Ostracon: Realtime Voting over WebSockets

Louis Rassaby
Tufts University

Scott Jacobson
Tufts University

Daniel Kim
Tufts University

Jeremy Goldman
Tufts University

Abstract

As the number of distributed applications continues to grow, having libraries to facilitate communication and state updates eliminates boilerplate code and allows developers to focus on challenges specific to their applications. This paper presents Ostracon, a library built in Erlang and JavaScript for realtime collection of votes and sharing of state over WebSockets.

1 Introduction

Ostracon was originally inspired by the February 2014 viral craze and social experiment, "Twitch Plays Pokemon," wherein millions of clients control a single character in a Pokemon game by giving input via Twitch's chat service. We initially wanted to build a system for faster game voting, i.e. through keystrokes rather than text. While we found the problem of a distributed game interesting, we saw that on a more general level, real-time voting accross computers was a powerful concept that could span several domains. Vote collection is useful in a variety of applications, including surveys, games, and collaborative editing. Based on the votes, a consistent idea of state can be achieved server-side, then each client can receive an identical state update.

Thus, the Ostracon was born. As we began to see more applications for this real-time system, several requirements became clear to us in order that our library reach its full potential. Perhaps most importantly, we didn't want to tether users of the system to any particular language, or even local network. The obvious choice, then, was to host Ostracon as a server that could be accessed over the Web.

1.1 Goals

We then set out to decide the exact technologies and methodologies we would use to implement Ostracon, keeping the following goals in mind:

- **Modularity:** We envisioned Ostracon becoming a solution for any domain where real-time voting is a necessity. The number of users and the data being collected for each user, therefore, must be completely agnostic to the application. Application-specific logic will be placed in modules that don't care about communication or agglomeration of votes.

- **Abstraction:** We wanted to abstract away the maximum amount of boilerplate from the developer building an Ostracon application. On the back end, the end-user doesn't have to worry about Cowboy, WebSocket, or beginning and maintaining connections - they will be provided populated *Vote ETS* on each round of voting, and Ostracon will take care of pushing their state to the clients. On the front end, we likewise didn't want an end-user to have to worry about the specifics of WebSockets or formatting votes in a way that Ostracon expects them - such work is handled for them in the methods contained within *Ostracon.js*, with connection, maintenance, and even recovering from a disconnect abstracted away.

- **Scalability:** "Twitch Plays Pokemon," the original inspiration for Ostracon, is simultaneously playable by thousands of participants from all over the world. In creating Ostracon, we decided to have scalability built into its core.

- **Simplicity for clients:** We didn't want clients to have to install software, compile anything, or otherwise mangle their computers in order to participate in an Ostracon-based application. It thus made sense to

implement our front end as a web application. Technically, any software that supports HTML5 WebSockets can connect to an Ostracon server (leaving the door open for a client application built atop python or almost any other modern language), but given that WebSockets are supported in every modern browser, we intend that the primary instances of Ostracon will be Web-based. The only requirement, then, is that users do not run IE7 or any similarly deficient web browser (in which case they don't deserve to play anyway!).

1.2 Design Decisions

Several design decisions were made early in the development process that fundamentally influenced the direction for the project.

1.2.1 Languages

Naturally, one of the earliest decisions to make was the primary programming language for the library. Initially, we had intended to use Python, since there are a plethora of packages for scalably handling HTTP connections, most notably, *Twisted*. Python would have also had a wider audience than the less commonly used Erlang.

However, we decided that Erlang's lightweight processes would be conducive to the scalable nature of the project. Additionally, since we had less experience with Erlang, we wanted to take advantage of the opportunity to expand our understanding of the language.

1.2.2 Data Storage

Another crucial decision was the storage mechanism for votes and state. Simple message passing to communicate the records of votes and state entailed too much overhead for massive scalability and would not suffice on its own.

Databases, both SQL and NoSQL, were considered for keeping consistent records of votes and state that multiple modules could refer to. However, using SQL databases added unnecessary complexity to the project and did not fit with the functional programming model we had been following. Thus, we decided that the best way to keep consistent records of votes and state was to use Erlang's ETS data structure, a thread-safe, constant time, key-value store accessible among multiple processes and modules.

1.2.3 Communication

Since voting is done over a web interface, we needed a method of fast, bidirectional communication. Our initial thought was to use the JavaScript library SockJS and a SockJS plugin for Erlang to set up sockets to send votes and receive updated states at the end of each cycle. However, we soon discovered Cowboy, an Erlang library for HTTP servers and HTML5 WebSockets, which suited our needs much better.

2 Implementation

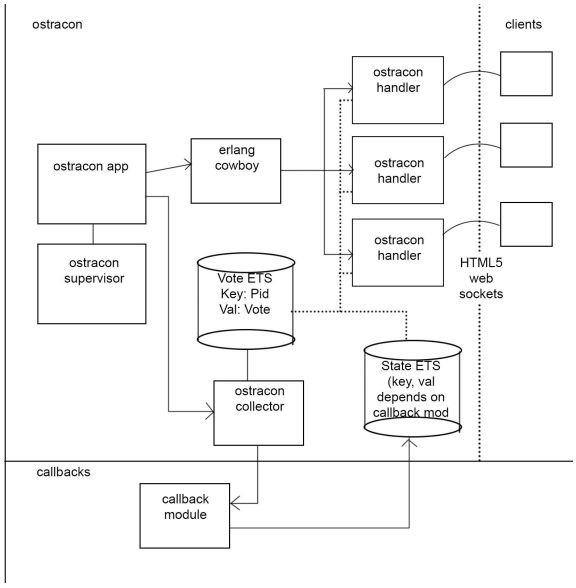

Figure 1: Breakdown of modules

2.1 General Flow

The library is started by *Ostracon App*, which has a supervisor link with *Ostracon Supervisor*. *Ostracon App* initializes the vote and state ETSs. When a client connects to the web interface for the first time, an *Ostracon Handler* process is spawned to set up a dedicated WebSocket with that client.

When a client votes, its handler receives the vote and records it in the vote ETS. If a client votes more than once in a cycle, only its last vote is counted. At the end of each cycle (whose length in milliseconds is specified in the *Callback Module*), the *Ostracon Collector* gathers all the votes from the *Vote ETS* and computes a histogram of the votes, sorted in descending order by most popular vote. The *Vote ETS* is then wiped clean of all records in preparation for the next cycle and the histogram is sent to the *Callback Module*, which

decides how votes should affect state. Whatever effect the votes have is recorded in the *State ETS* by the *Callback Module*.

When a state request is made, the *Ostracon Handler* translates the *State ETS* into JSON and sends the information over the WebSocket.

2.2 Ostracon

The Ostracon library consists of the following modules.

2.2.1 Ostracon.js

This is the front end JavaScript module that connects with application-specific front end code. It sends votes to the handler and receives the updated state from the handler to give to the client. The Ostracon front end library exports the following functions:

```
ostracon.start();
ostracon.getState();
ostracon.pushVote(vote);
ostracon.requestState();
```

2.2.2 Back End

- *State ETS* – This is the record for the state, which is determined by the callback module. The options are *named_table* (so that the table can be referred to by its atom name), *set* (for have constant time access and to ensure that each client only gets one vote per cycle), and *public* (so that other modules can read from and write to the ETS).

- *Vote ETS* – This is the record for each cycle's vote and has the same options as the *State ETS*.

- `ostracon_app.erl` – This module starts off the voting procedure by creating the *Vote ETS* and *State ETS*, then initiates the collector module. It also establishes a link with a supervisor. Finally, it sets up Cowboy to spawn instances of *Ostracon Handler* upon connection requests.

- `ostracon_sup.erl` – This module monitors *Ostracon App* for failure and has a one-for-one restart policy with it.

- `ostracon_collector.erl` – The collector gathers votes from the *Vote ETS* every cycle and computes a vote histogram to send to the callback module. The length of each cycle is determined by the *Callback Module*.

- `ostracon_handler.erl` – This module communicates with the client via WebSocket; it receives the client's vote to record in the *Vote ETS* and sends the updated state to the client. *Ostracon Handler* also sends the current state to any new client that joins in the middle.

2.3 Application-specific

- `callback_module.erl` – This is the back end written by the application developer using Erlang. It determines how each vote should affect state, writing state updates directly to the *State ETS*.

- Front end code – Application-specific display logic that communicates with the back end using *Ostracon.js*.

3 Proofs of Concept

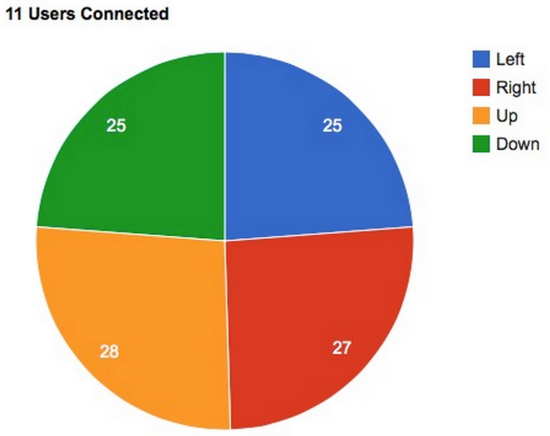

Figure 2: A basic pie chart demo

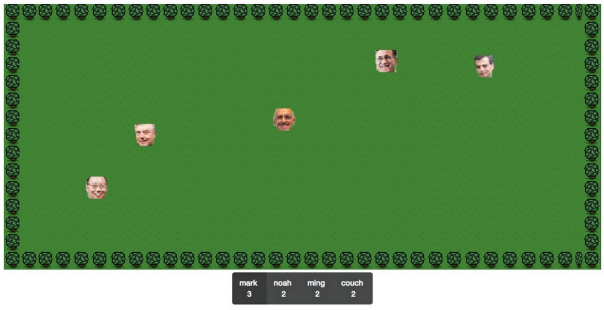

Figure 3: "Let's Have Some Fun[ding]", a game built on Ostracon

We built two examples to demonstrate the efficacy of our library.

The more basic of the examples, the pie chart of keystrokes in Figure 2, provided an interactive visualization of how the votes were counted and aggregated in real time. It served as a basic proof of concept for how a shared application could be written using the Ostracon library, and only took about 100 lines of application-specific code.

The game in Figure 3 was a more ambitious project. It included discrete rounds of voting, as well as the more complex notion of team-specific votes and keeping track of states across rounds. Though imperfect, this example achieved what we had intended Ostracon to accomplish: multiple users controlling multiple players, all voting on and streaming the same global state at the same time.

4 Conclusions and Reflections

There are numerous possibilities for applications using Ostracon. Because the *Callback Module* and the front end can be configured to the developer's specifications, the possibilities are almost limitless. Any keystroke, button push, mouse swipe, text input, etc. has the possibility to count as a vote.

4.1 Results

The minimum deliverable of building the underlying voting software and a simple callback module game in which multiple clients per character could race toward a common goal was achieved. Though it was not precisely a maze game, the game idea was at its essence the same, so the minimum deliverable was achieved. We're happy to report that the essentials of maximum deliverable of additionally creating a web app that uses Ostracon were also achieved. Despite some abstraction leakage, the project's components came together well.

Though Ostracon tests well with small numbers of users (i.e. less than 50), it is also worth noting that more extensive evaluation must be done to ensure it performs well with more users.

4.2 Design Evaluation

Overall, the design is appropriate and works well. The project was divided into task-specific modules and the parts connect and interact successfully. Perhaps the best decision was to use Cowboy for the WebSockets, since Cowboy integrated into our application easily and scaled cleanly.

One aspect of the design that could have been improved was the way in which state is requested and exchanged between the server and client - as it stands now, as implemented in the professors game, at every draw frame, the client pings the server with a state request, to which the server responds with the entire state. In an optimized version, the server would take advantage of its unique knowledge of when state changes, at which point it would automatically push the new state to every client. This would result in much fewer messages being passed between the client and server; the client could now expect the state cached in its Ostracon instance to always be up to date without sending constant state requests, and the server would avoid sending extraneous state updates when state hasn't changed.

We also feel that Ostracon misses the definition of a library somewhat. In the current version of Ostracon, the callback module must be in the same directory as the Ostracon code, which breaks the abstraction boundary Ostracon should achieve to be considered a library.

4.3 Division of Labor

As with any group project, dividing labor equally between four members proved difficult. Dan and Jeremy's efforts were marred by difficulty getting the project compiling, which slowed the momentum they had demonstrated during the design phase. Ultimately, Louis shouldered the majority of the foundational work of getting Ostracon up and running on Cowboy. This set the stage for Scott to take ownership of most of the front end for the Professors game, and in doing so spawned a productive balance of work between him and Louis in extending the back end of the game via its callback module. Dan and Jeremy put together much of the basic demo and investigated use of the Google Charts API.

4.4 Bug Report

The most difficult bug we found is not yet resolved completely: latency errors in collecting votes and updating state when sufficiently many clients are connected.

The bug was identified only in the later stages of the project, when we began testing with more clients. As a result, the goal of scalability was not completely fulfilled. To find this bug faster, we should have done unit testing and large scale tests earlier in the development process.

Though we have not yet solved this bug, we have suspicions of where it might be occuring. One likely bottleneck, as mentioned above, is that the server sends back the entire state every time a client pings the server with a state request, rather than propagating updates as they occur. Another possibility is that the *Vote ETS* is being accessed too frequently or wrongly. Further investigation into these bottlenecks will allow us to solve the delay we have encountered.

5 Future Work

In the future, we want to expand the project by refining the library and building more sophisticated applications using it.

5.1 Library Improvements

Our first priority will be solving the scalability errors we recently hit. Additionally, we want to make Ostracon into a bona fide library that can be included in Erlang applications and integrate with other back ends as well.

In addition, we could allow for quick patches to the Ostracon user's callback module by implementing code hotswapping without the loss of state. Doing so would greatly simplify implementing any significant changes that the Ostracon user feels are necessary.

A final step towards making Ostracon into a bona fide library is providing clear and ample documentation; as it is now, Ostracon can be quite cryptic to a potential user who may come across it on the Web. One of the most important parts of the documentation will be clear, easy-to-understand code examples.

5.2 Use Cases for Ostracon

Some obvious examples of additional possible uses of Ostracon include a democratically distributed Final Fantasy game, where the most popular keystroke among players connected to the website is the one the game executes.

One could also build a cooperative maze racing game, where multiple teams race through a maze, each team controlling one character. In this example, the most cooperative team would likely win since conflicting and unclear strategies would cause the maze character to backtrack or twitch.

Yet we believe the most interesting and potentially useful applications of the Ostracon library are not obvious, even to us. Ostracon's facilitation of democratic decision making through the Web may prove useful to applications other than goofy games. PR firms may use it to edit their advertisement campaigns or corporate branding. Machine-learning could be used by the callback module to analyse patterns in keystrokes in real time. Ostracon's functionality will be explored as the library continues to develop, and is distributed and used across various applications on the Web.