client and server. This will have a clean interface that consists of four functions: `connect`, `vote`, `requestState`, and `disconnect`.

Development Plan

With the design in place, we are ready to divi up the work and see how the pieces will fit together. We will, as a group, set up the basic Cowboy server and sockets so that we can observe how the pieces fit together and make sure the entire team has a working understanding of this piece of communication. We'll then standardize the object that will be passed over this WebSocket connection.

We will also define, write, implement, and test the API for the ostracon-client package we are building.

On the server side, we will determine the contract between the Callback module and the State Module, and build an example state module to match the initial game.

Timeline (tentative)

Nov 15 -- Set up Cowboy Server and sockets and JS front-end (done) Nov 20 -- Integrate vote collector, start callback module (maze game) Nov 21 -- Refine and test vote collector, continue work on game Nov 23 -- Test front-end, test consistency of state Nov 24 -- Put it all together, see if random people in Halligan can play maze game

Breakdown:

Initial JS Game (with hooks for initializing state and taking in final votes) - ***Dan***

Setting up Cowboy server with Erlang-SockJS configured and integration and testing of our ostracon-erlang modules with `ostracon handler` on top of it - ***Jeremy***

Designing and testing `ostracon-client` and `StateModule`, and related APIs - ***Scott***

Designing and testing `ostracon-erlang` modules of `VoteCollector` and `Surveyor` - ***Louis***

Foreseeable Problems

Timing distributed systems is an incredibly complex problem. Delay on networks could cause votes to arrive outside of their expected time block. Depending on the application, this may or may not be an important concern.

Also, we have to decide the mechanism by which an application can collect data when our program is ready to give out that data in real time.

Overall, the timing of this concurrent system will make it a complex problem.

Our choice to use an Erlang ETS for the votes and for the state is one way we address this problem. Reading and writing with an ETS should prove to be faster and more convenient than using something like a SQL database.

Another foreseeable problem is a bottleneck at the computation of the vote histogram. We hope to address this problem by having the collector module spawn multiple processes to expedite the process of computing each vote histogram.

Our repo is at this link: https://github.com/lrassaby/ostracon