Final Project - Ticketing System - Zeyu Li

Introduction

The project implements a streamlined ticket-purchasing application using Spring Boot, Redis, RabbitMQ, DynamoDB, and MySQL. Testing will focus on highly concurrent purchase requests, pushing the system to its limits with intentional seat-conflict scenarios to verify correctness under load.

A fixed set of Venues and Events is designed for the simulation. However, the structure is designed so that when a user selects a different venue or event, the available Zones, Rows, and Seats update accordingly, mimicking the real-world flow of browsing and choosing.

A CQRS format is utilized in this project, a NoSQL database that quickly saves the information to enable quick writing to potentially increase ticket purchasing speed. And a relational SQL database that provides query availability for backend or user queries. DynamoDB will be used for the NoSQL database, and MySQL is being used for the SQL database.

The usage of Redis cache is utilized to save the status of all seats and the zone's status. The current usage is to provide a fast reject when a seat is taken without the need to visit the database to provide a faster response and reject

RabbitMQ is being used for decoupling the API service and the persistence service, and at the same time, through the exchanger, RabbitMQ divides up two queues, which allows a construct of 2 consumers to consume from the queue separately to achieve the CQRS structure.

Project Structure

I have designed the project to run on Docker containers and also to deploy on AWS. But the usage of the deployed DynamoDB table has been denied my permission since it requires IAM authentication. To describe the overall structure, there are two different images created.

To describe the overall structure, I built two distinct Docker images. The first image includes two REST controllers, one for ticket creation and one for ticket queries. The image includes Spring Boot bootstrap, the web API layer, the service layer that calls Redis, and the RabbitMQ producer that publishes ticket events.

The second image contains the RabbitMQ consumers for both SQL and NoSQL pipelines, the MySQL and DynamoDB DAO implementations, and all persistence configuration needed to save tickets into each data store.

Locally, these images are staged via Docker Compose to bring up Redis, RabbitMQ, MySQL, and a local DynamoDB. On AWS, all parts are running on separate EC2 instances. Server Instance runs on one server, RabbitMQ and RabbitConsumer run on another server. Redis and

DynamoDB are also running on 2 separate instances since AWS Redis and DynamoDB services are not allowed to be used.

Data Model

The data model has the following main models and those are being used in my design to simulate a real ticket purchasing program backend:

- **Venue**: venueId, List of Zones
    - The List of zones is different according to different Venues, for future proofing
- **Event**: eventId, venueId, name, type, date
    - Basic Information of events, combined with VenueId to pin to a specific event.
- **Zone**: zoneId, ticketPrice, rowCount, colCount
    - The price in my simulation is assigned according to zones.
- **TicketCreation**: ticketId, venueId, eventId, zoneId, row, column, status, and createdOn timestamp
    - The write-model DTO carries the object that is generated from the TicketCreation, which will be written to the NoSQL database
- **TicketInfo**: ticketId, venueId, eventId, zoneId, row, column, and createdOn timestamp
    - The read model used by query APIs. And also, the structure is saved to the SQL database.

Component Design

Redis Layer

In Redis, a bitmap, one bit per seat, together with a zone counter and a per-row counter, is used to provide the fastest possible rejection when there is a duplicate ticket purchase. When a POST request comes in, three layers of checks fire in sequence. Read the zone counter to see if the entire zone is sold out, then we read the row counter to see if that specific row is full, and finally, we inspect the bitmap bit for the exact seat. Enabling O(1) occupancy checks and immediate rejection when sold out.

Since there are counters and bits involved, and the thread will need to change the information in Redis, a race condition needs to be considered. I found an interesting way that uses a Lua script to ensure the atomicity by performing a get bit, decrement on counters, and set bit of the Bitmap in the same action, and returns distinct codes for success (0), duplicate (1), zone-full (2) or row-full (3).

Because Redis runs Lua scripts in its single-threaded loop, each script request is serialized, which means only one client can execute the script at a time, which means it performs like a built-in synchronized block without any external lock. A releaseSeat script performs the set bit back to 0 and increments the counters to roll back a reservation if later processing fails. This Lua approach achieves the fastest, simplest concurrency control for seat allocation under extreme load.

RabbitMQ Layer

In RabbitMQ, a dedicated exchange is used between the ticket service and the persistence consumer service, the MQ decouples message production from consumption. When a seat is reserved, the service publishes a URL query to that Thicket exchange, and the broker routes it into two durable queues. One for the SQL pipeline and one for the NoSQL pipeline, based on a shared routing key.

Each queue buffers messages until the consumer starts to pull and consume them, enabling an asynchronous persistence service. Consumers are configured with manual acknowledgments and tuned prefetch limits so that each message is only removed from the queue after a successful write.

On failure, they explicitly NACK(not ACK) to requeue, ensuring one more attempt at persistency. By catching duplicate-write errors or using conditional writes on the downstream database, each consumer achieves idempotent processing before sending its ACK. The use of RabbitMQ allows consumers to get the Ticket purchase response as soon as the Redis confirms that the seat is empty and changed to full to prevent double booking. And also has a separate pool of thread to make purchase messages persistent to both SQL and NoSQL databases.

Service & Controller

There are two different endpoints for different services in my project: tickets and get. The tickets endpoint is in charge of all actions that are DynamoDB related, and the get/ endpoint is used for everything that is MySQL related, which are query requests.

For Tickers purchase, it goes through the HTTP layer. If Redis returns success with no duplicated seat, it returns ok and proceeds to process ticket information to MQ and returns to the user the ticket information with the HTTP code 201. If enqueue fails, the Redis space will be released to allow further purchases. Get endpoints, bypass the messaging layer, and execute synchronous SQL queries to fetch real-time counts and revenue, ensuring up-to-date data. This clear separation of fast, non-blocking writes with maximum capability to support complex queries.

Confituration And Testing

Here are the key concurrency-related and pool settings which is being used in the testing process

MySQL (HikariCP) Connection Pool

- Minimum Idle Connections: 400
- Maximum Pool Size: 2000

Redis (Lettuce) Connection Pool

- Max Active Connections: 8
- Max Idle Connections: 8
- Min Idle Connections: 0
- Max Wait Time: infinite

RabbitMQ Listener Container

- Initial Concurrent Consumers: 100
- Maximum Concurrent Consumers: 300
- Prefetch Count: 300 messages per consumer

DynamoDB HTTP Client Pool

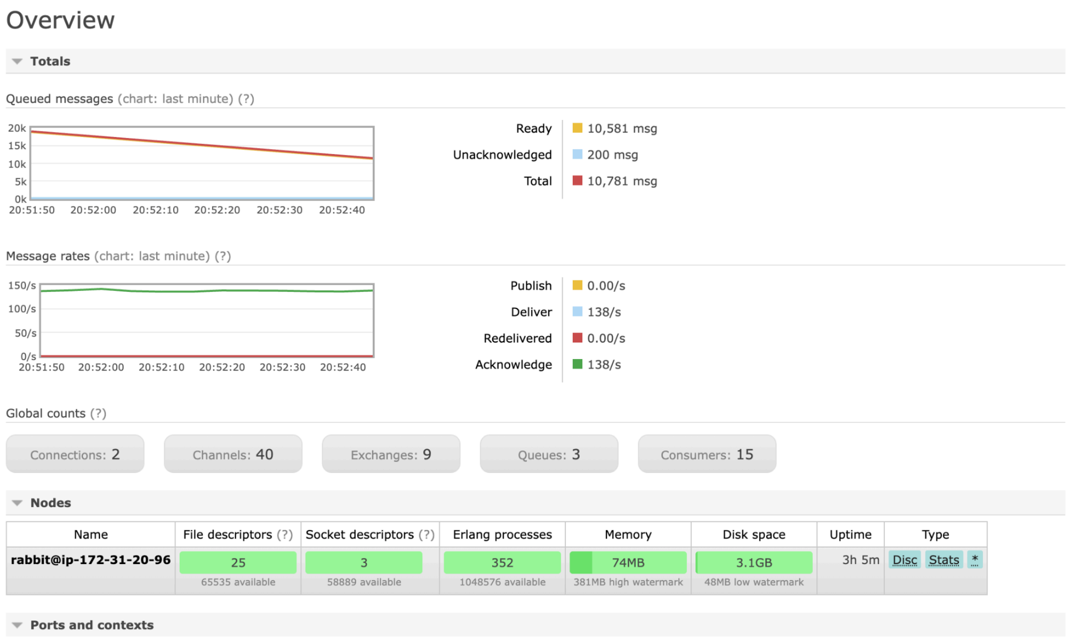
- Max Connections: 3000
- Connection Timeout: 10 seconds
- Socket (Read/Write) Timeout: 30 seconds

JMeter Load Test

- Client Threads: 1000 concurrent threads

Testing

In this Test, I want to test if the use of Redis could block the duplicate results without sacrificing the throughput and efficiency. The load test is performed by using JMeter. I wanted to do the test on AWS, which sets up an EC2 for a single instance, including DynamoDB through a Docker container and Redis. During the test, there was a problem that the RabbitMQ connection was not adjusting when I adjusted the settings, and after spending quite a long time and seeing no improvements, I changed to local testing.



Overview

Totals

Queued messages (chart: last minute) (?)

Ready	10,581 msg
Unacknowledged	200 msg
Total	10,781 msg

Message rates (chart: last minute) (?)

Publish	0.00/s
Deliver	138/s
Redelivered	0.00/s
Acknowledge	138/s

Global counts (?)

Connections: 2 | Channels: 40 | Exchanges: 9 | Queues: 3 | Consumers: 15

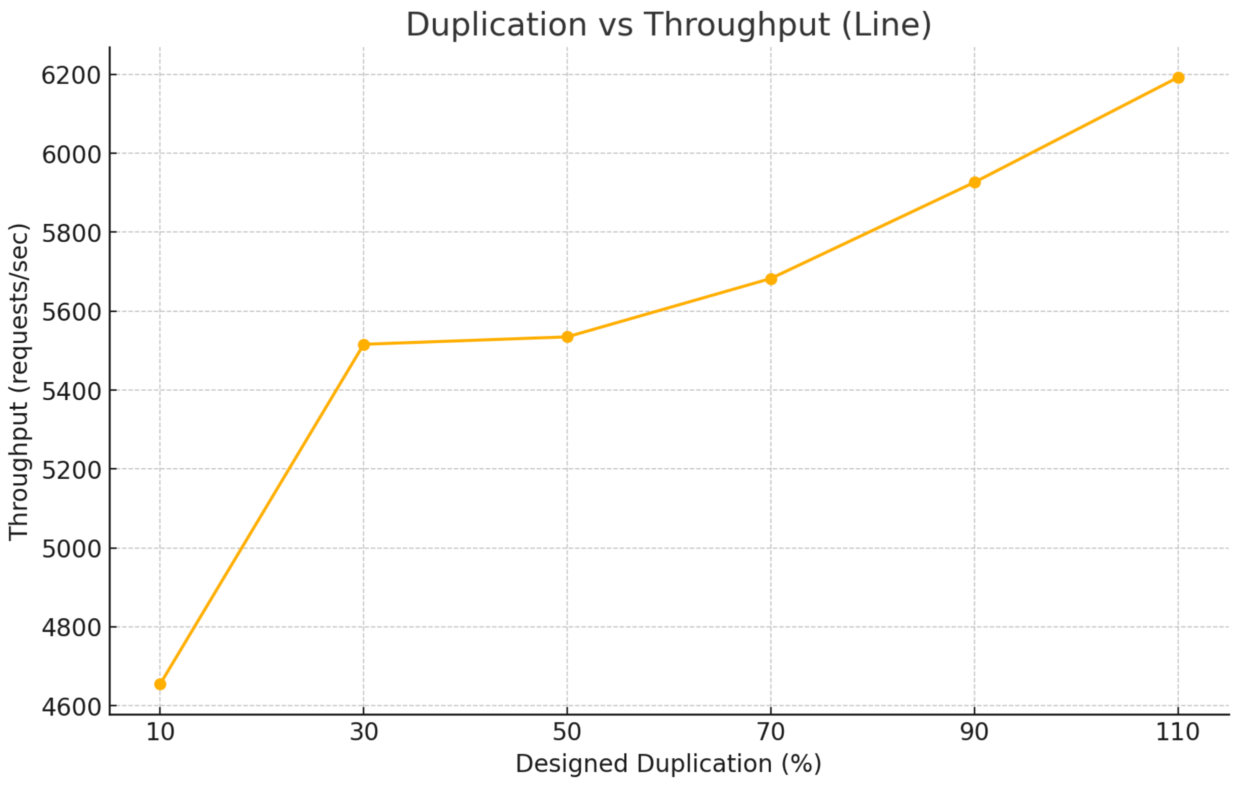
Nodes

Name	File descriptors (?)	Socket descriptors (?)	Erlang processes	Memory	Disk space	Uptime	Type
rabbit@ip-172-31-20-96	25 65535 available	3 58889 available	352 1048576 available	74MB 381MB high watermark	3.1GB 48MB low watermark	3h 5m	Disc Stats *

Ports and contexts

Setup: 1,000 concurrent threads, ramped from 10% to 110% of target load, all targeting the same zone/row combinations to force conflict scenarios.

Designed Duplication	Label	samples	99%	error	throughput	received	sent
10%	Post/Ticket	85800	1106	9.09%	4654.947917	1367.969513	1189.285278
30%	Post/Ticket	101400	584	23.08%	5515.965838	1522.75864	1409.26665
50%	Post/Ticket	117000	468	33.33%	5534.79351	1456.616168	1414.071277
70%	Post/Ticket	132600	534	41.18%	5682.69478	1439.735413	1451.862048
90%	Post/Ticket	148200	476	47.37%	5926.340625	1455.504334	1514.110502
110%	Post/Ticket	163800	533	52.38%	6192.114316	1481.911241	1582.009617



Future Work

It feels like this project is not the end of the journey, but the start of a potentially larger platform. There are a few areas I want to improve on, mostly around our use of Redis.

I would like to spin up a second Redis node, since our current instance (and its single-threaded Lua script) limits concurrent capacity. By sharding across two Redis servers and hashing by zone number, we could run two Lua scripts in parallel and potentially double our reservation throughput.

I would also cache newly created ticket details in Redis to give users a fast path for viewing their tickets right after purchase. That cache would live on a separate, read-only Redis instance, so heavy write traffic on the sharded masters never slows down lookups.

Finally, I'd introduce a thin, transparent routing layer in the ticketing system that forwards each request to the correct shard based on zone range. We could use a dynamic waiting time before the start of ticket sales, to process the more expensive tickets first, and then start to release other zones' purchases.

Summarization

In summary, this ticketing system combines a lightweight API layer with an atomic, in-memory seat reservation mechanism, an asynchronous message processing, and dual persistence stores to deliver both high throughput and strong consistency under contention. Redis with Lua scripts enforces single-seat guarantees at O(1) cost, RabbitMQ decouples writing from the user path, and DynamoDB/MySQL implement a CQRS-style write vs read model. By careful tuning the connection pools and consumer threads, the platform scales to thousand of concurrent users, while JMeter tests confirm that duplicate requests are rejected without throttling overall throughput. Redis sharding, a read-only cache, and a transparent routing layer will be some of the future improvements. Those improvements will provide a higher concurrency level, distribute the access stress of a single server, and provide a better service to the consumers.