Use Redis to Avoid Ticket Collision - Ticketing Platform

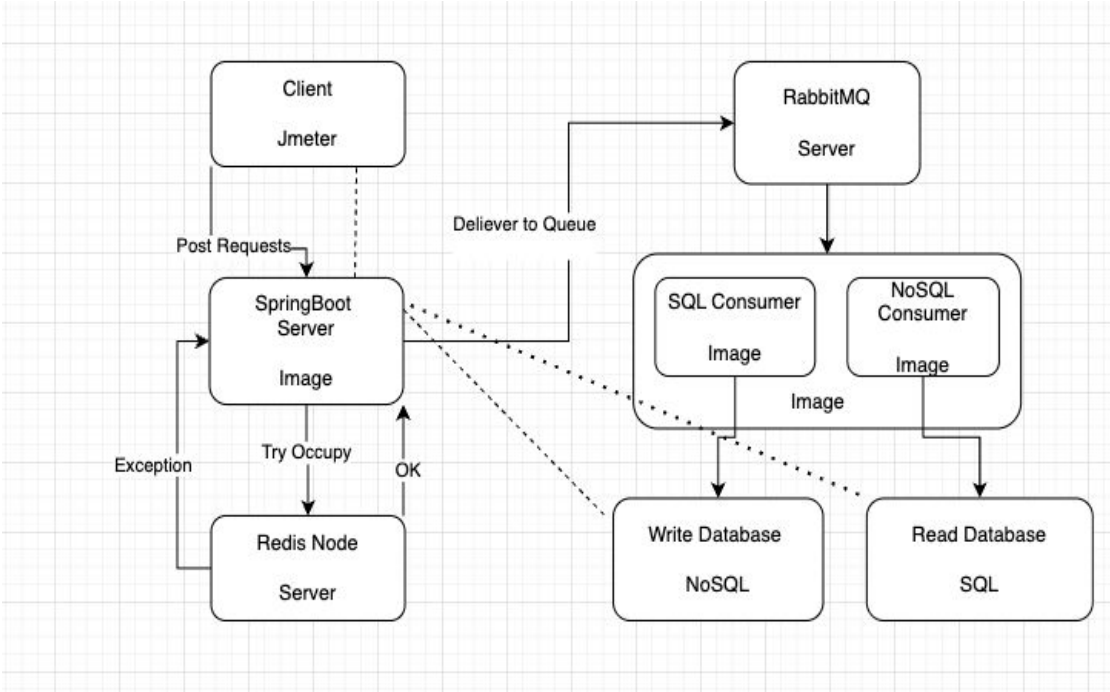
CS6650 Final Project – Zeyu Li

Project Design

- Simulate Ticket Purchasing Platform that users wait until start time to purchase the ticket directly
- Concurrent requests all together in the beginning phase, including duplication scenarios
- 100 Zones * A-Z Row * 30 seat
- Structure
  - **Venue**: venueId, **List of Zones**
  - **Event**: eventId, venueId, name, type, date
  - **Zone**: zoneId, **ticketPrice**, rowCount, colCount
  - **TicketInfo**: ticketId, venueId, eventId, zoneId, col, row, createdOn
  - **TicketCreation**: **status**

Project Structure

- Client: Jmeter
- Server: Java Spring Boot
  - Two Controllers
- Cache: Redis
- MQ: RabbitMQ
- NoSQL: Dynamodb
- SQL: MySQL



CQRS - Tickets/ and Query/

- Commands mutate state (POST /ticket → Redis → MQ → write store)
- Queries read state (GET /get/xxx → direct SQL and MySQL access)

Write Model (Commands)

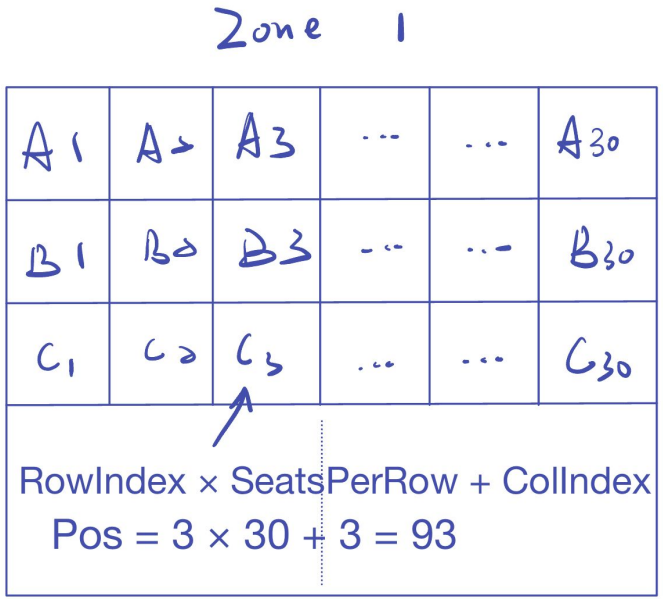
- Redis + Lua for atomic seat reservation
- RabbitMQ to enqueue reservation events
- Designed DynamoDB for fast and idempotent ticket writes

Read Model (Queries)

- MySQL for real-time counts, revenue, and reporting
- Direct JDBC queries bypass the message queue

Redis - Zone → Row → Seat

- **3 Step Check to Provide Feedback if Duplicate Seat Purchase**
- **Bitmap: 1 bit per seat for O(1) occupancy checks**
- **Zone Counter: tracks total available seats per zone**
  - **event:{eventId}:zone:{zoneId}:remainingZoneSeats**
- **Row Counter: tracks available seats per row**
  - **event:{eventId}:zone:{zoneId}:row:{rowIndex}:remainingSeats**



Redis - Lua Scripts

– occupySeat.lua: GETBIT, DECR counters, SETBIT in one atomic script

– releaseSeat.lua: SETBIT=0 and INCR counters for rollback

– Single-threaded executions

- Atomicity without external locks
- Only one script at the same time

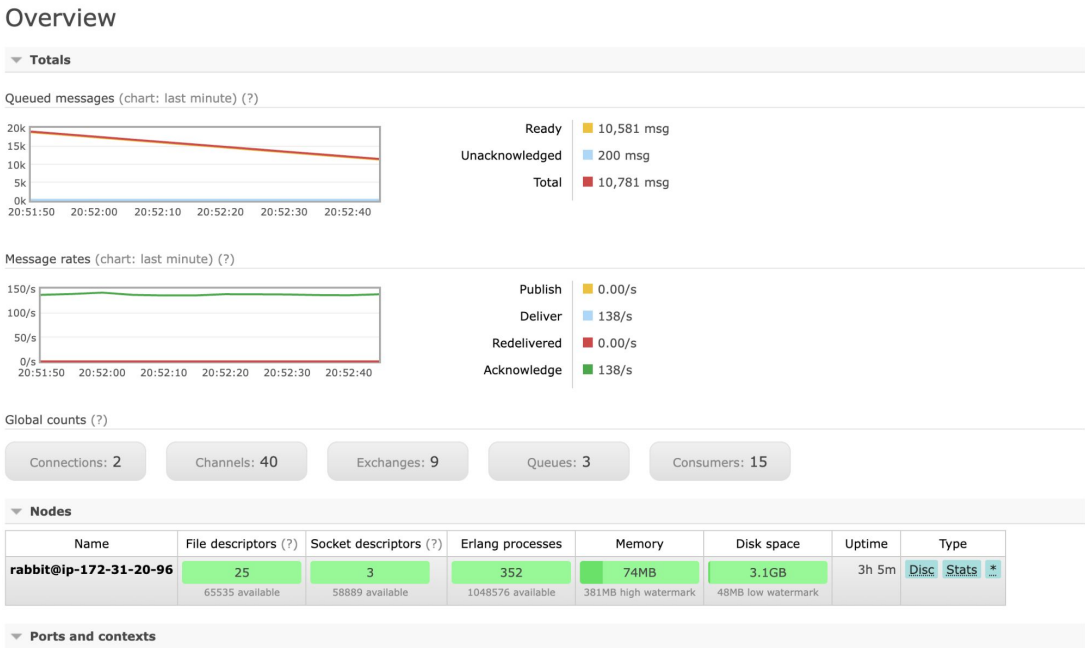
– Lettuce Connection Pool: max-active=8 to manage client concurrency

Deployment Attempt

EC2 deployment

- DynamoDB - IAM requirements
- RabbitMQ throughput problem

Local Docker Image Deployment



Local Test Thread Setting

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

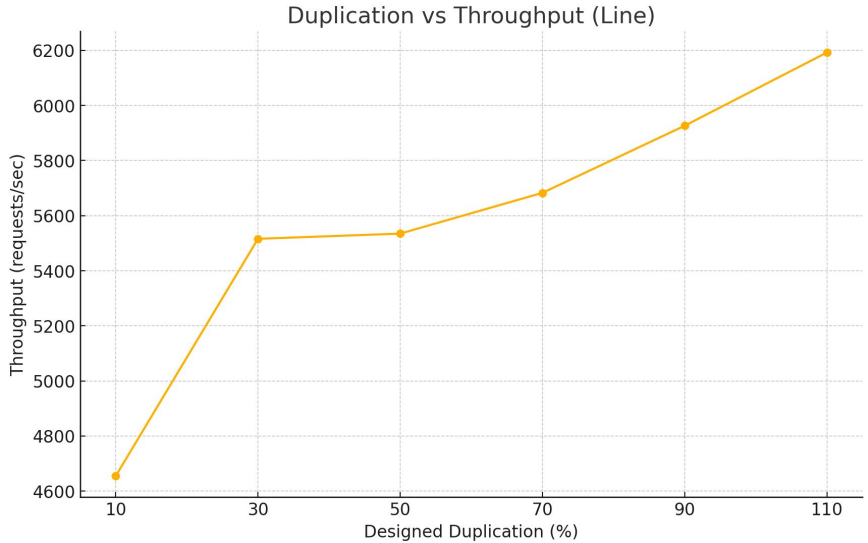
- Max Connections: 3000
- Connection Timeout: 10 seconds
- Socket (Read/Write) Timeout: 30 seconds

JMeter Load Test

- Client Threads: 1000 concurrent threads

Test Result - Increase duplication

At the same setting of threads and environment, increase the duplication rate in the dataset to mimic more buyers for the same ticket.



Designed Duplication	Label	samples	99%	error	throughput	received	sent
10%	Post/Ticket	85800	1106	9.09%	4654	1367.969513	1189.285278
30%	Post/Ticket	101400	584	23.08%	5515	1522.75864	1409.26665
50%	Post/Ticket	117000	468	33.33%	5534	1456.616168	1414.071277
70%	Post/Ticket	132600	534	41.18%	5682	1439.735413	1451.862048
90%	Post/Ticket	148200	476	47.37%	5926	1455.504334	1514.110502
110%	Post/Ticket	163800	533	52.38%	6192	1481.911241	1582.009617

Future Work

- Redis Sharding:
  - add a second node and hash by zoneId to run two Lua scripts in parallel
  - higher reservation throughput
- Read-Only Cache:
  - cache new ticket details in a separate read-only Redis instance for instant “My Tickets” lookups without impacting write load
- Transparent Routing & Holds:
  - Support dynamic hold durations
  - Support a transparent tiered processing speed model

Thank you!