Krikey Exercise

Ben Hornedo
ben@uptown4.com
10/27/2020

API Scaling

1. To Scale this service I use load balancing (Kubernetes or other)
   to allow multiple replicas of the player state calculation service. If this state calculation is very expensive,
   I would also consider adding a queueing mechanism, to queue state calculation requests, and have multiple replicas of the state calculation service to consume messages from that queue. Since this would now be an "asynchronous" service,
   I would need a mechanism to let the client know that state calculation is complete. I may consider implementing a socket-based mechanism (Socket.io/SignalR) to provide a channel for sending updated info back to the client.

2. To reduce query time I would consider precalculating player state,
   and using a key-value lookup for retrieving the latest precalculated state. Maybe REDIS.

Kubernetes

Build

Build service docker image:

1. docker build -t state-calc-service:latest .

Deployment

1. Confirm the Kubernetes cluster with at least 3 nodes.
2. Deploy the initial deployment.yml file to the cluster: kubectl apply -f ./deployment.yml
3. Deploy service to cluster: kubectl apply -f ./service.yaml

Notes

1. Container resources requests and limits are based on the initial baseline for a NodeJS service. Further runtime behavior analysis would be done to refine those numbers.
2. I would configure Liveness and Readiness probes as HTTP probes and implement liveness and readiness response endpoints in the service.
3. For autoscaling I would research using Kubernetes Horizontal Pod Autoscaler, but currently don't have much detailed experience with it.
4. Cluster nodes would need enough CPU and memory to handle the given estimated workload. In this case, a minimal node configuration 4GB RAM / 2 CPU cores, should handle the given workload in a 3 node cluster. Three nodes, 4GB RAM/2 CPU per node. This configuration would also provide node fault tolerance.