Scalable Recreation of r/place

Amazon Web Services Project

Parmbir Nagra

Andrew Juskiewicz

Nicholas Choi

FALL 2020

Table of Contents

1 Architecture

1.1 Diagram

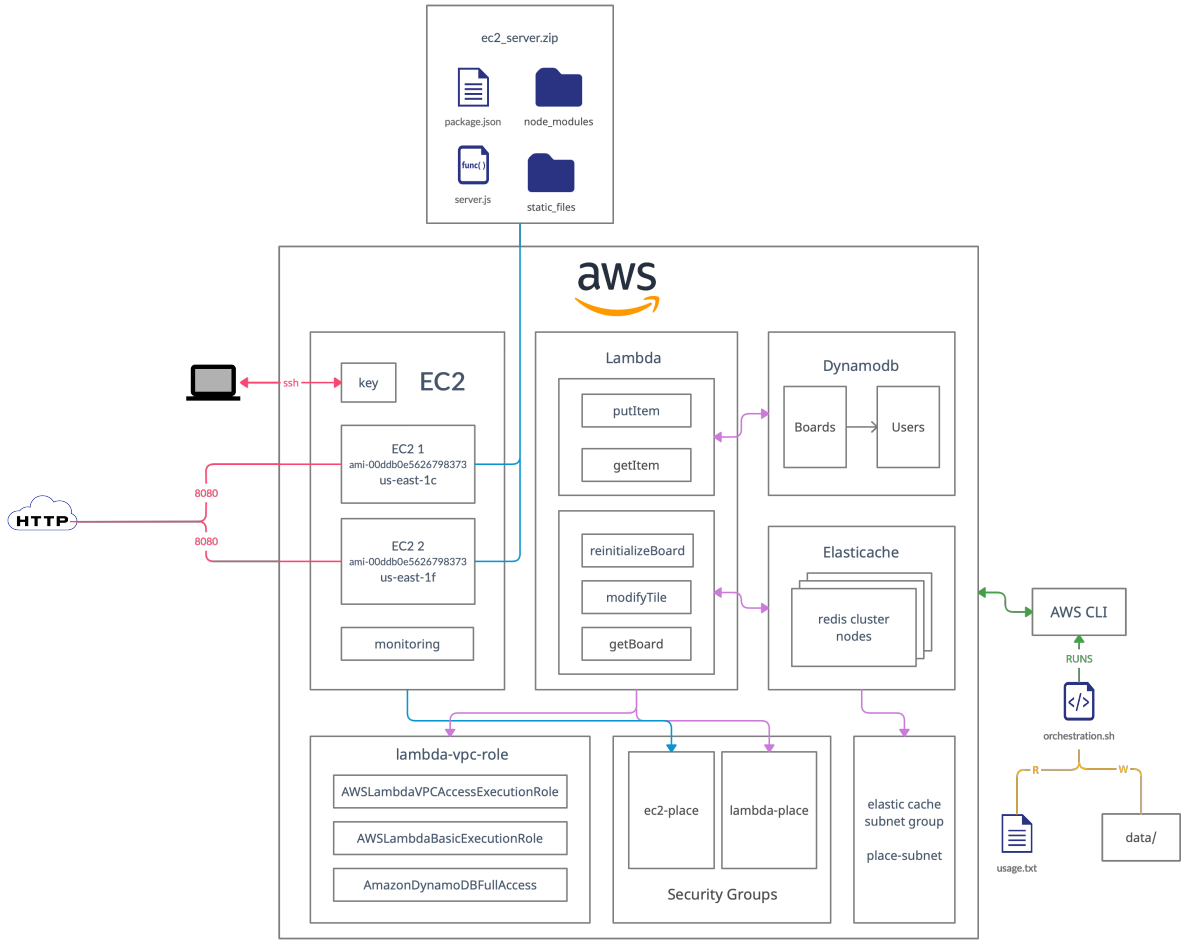

Figure 1: Diagram outlining the system architecture

2 Running The System

1. Download aws cli and do aws configure with your account:

   - Change line in `orchestration.sh` with your parent directory of your `.aws`

     ```
     #!/bin/bash

     # Change this line to the parent directory of your .aws
     AWS_PAR_DIR="/Users/parmbirnagra/"
     ```

2. Run `./orchestraction.sh` on terminal while at the base-level of the project or look at `usage.txt` to see the usage how to START or STOP the system:

   ```
   Usage: ./orchestration [OPTION] <COMMAND>
   Options:
     [ -c  | --create  ]  <ro, sg, cc, db, lf, kp, e2>
     [ -d  | --delete  ]  <e2, 1, 2 kp, lf, db, cc, cg, sg, ro>
     [ -t  | --status  ]  <cc, lf, e2>
     [ -s  | --start   ]  <e2>
     [ -k  | --kill    ]  <1, 2, e2>

   Usage: ./orchestraction [-c | --create] COMMAND
   Create commands:
        ro : Create role
        sg : Create security groups
        cc : Create cache subnet group and cluster
        db : Create database
        lf : Create lambda functions
        kp : Create key pair
        e2 : Create ec2

   Usage: ./orchestraction [-d | --delete] COMMAND
   Delete commands:
        e2 : Delete all ec2 instances
         1 : Delete ec2 1 instance
         2 : Delete ec2 2 instance
        kp : Delete key pair
        lf : Delete lambda functions
        db : Delete database
        cc : Delete cache cluster
        cg : Delete cache subnet group
        sg : Delete security groups
        ro : Delete role

   Usage: ./orchestraction [-t | --status] COMMAND
   Status commands:
        cc : Status cache cluster
        lf : Status lambda functions
        e2 : Status ec2

   Usage: ./orchestraction [-s | --start] COMMAND
   Start commands:
        e2 : Start ec2 node servers

   Usage: ./orchestraction [-k | --kill] COMMAND
   Kill commands:
         1 : Kill ec2 1 node server
         2 : Kill ec2 2 node server
        e2 : Kill all ec2 node servers
   ```

3. (a) The output of the aws cli commends go to the data folder

4. The order to CREATE the system do:

   (a) Create Role `./orchestraction -c ro`

   (b) Create Security Groups `./orchestraction -c sg`

(c) Create Cache Subnet Group and Cluster `./orchestraction -c cc`

(d) Create Database `./orchestraction -c db`

(e) Create Lambda Functions `./orchestraction -c lf`

- Make sure Security Groups are created
- Make sure Role is created
- Make sure Cache Subnet Group and Cluster is created

(f) Create Key Pair `./orchestraction -c kp`

(g) Create EC2 `./orchestraction -c e2`

- Make sure Security Groups is created
- Make sure Role is created
- Make sure Cache Subnet Group and Cluster is created
- Make sure Key Pair is created

(h) Start EC2 Servers `./orchestraction -s e2`

5. To order to DELETE the system do:

(a) Create Kill EC2 Servers `./orchestraction -k e2`

(b) Delete EC2 `./orchestraction -d e2`

(c) Delete Key Pair `./orchestraction -d kp`

(d) Delete Lambda Functions `./orchestraction -c lf`

(e) Delete Database `./orchestraction -c db`

(f) Delete Cache Cluster `./orchestraction -c cc`

(g) Delete Cache Subnet Group `./orchestraction -c cg`

- Assuming Cache Cluster is deleted

(h) Delete Role `./orchestraction -d ro`

(i) Delete Security Groups `./orchestraction -c sg`

- Assuming Cache Cluster is deleted

3 Specifications

3.1 Spam Protection

Similar to Reddit's blogpost: *"How We Built r/Place"*, we decided to record each pixel placement or *transaction* in our board within our DynamoDB backend. Specifically, each row within our Board table displays a pixel placement in the form: {x, y, author, colour, time}.

In more detail, whenever a user tries to place a pixel, our NodeJS backend will receive information about the colour, x-y coordinates, author (remote IP address of websocket), and will call our AWS Lambda function: PutItem with these parameters. This function: PutItem will get a current timestamp and query our DynamoDB database to see if there exists a transaction within the past five minutes from this specific user. If there is a transaction from this user within the past five minutes, then our PutItem lambda function will return 403 to our NodeJS. Then our NodeJS server will return a message to our user telling it that it's request was denied because the user needs to wait for 5 minutes after the previous pixel placement.

3.2 Broadcast Updates

In our NodeJS server, we use the Websockets/ws library to handle connections because it is lightweight and faster than Socket.IO. Whenever a board update occurs (user asked to place a pixel, it was approved, and the board was updated), we broadcast a message to every live websocket with the new board details so that they can update their canvas on the front-end. Therefore, all users will automatically get updates because every time a tile is approved and placed, our NodeJS server sends this new information to each websocket.

Also, to have two availability zones, we have two different EC2 instances which each run a NodeJS server instance. In order for these two servers to coordinate, we utilize Redis publications and subscriptions. In other words, if one NodeJS server approves a tile, then it will publish that message through the channel and it will be received by the other NodeJS server. That server will then broadcast and update all of it's live websockets with the new information.

3.3 Scalability

In order to handle, manage, and update the board state for a 1000 x 1000 board (1 million pixels), we decided to follow the design in Reddit's blogpost: *"How We Built r/Place"*. Specifically, we utilize Redis, a fast, in-memory cache, to keep track of our board state. Also, to keep our data compact, we store the entire board state in a bitfield where each pixel is represented within 4 bits (the 4 bits represent the colour, and the offset represents the location). In order to make our Redis solution scalable, we utilize Amazon's Elasticache service which allows us to easily vertically scale our system because we can easily change the specifications of our cluster machines by selecting different ones. For instance, in a production environment, we can easily upgrade our Redis cluster machines by selecting cache.r6g.16xlarge machines with 64 virtual CPUs, 420GB memory, and a 25Gb network connection to replace our current free-tier cache.t2.micro machines with 1 virtual CPU, 0.555GB of memory, and a low to moderate network connection. Also, this is easily horizontally scalable because Elasticache allows us to easily increase the number of nodes and replicas we use in our Redis cluster. Since all of our systems are on AWS systems, we can do the same thing and easily vertically and horizontally scale all of our other systems this way (*ie.* changing AWS Lambda settings so it can handle more concurrent connections).

However, one weakness of our application is that we did not have enough time to implement clustering/multithreading on our NodeJS server. In otherwords, even if we upgrade the CPU on our EC2 machines, we cannot utilize additional CPU cores, because our NodeJS instances currently only utilize a single V8 instance. However, we can make use of vertically scaled memory (additional RAM), because we added the -max-new-space-size=MEMORYSIZE option when starting our NodeJS servers. We determine this MEMORYSIZE when we run our bash orchestration.sh script. This allows us to utilize as much system memory as we want in comparison to NodeJS's default of 512Mb. We can horizontally scale our system though, by creating additional EC2 instances which will run additional instances of our NodeJS server. The only issue is that we currently don't have a reverse proxy which creates one central endpoint for all users to interact with.

3.4 Availability

We run two different EC2 instances of our NodeJS server to allow for our service to be available on two different availability zones. We can easily increase the number of availability zones by adding additional EC2 instances (cloning instances). However, as stated previously, one weakness is that each EC2 server will have a different endpoint, rather than one centralized endpoint for all servers (with a reverse proxy).

3.5 AWS

AWS Services:

1. Elasticache - We use Elasticache to host, monitor, scale, and conduct automatic healing for our Redis cluster and service. Elasticache provides a central endpoint so that we can access our Redis cluster and service.

2. AWS Lambda - We use AWS Lambda so that we can host and access our functions (which we made to call our Redis and DynamoDB services) on the cloud. The benefit to AWS Lambda is that it doesn't have any servers we need to manage and it is continuously scaling.

3. DynamoDB - We use DynamoDB as a backend database solution so that we can store transaction/row information for our board.

4. EC2 - We use EC2 instances in order to host and run our NodeJS server. Similar to Docker, EC2 allows us to monitor and manage our service. Also, EC2 allows us to provide our application in multiple availability zones (like CDNs like Fastly).

3.6 Security

Since all of our services are in AWS, our system is secure because in order to access any of our services (*ie.* DynamoDB tables, Redis cluster, etc), you need to have the proper AWS credentials for a role with proper permissions to these services. Also, services like our Elasticache Redis cluster are secure because you also need a specific VPC Role and Execution Policies that allow you to access the services we use (*ie.* AWSLambdaVPCAccessExecutionRole policy for our AWS Lambda functions to allow access to Elasticache). Moreover, our Elasticache Redis cluster is secure because you cannot even access the service (endpoint) outside of these AWS services. Lastly, we made sure not to add any outbound rules, and we only added the exact inbound rules that we required for our service (*ie.* port 6379 so our AWS Lambda function can access our Elasticache endpoint).

4 Summary of AWS Components

4.1 DynamoDB

In our system we utilise a DynamoDB database to store information about users and the board. We have 2 tables setup, one that keeps track of users and the timestamp of their last placement and a second table that keeps track of the pixels on the board (as a backup to the Redis cache). Our DynamoDB tables are modified and accessible by the NodeJS servers through our AWS lambda functions PutItem (to place a tile) and GetItem. As stated before, the PutItem function will return a 403 response if the user had already placed a tile within the last 5 minutes. Both tables are currently configured with read and write capacity units of 10 (representing 10 strongly consistent reads or writes per second). These configurations where chosen due to the free tier limits, however for paying customers DynamoDB is able to scale up the read and write capacities on demand (more info here: https://aws.amazon.com/dynamodb/pricing/on-demand/).

4.2 Redis and Elasticache

As stated before, we store our entire board state in memory through an Elasticache Redis cluster. We store the board state as a bitfield where every 4 bits is used to represent one pixel (the 4 bits represent the colour, and the offset represents the pixel location). Elasticache allows us to easily increase the number of Redis replicas (so that we can scale our cluster) and provides monitoring and healing. Because we were worried about exceeding free-tier limits, we have our Elasticache cluster set to cache.t2.micro with only one node. But, in production, we would use a much better machine with multiple replicas and the Multi-AZ option to increase availability.

4.3 Amazon Lambda

AWS lambda functions are used in our project to interact with the DynamoDB tables and the Redis cache (see relevant sections for details on these functions). We use AWS lambda functions because they are a serverless solution that scales continuously. In other words, if we have more and more concurrent users and requests, AWS Lambda knows to automatically scale our service by allocating more resources to run our functions/services. In addition, it makes our system very modular and if we ever want to change these functions, we just need to modify the code and redeploy the functions on AWS lambda (without touching the other parts of our system).

4.4 EC2

EC2 is used in this project to host the NodeJS servers for our website, with multiple instances running in different availability zones, that communicate updates to the board between each other (using Redis pub sub). The EC2 instances we ran used a basic Ubuntu image and t2.micro machine (due to free tier limits). Similar to Docker, EC2 provides monitoring, disaster recovery, and allows us to easily clone/duplicate EC2 instances.