Josh Mermelstein
Anschel Schaffer-Cohen
Andrew Shum
Jonathan Wilde

PresidentialFanFiction: Final Report

Table of Contents

Definitions

- Voting scheme -- see http://en.wikipedia.org/wiki/Voting_system for details

Project Deliverables and Status

Minimum

- ☑ At least two voting schemes
- ☑ Distributed voting system with multiple polling centers and concurrent results for different schemes
- ☑ Abstract interface for new schemes
- ☑ No one can vote twice at different stations
- ☑ (Addition during refinement.) Ability to add new voting booths during registration.
- ☑ (Addition during refinement.) Convenient command to start up the entire system.

Maximum

- ☐ Crypto
  - Not a concurrency problem.
- ☐ All the interesting voting schemes
  - We implemented a lot, but there's more.
- ☑ Websites for voting and for live results
- ☑ Standard voting theory counterexamples
  - There are criteria which not everyone can meet. There are classical examples of ballots which demonstrate this. It would be nice if we could look at those examples easily in this system.
- ☐ Recount.
  - After careful discussion, we decided this was impossible given anonymity
- ☑ (Addition during refinement.) Handle situations where booths, talliers, etc. fail.
- ☑ Add voting schemes during a vote in progress.

Reflection

- Achieved all minimum deliverables and most relevant maximum deliverables.
- Maximum deliverables we did not accomplish were either not feasible given anonymity goals or not really relevant to a concurrency project.

Design Decisions

Key Problem

- How do we ensure that people don't vote more than once without having a *gigantic bottleneck*?
- How do we keep detailed voting records relatively anonymous?
- Furthermore, how can we partition voting across different possible booths efficiently to prevent a booth from getting deluged with people?

Our Solution

- People can register long before they vote, so it’s okay if registration is slow.
- What matters most is ensuring that there’s no potential for people to register twice with the same credentials.
- So, we have a small bottleneck in the form of a centralized registration server. Writes have to go through a single node. The registrar tells each booth who will be allowed to vote there.

Ideas we rejected

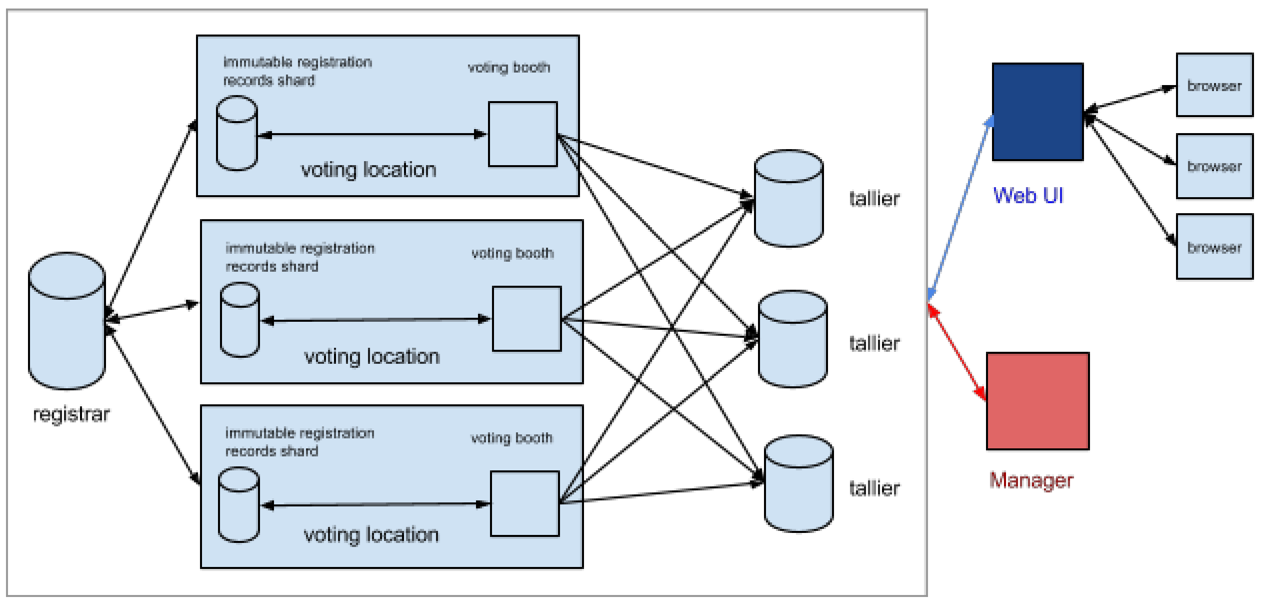
- No registration server:
    - Have the booths communicate with each other:
        - High level of complexity--all registrations have to talk to all booths, leading to lots and lots of cross-node traffic on voting day; and more latency for voting.
        - Doesn’t really solve the problem, on top of introducing highly confusing semantics.
        - Reduces anonymity, because every booth has to be aware of every possible voter and when they’re voting.
    - Have some sort of process sift through detailed voting records after the fact to deduplicate.
        - In order to identify duplicate votes we have to sacrifice anonymity
- Per-booth registration servers:
    - Potentially makes it marginally harder to de-anonymize voting data because there’s now two separate systems to correlate.
    - But actually, gets us back to the complex communications situation of having booths communicate with each other.

Reflection

- Best decision: liked how we handled the passing of votes--resulting in security properties:
    - Watching any one node does not break anonymity.
    - Snapshot of entire system does not break anyone’s anonymity.
    - Cannot vote twice with same credentials.
- What we would have done differently:
    - Define terms more clearly and use more consistently across the project:
        - Vote vs. Ballot
        - Voting scheme vs. Tallier
        - Voting location vs. Booth
        - User in lots of different places for different things

Overall System Design

Everything here is going to be implemented with Erlang. We're using Yaws (installed via Rebar) as our webserver framework.



- This diagram demonstrates the central registration server. It receives registration requests from web browsers, and forwards the changes to booths, which hold a set of voters that are registered for that booth.
    - When the voter registers, the registrar processes the request and sends it down to the appropriate booth. When the booth returns a valid response, the registrar informs the web UI that they are ready to vote.
- Each booth takes the vote in the format that we will describe later, checks that the person is registered and didn't vote already, stores the vote locally, and marks that that person has voted.
    - When there is a sufficiently large set of untallied votes, it batches the votes (tallied in different ways for different schemes), and sends the batches down to their corresponding talliers.
    - Notes on this decision: Sending batches of votes out anonymously makes it much harder for an attacker to determine from observing the network or the raw data at a single given node in the system to determine how a single person voted.
- Talliers have a function for combining a batch with their existing internal tally. They also have a function to output the winner and results, sending that information to a results server.
    - Note: each tallier corresponds to one voting scheme.
- The results server can show combined statistics from across the talliers to a user all on one page. These results update asynchronously using server-sent events.
- A single web server that acts as a UI for everything. Booth pids will be passed in via URLs.

- Adding a manager module to start up the entire system and monitor when parts of it fail.

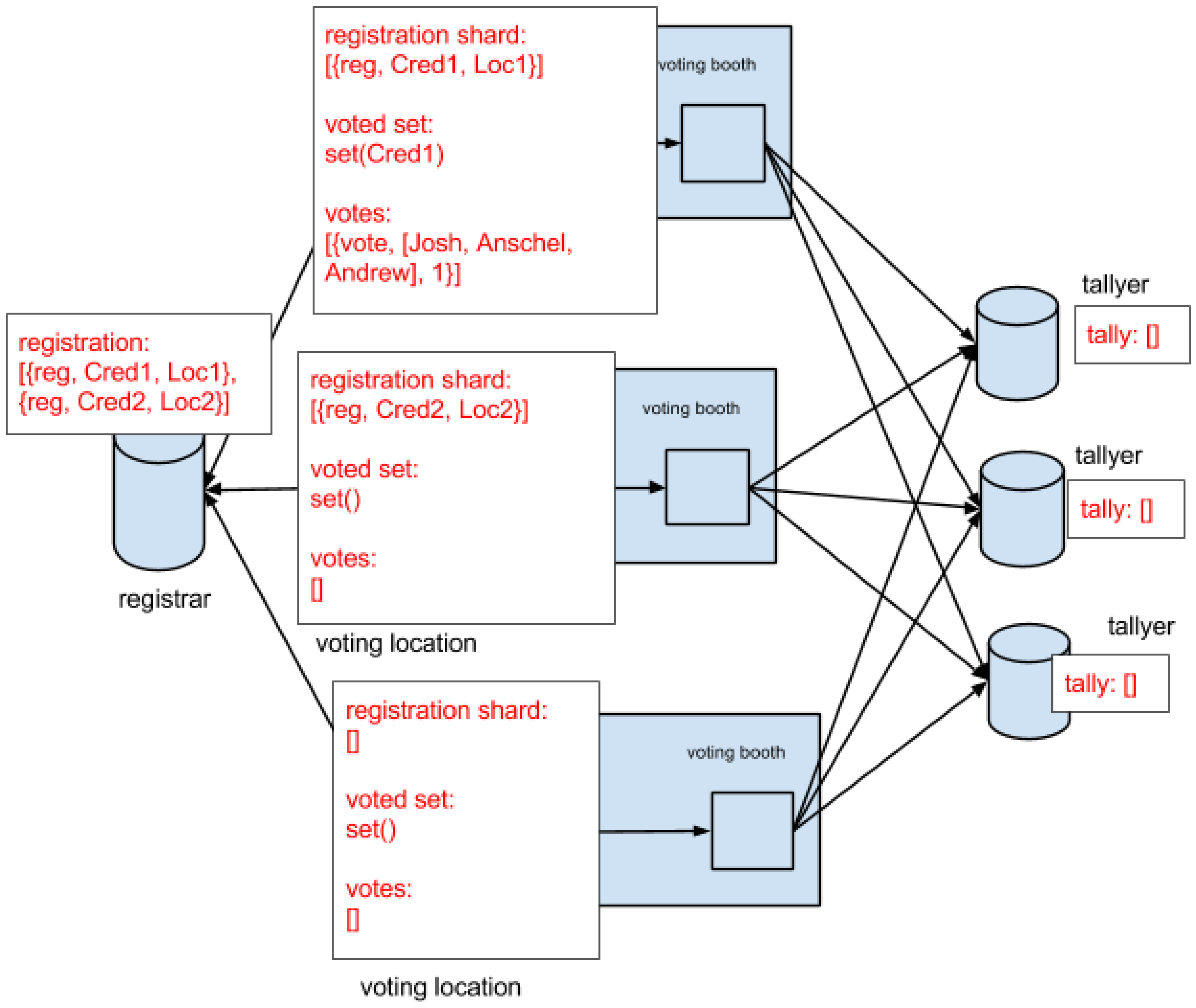
Key Data Structures

In this second, we document the message formats used to communicate between modules that different team members wrote.

- Registration Request
    - {register, Credential, UserPid}
    - UI sends to registrar.
    - Credential identifies the user.
    - UserPid denotes where to send the Registration Confirmation.
- Registration Request for Booth
    - {registration, RegistrarPid, VoteCredential}
    - Registrar sends to booth to register a person there.
    - RegistrarPid denotes where to send Registration Confirmation.
    - VoteCredential is a nonce that the registered user will use at the booth.
- Registration Confirmation for Booth
    - {registered, VoteCredential, BoothPid}
    - Booth sends to register to denote that registration was successful.
    - VoteCredential is the nonce described above.
    - BoothPid is the booth sending the message.
- Registration Confirmation
    - {success, {VoteCredential, BoothPid}, RegistrarPid}
    - {failure, no_booths, RegistrarPid}
    - {failure, registered, RegistrarPid}
    - Registrar sends to UI once registration is complete.
    - VoteCredential is the nonce above.
    - BoothPid is the booth where the user can vote.
    - RegistrarPid is the registrar.
- Ballot Request
    - {ballot, UserPid, VoteCredential, Vote}
    - UI sends to booth to submit a vote.
    - UserPid denotes where to send ballot confirmation.
    - Vote Credential is the above nonce.
    - Vote
        - {vote, [CandidateA, CandiateB, ...], Approved}
        - Candidate* are strings of candidate names.
        - Approved is the number of Candiate* instances in the list that the user has checked “approval” boxes next to.
        - Example 1: Suppose somebody supports only Josh from candidates Josh, Anschel, Andrew, Jonathan:
            - {vote, [“Josh”, “Anschel”, “Andrew”, “Jonathan”], 1}

    - Example 2: Suppose somebody rank votes Josh, Andrew, Jonathan, Anschel and does not approve of Anschel:
      - {vote, [“Josh”, “Andrew”, “Jonathan”, “Anschel”], 3}
- Ballot Confirmation
  - success
  - {failure, invalid_registration}
  - Booth sends to user when ballot request has been processed.
- New Scheme
  - {newScheme, Batchfun, Pid}
  - Announces to a booth that a new tallier is available
  - Batchfun is the function to use when batching votes for this tallier
  - Pid is the pid of the tallier
  - From now on the booth will send batches to this tallier along with any others it knows about.
- Voting Scheme
  - Either a tuple of four functions or a module that exports four functions
    - empty/0 is the result when no one has voted
    - batch/1 turns a list of votes into scheme-specific data structure
    - combine/2 combines two batches
    - winner/1 takes a batch and returns a list of winners

Example Mid-Run



- What this diagram demonstrates is that:
    - Two voters have registered.
    - One has voted.
    - No batch data has been sent to the talliers yet, so they are empty.

Most Recent Changes to Design

- We added a “winner collector” to do two things:
    - Maintains cache of most recent winners that we can send to UI when the page loads initially.
    - Track all pids of talliers and allow processes spawned by YAWS to subscribe to new results from those talliers and unsubscribe on termination.
- We added confirmation message to vote submission.
- This is short, because we had almost completely finished implementation by time of the first design refinement.

Bug Overview

- When using magic.yaws to submit a large number of ballots in a short space of time, some nondeterministic behavior was observed.
- We found it when prepping for our presentation, we constructed some ballots to show that different schemes produce different results.
    - Note that during our presentation, the results presented were incorrect as a result of this bug
- Anschel found it in a half hour
- The bug was caused by the interaction of two mistakes
    - There should have been a nested receive to handle the registration confirmation
    - There was a catchall receive elsewhere that caught the registration confirmation when it should have been sending out another vote, causing that vote to be dropped
- How it was fixed
    - Two fake voting schemes were created to dump all ballot information to the webUI.
    - These made it clear that some votes were missing
    - Then Anschel walked through the code manually and found a message being sent by the booth that wasn't being caught anywhere
    - He wrote code to receive it with the goal of understanding the message's contents and that unexpectedly fixed the bug
    - Jonathan later explained how this change had fixed the bug and then removed the catchalls
- How we could have found it in less time
    - Having a code style policy to avoid catchalls could have prevented this bug in the first place
    - Having debugging schemes in place by default could have made the bug hunting go faster
        - It would have helped us pinpoint the source of the problem

Project Development

Division of Labor

Plan

Josh:

- Booths.
- Supervisor process.

Anschel:

- Tallying scheme framework.

- Specific schemes.

Andrew:

- Registration servers.

Jonathan:

- All web servers code for users, ballots, and winners.

Reflection

- The original division assumed that the web work was a lot easier, but turned out one of the most time consuming portions of the project.
- Because we did very effective planning on message formats, people were able to easily work independently on different sections of the project.
- The only real two things that were blocked until late in the project were:
  - manager.erl: Needed all components to exist in order to start them.
  - magic.yaws: Needed manager.erl to exist to flush votes.
- In terms of what we would've done differently:
  - Better planning--especially with web stuff--surrounding timing of integration testing and managing Thanksgiving travel.

Overview of Our Code

Josh

- booth.erl
  - Is a server with an init/0
  - Manages the routing of votes between voters and talliers
  - Receives credentials that are cleared to vote from the registrar
  - Receives ballots from voters
    - (which it doesn't actually examine)
  - Receives talliers pids and functions to use when batching data for those talliers
  - Batches votes and sends the batches to their corresponding talliers
- manager.erl
  - Is a server with an init/0
  - Starts all other servers and informs different components about each other
    - Registrar
    - All booths
    - All schemes
    - The frontend supervisor
    - The winner collector
  - Also informs relevant components when a new component (booth or tallier) added

Anschel

- all_votes.erl
    - Fake “Voting Scheme” used for debugging.
    - The “winner” is just a list of all votes.
- vote_count.erl
    - Fake “Voting Scheme” used for debugging.
    - The “winner” is the number of votes cast.
- count_map.erl
    - A module for maps whose values are numbers
    - Combining two count maps adds their values
    - This is used in almost every voting scheme
- manager.erl
    - Is a server with an init/0
    - Starts all other servers and informs different components about each other
    - Also informs relevant components when a new component (booth or tallier) added
- schultze.erl
    - The scheme generally regarded as best by voting scheme nerds
    - The details of how this works (and why it’s good) are kind of complicated. More information at Wikipedia: http://en.wikipedia.org/wiki/Schulze_method
    - Note: this file includes an efficient, functional implementation of the Floyd-Warshall algorithm. This isn’t a concurrency problem but I’m proud of it.
- debug.erl
    - Various functions useful for testing talliers
- pairwise.erl
    - Functions for turning a ballot into a weighted directed graph of head-to-head votes.
    - This is used heavily by schultze.erl; it could also be applied to other methods we did not implement, like:
        - Single-runoff
        - Calculating the Schwartz set.
- tally.erl
    - Tallier
    - Unlike other init functions, tally:init/2 takes two parameters. The first specifies the voting scheme (either a module or a tuple of functions) and the second is a callback with which to announce the winner.
    - The callback exists essentially because tally.erl was written first; if it had been integrated later in the process it would probably be sending a message to the YAWS code directly.
- point_scheme.erl
    - Turns out many voting schemes work like this:
        - Each ballot gives some points to each candidate
        - The candidate with more points than everyone else wins
    - All such schemes can be represented by just the function from ballots to points

Andrew

- registrar.erl
    - A server with functions init/0 (current node) and init/1 (remote node)
    - Manages registration and booth assignment for voters
        - Prevents multiple registration by one person
    - Receives registration information from the web UI
    - Generates unique voting password for each voter
    - Notifies appropriate booth of valid voting passwords
    - Receives acknowledgement of valid voting passwords from booth
    - Sends confirmation of registration to web UI

Jonathan

- frontend.erl
    - Configures the embedded YAWS server. Takes list of processes from YAWS and tells frontend_sup.erl to start them. Patterned after YAWS example code.
- frontend_config.erl
    - YAWS breaks .hrl includes for records in .yaws files.
    - Contains convenience methods to read fields (RegistrarPid, ManagerPid, Candidates) from a configuration record for the web server.
- frontend_sup.erl
    - manager.erl starts this. Supervises processes started by frontend.erl. Patterned after YAWS example code.
- result_events.erl
    - Catches requests on /result_events, spawns a child process on correctly-formed requests for server-sent events using gen_server to pass messages from winner_collector.erl to client.
    - Requests full winner list from winner_collector.erl.
    - Subscribes and unsubscribes from winner_collector.erl.
- vote.app.src
    - Mostly autogenerated file by rebar to tell rebar where to look to compile things.
- winner_collector.erl
    - Receives winners from all talliers.
    - Allows other processes to subscribe to notifications about results from talliers.
    - Allows other processes to request a list of the most recent results received from all talliers so far.
    - Provides function that can generate another a function that can be passed as a tallier's results method.
- .../www/yaws/:
    - index.yaws
        - Shows a list of other pages in the app.
    - magic.yaws

        - Contains simulate_votes method to register users and submit ballots for them to associated booths.
        - Tells manager to inject votes when the user clicks a button.
        - Tells manager to flush all votes when the user clicks a button.
    - register.yaws
        - Displays a registration form.
    - register_post.yaws
        - Processes submissions from registration form.
        - Displays success/error messages.
    - vote.yaws
        - Displays a vote form, using list of candidates from the record unpacked by frontend_config.erl.
    - vote_post.yaws
        - Contains get_normalized_ballot, to normalize and validate the voting form submitted by the user.
        - Processes the ballot from the vote form.
        - Displays success/error messages.
    - winners.yaws
        - Connects to /result_events using browser EventSource API, managed by result_events.erl.
        - Formats list of winners that come in into table.