Report: /20

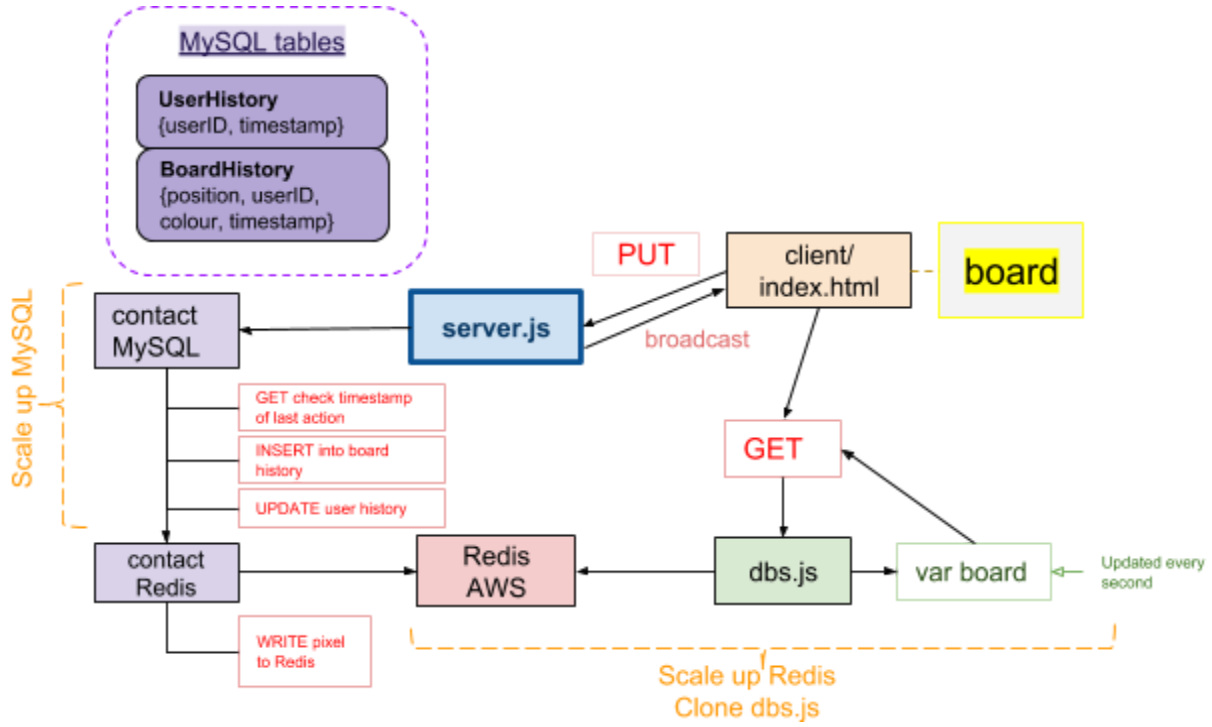
60% outline of architecture

80% understands strengths and weakness

80% as above, but some proof (even if it does not work)

Assignment 3 Report

Adam Lee
Selby Gomes
Carrie Aiko Fallows



Our r/place starts at our main server.js, which uses AWS EC2 (Elastic compute Cloud), where we are listening for incoming clients accessing our html page. The client displays the board, which is filled with the most current pixels from the Redis DB as of 1 second. This board-grabbing is done directly through the html page, where a GET request is made to dbServer.js. dbServer.js has a variable “board” which contains/caches the latest board data up to and no later of data than one second ago. This variable is renewed with the freshest DB from Redis AWS, which is stored as a key value pair of “canvas” and a bitfield of the image. dbServer.js takes the bitfield and divides into UInt8Arrays with bit shifting to get the colour for each pixel of a four-bit size. html GET requests to dbServer will receive whatever sits in the variable at the time.

We have scaled image handling by assigning a replica of dbServer.js to a quadrant of the image, which accesses a “canvas2” key and a different bitmap. (Note, we would have created a

new redis DB but for cost restrictions in AWS we keep it as another key). Currently, we have two redis DBs keys, and two dbServer.js accessing one each, to show proof of concept for dividing up the image into quadrants. To accommodate this, we also divided the INSERT from server.js to enter pixel data into the correct redis key.

The client also has the ability to send in a pixel; the html page has input boxes to submit a form through a put request to server.js. This request consists of an x and y value, and a colour (selection of 16 different colours). server.js then contacts MySQL (uses AWS RDS), which makes a few requests:

- First is a GET timestamp, to check the user's submission history. Users can only submit a pixel change every X minutes, so a change will be denied if they are under the X minutes, otherwise they will proceed
- Second, an INSERT will be performed to log the board history
- Lastly, we UPDATE the user history with the new timestamp

Then (assuming all of the above was successful) server.js will write the pixel to Redis (which uses AWS ElastiCache), with {canvas (quadrant) ,x, y, colour} (converted to the correct bit/offset placement in the bitfield), and broadcast the pixel to all of the other clients connected to server.js's webSocket so that their board is updated with the latest pixels.

How this can be scaled:
dbServer.js

- Partition the image into quadrants, and create [partition] number of dbServer.js
- Each dbServer.js takes a quadrant, and makes a call to the respective Redis DB that handles the storage of such image quadrant
- We currently have two dbServer.js and Redis key instances, but we can create more, and add a load balancer so that we have a handler redirecting to the appropriate dbServer.js for the quadrant

Our r/place has point-colour submission through a form (which tracks user pixel submission for the X minute limit) as well as scribble functionality that is available for all users the same timestamp limits.

General Reflection:

We believe our handling of Redis is okay since we are only pinging Redis once per second. I.e our node server can handle numerous connections but they're all returning a single local variable instead of pinging Redis once per call.

Having only one Server.js is okay since according to https://blog.jayway.com/2015/04/13/600k-concurrent-websocket-connections-on-aws-using-node-js/ web sockets can handle 600k concurrent connections so this shouldn't be too much of an issue since that is more of a workload than actual r/place.

If we had more resources we could actually create separate instances of redis under elasticache to hold different segments of the canvas (At the moment they're just separate keys accessed by separate servers)

It would also be good to look into making Server JS scalable in order to reduce the bottleneck OR perhaps making each canvas a Pub/Sub relation per server JS

-> In otherwords Each quadrant represents its own WebSocket connection that clients Subscribe to in order to get the updates in that quadrant

Another bottleneck might be the MySQL query but they're done asynchronously so we are not waiting for these queries to be made.