**Objective**

A multi-user system needs to be designed to allow users to rotation of a cube and enable color changes on its faces.

Rotation is enabled by dispatch of x, y, z co-ordinates (floats), while the color change happen on click to the cube. The rules to select the color to change on the cube face are not mentioned, but it can be assumed that this is made available through some means in the form of the RGB components of the color (3 bytes).

History of color changes need to be maintained over the complete session. These may be accessed by any new player joining the collaboration.

Certain assumptions and design considerations are made while proposing this architecture

1. Collaborators may join the collaboration session across multiple regions. Cross region latency concerns need to be addressed.
2. The collaboration is built as a part of a larger web system, in which the web component latency requirements are not as stringent as those of the collaboration sessions.
3. The collaboration component that allows action on the cube is a separate component.
4. Authentication and authorization are requirements indeed, but do not need to be specified a part of this submission.
5. Since no concept of users getting a chance to make changes to the cube has been specified, it is assumed that the different users can concurrently make changes.

**Proposed Technical Stack**

1. AWS (EKS, ELB, API Gateway, Elasticache, Kinesis, DynamoDB, AWS Lambda, Cloud monitoring (Cloud Watch, Cloud Trail, LightSail etc))
2. Unity
3. Node.JS
4. JavaScript (originally I thought of TypeScript, given Google’s recommendations, but was not sure if Unity’s libraries supported TypeScript directly or indirectly)
5. Socket.io
6. Socket.io-stream
7. Another relational DB store (for relational use cases), MySQL
8. Redis (Redis Topics, pub/sub)

**Proposed Protocols**

1. Http(s)
2. WebSockets
3. RESP (pub/sub from Redis)

**Proposed System Architecture**

The system architecture is proposed on an AWS environment. However, with the component use proposed, the same may be ported to any other cloud vendor, who provide matching services.

|  |
| --- |
| Diagram  Description automatically generated |

The above architecture fronts the load via Application Load Baancer(s), which route the initial unauthenticated traffic to an API gateway from where the requests are routed to appropriate Node.JS instances deployed as EKS deployments. The request routing is performed by AWS Lambda functions.

Once authentication is completed and the necessary authorization grants established, the user will be able to access the different parts of the system via the web application or the collaboration room.

Once the user accesses the collaboration room, the Collaboration Room client component sets up WebSocket requests to the applicable region’s Node.JS gaming servers. These servers are developed using the Unity library and Socket.io.

The Collaboration Room is created by the first user who wishes to establish collaboration. Each collaboration room is uniquely identified by an identifier. This cause a new instance of cube to be created and cached in Elasticache Redis instance. This cube is uniquely identified by the collaboration room identifiers. The collaboration room and cube state are stored in the MySQL DB.

All users who are connected to the collaboration room are also marked as subscribers for Redis data change events.

Collaboration done on the cube are all routed vial the WebSocket connections. The changes done are done on the cache object, which raises events that are routed to the subscribing users and also invoke Lambda functions, that stream this data to AWS Kinesis and MySQL. AWS Kinesis further streams this data to DynamoDB store. A CDC between DynamoDB and MySQL ensures that the data are synchronized, even if the lambda updates to either one source failed.

**Authentication and Authorization**

This requirement is not specified in the problem statement, but is being added for completeness

The system enforces a user to authenticate, after which the user’s authorization grants are determined. The user’s session is identified using session tokens, which are proposed to be cached in the Redis in-memory cache layer. The average expiration life of this Redis entry can be set to be the average duration of collaboration sessions. A proper logout will enable the token to be cleaned out of the Redis in-memory DB. However, if the user abandons the session by closing the browser or if the browser crashes, the expiration will ensure that the token gets removed after some time.

**Cube Operations**

To perform cube operations, the user will navigate to a collaboration room, which will be created by the first user and the subsequent users will ask permission to enter the collaboration room or enter some passcode to gain entry. The collaboration room holds the cube instance. All changes on the cube are emitted to the connected users via the collaboration room.

The collaboration room is a web page that contains the component for displaying the collaboration area. The collaboration component is essentially built using the Unity libraries.

For the collaboration purposes, it is assumed that this collaboration component will establish WebSocket connection with the game server instances. The user’s request is routed to the correct game server instances by the AWS Application Load Balancer.

At this stage the collaboration component will be directly interacting with the Node.JS gaming instances and all the latency layers and possibilities have been removed from this series of interactions.

1. When the collaboration room is created, a Cube instance is created in a default position and the same instance is cached in the Elasticache Redis instance.
2. All connected users, aside from their socket connections are also registered as subscribers to the Cube change events in Redis.
3. A user’s actions on the cube will first cause the cube on the collaboration component to display a change.
4. The change made will be transmitted to the applicable region’s gaming server (Node.JS instance). All interaction of the client with the gaming server happens over socket communication and uses streaming.
5. **Concurrency handling -** The gaming server instance will acquire a Redis Lock on the cached cube and apply changes to the applicable attributes.
6. Using the Redis Pub/Sub pattern, the cube’s state changes are streamed to the different subscribers. This contains the data changes done.
7. The Collaboration rooms of the connected user’s receive the message and ensure that the cube is reoriented/recolored per the changes. (**Objective 1 is met**)
8. Pursuant to step# 5, the change events in the Redis also invoke Lambda functions that cause this data to be streamed to AWS Kinesis, from where it is streamed to DynamoDB store.
9. This very streaming can be used to also update the MySQL store or a CDC can be set up between DynamoDB and MySQL to transmit the change data. This change data can be used by newly logged in user to track the color changes that happened to the cube. (**Objective 2 is met**)

**Expected Scalability & Availability**

Essentially, the website and gaming system instances are hosted on EKS (Docker) containers. It is proposed that these instances be deployed as Load Balancer Deployments (Kubernetes Deployment objects) that are configured to run with the Application Load Balancer. The replica set can be defined to have min instances and max instances per the testing thresholds. This takes care of the scalability requirements.

Availability requirements are handled through a combination of redundancy and proactive monitoring and health dashboards that are proposed using the AWS tooling.

**Fault Tolerance**

In general, multi-zone deployments and multi-region deployments (clients access region closest to them) ensure that a full failure of this system does not happen. In cases of outages, or failures, the service levels may degrade for a short time, but then these are expected to be proactively addressed.

Failure of instances – The Kubernetes deployment definitions ensure that a certain minimum number of PODS are always up and running. A failure will cause Kubernetes to spin up another POD to ensure that the descriptor’s definitions are always honored.

The database instances (MySQL and DynamoDB), Kinesis are proposed to be clustered and the redundancy will ensure that a compete failure of the persistence layer never happens.

Risks

1. Internet latency of the users
2. Haphazard patching of AWS instances