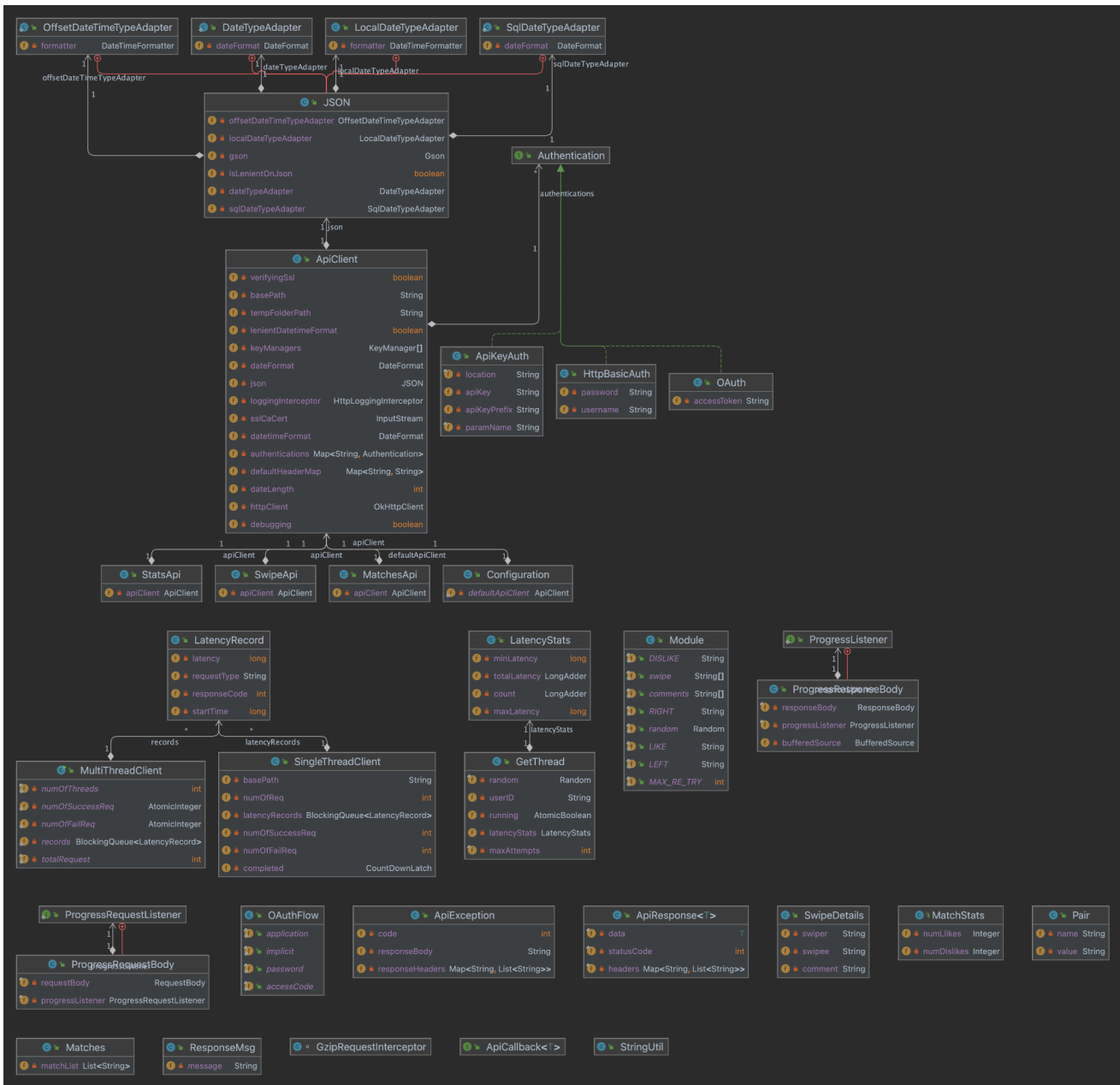


1. Git repo: <https://github.com/Oliver1024/DistributedSystemDatingApp/tree/main/assignment3>
2. Classes diagram, solution architecture, database design and deployment technology:
Client:



Consumer:

ConsumerRunnable		
f	batchSize	int
f	swipeBatch	List<Document>
f	database	MongoDatabase
f	executor	ThreadPoolExecutor
f	processedMessageCount	AtomicInteger
f	connection	Connection
f	threadPoolSize	int
m	processMatches(ConcurrentLinkedQueue<Document>)	void
m	updateStatsCollection(MongoCollection<Document>, Map<Integer, Integer>)	void
m	run()	void
m	flushBatch(List<Document>)	void

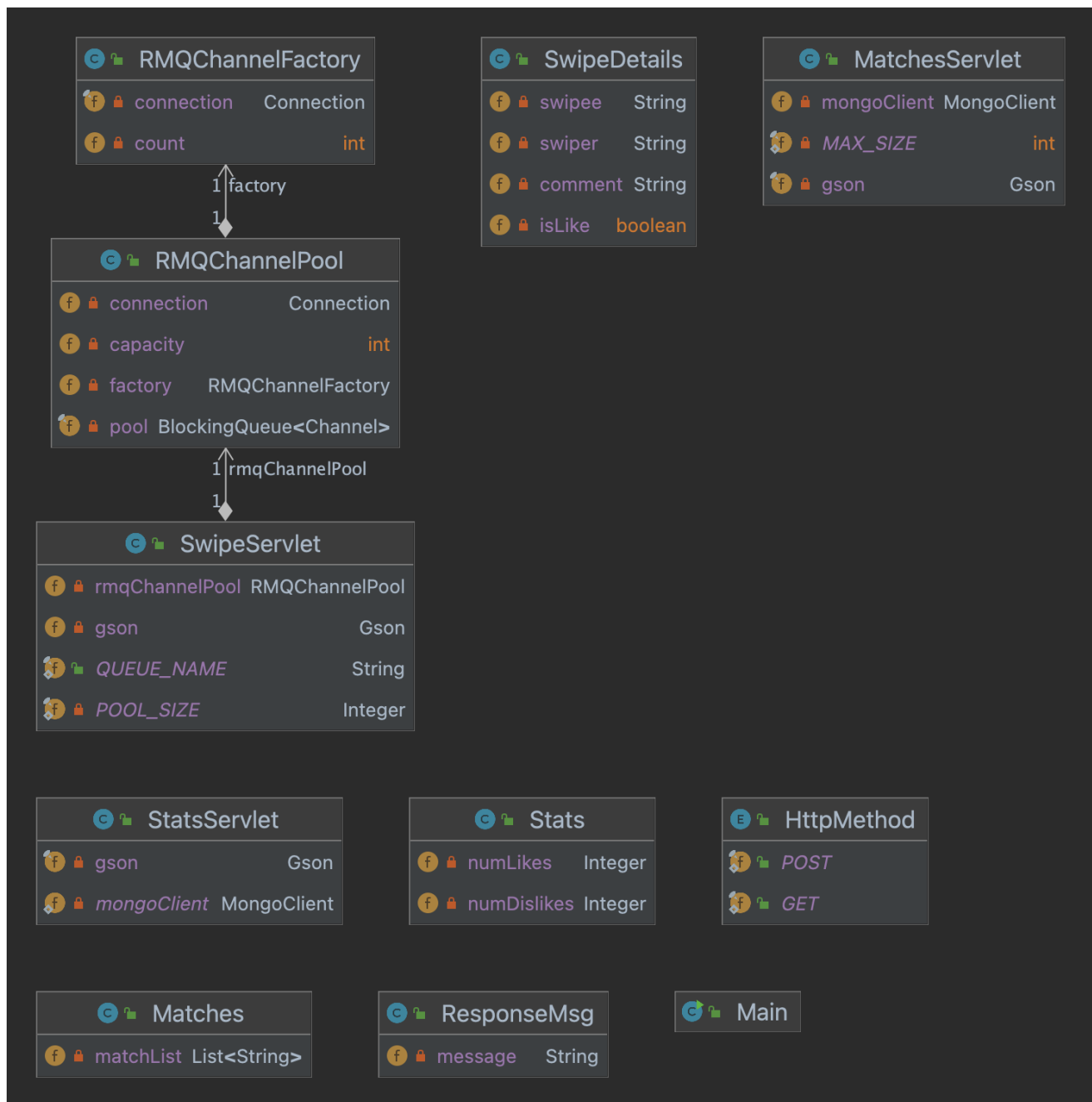
SwipeDetails		
f	swiper	Integer
f	isLike	boolean
f	swipee	Integer
f	comment	String
m	getComment()	String
m	getSwipee()	Integer
m	getSwiper()	Integer
m	setSwiper(Integer)	void
m	getLike()	boolean
m	setLike(boolean)	void

SwipeRecord		
f	swiper	Integer
f	isLike	boolean
f	likeOrDislikeMap	Map<Integer, int[]>
f	listSwipeeRight	Map<Integer, Set<Integer>>
f	listSwiperRight	Map<Integer, Set<Integer>>
f	swipee	Integer
m	addToLikeMap(Integer, Integer, boolean)	void

MultiThreadConsumer		
f	NUM_PER_THREADS	Integer
f	mongoClient	MongoClient
m	collectionExists(MongoDatabase, String)	boolean
m	main(String[])	void

Constant		
f	QUEUE_NAME	String
f	NUM_PER_THREADS	int
f	HOST_NAME	String

Server:



Solution Architecture:

Client:

GetThread class implements the Runnable interface, with its primary responsibility being to send GET requests to the server. It randomly selects a userID from 1 to 50,000 and then sends a GET request to either the MatchesApi or StatsApi. It records the latency of each request and updates the LatencyStats object.

MultiThreadClient class creates an array of **SingleThreadClient** instances and launches them concurrently. It also starts a **GetThread** instance to perform GET requests concurrently with the **SingleThreadClient** instances. Once all **SingleThreadClient** instances have finished, the **GetThread** is stopped, and the program calculates and prints the total throughput, successful requests, failed requests, and latency statistics.

SingleThreadClient class, which also implements the **Runnable** interface, is responsible for sending POST requests to the server. Each instance of the class sends a predefined number of POST requests, with random data generated using the **Module** class. It calculates the latency of each request and adds it to the **latencyRecords** **BlockingQueue**. The class also tracks the number of successful and failed requests.

Servlet/tomcat server:

MatchesServlet: Handles GET requests to fetch the list of matches for a specific user. It connects to the MongoDB database, queries the "matches" collection to retrieve **matchedSwipees**, and sends the response as a JSON object.

StatsServlet: Handles GET requests to fetch the number of likes and dislikes for a specific user. It connects to the MongoDB database, queries the "stats" collection to retrieve **numLikes** and **numDislikes**, and sends the response as a JSON object.

SwipeServlet: Handles POST requests to submit a new swipe action (left or right) performed by a user. The servlet validates the input data, creates a **SwipeDetails** object, and sends it as a message to the RabbitMQ messaging system through the "swipe_queue".

RabbitMQ Message Broker: A message broker that receives swipe data from a producer and distributes it to the consumer and the consumer then writing the data to MongoDB in EC2 instance.

MultiThreadConsumer: A multithreaded consumer application that processes swipe data, processes matches, and updates the MongoDB database.

MongoDB: A NoSQL database that stores and manages the swipe data. Like stored the number of likes and dislikes and also for the swiper and swipee matches information.

Database Design:

matches collection: Stores mutual matches between users.

_id: user ID

matchedSwipees: list of mutually matched user IDs

admin.matches

Documents

Aggregations

Schema

Explain Plan

Indexes

Validation

Filter



Type a query: { field: 'value' }

ADD DATA

EXPORT COLLECTION

```
{
  "_id": 21894,
  "matchedSwipees": [
    16689,
    17933,
    5733,
    22308,
    22768,
    22186,
    47127,
    7596,
    41957,
    40982,
    14741
  ]
}
```

```
{
  "_id": 25991,
  "matchedSwipees": [
    12312,
    21855,
    31029,
    26574,
    25047,
    20008,
    18619,
    49098,
    14635
  ]
}
```

stats collection: Stores the number of likes and dislikes for each user.

_id: user ID

numLikes: number of likes for the user

numDislikes: number of dislikes for the user

admin.stats

Documents

Aggregations

Schema

Explain Plan

Indexes

Validation

Filter



Type a query: { field: 'value' }



ADD DATA



EXPORT COLLECTION

```
▼ {  
  "_id": 4100,  
  "numDislikes": 4,  
  "numLikes": 3  
}
```

```
▼ {  
  "_id": 28426,  
  "numDislikes": 4,  
  "numLikes": 1  
}
```

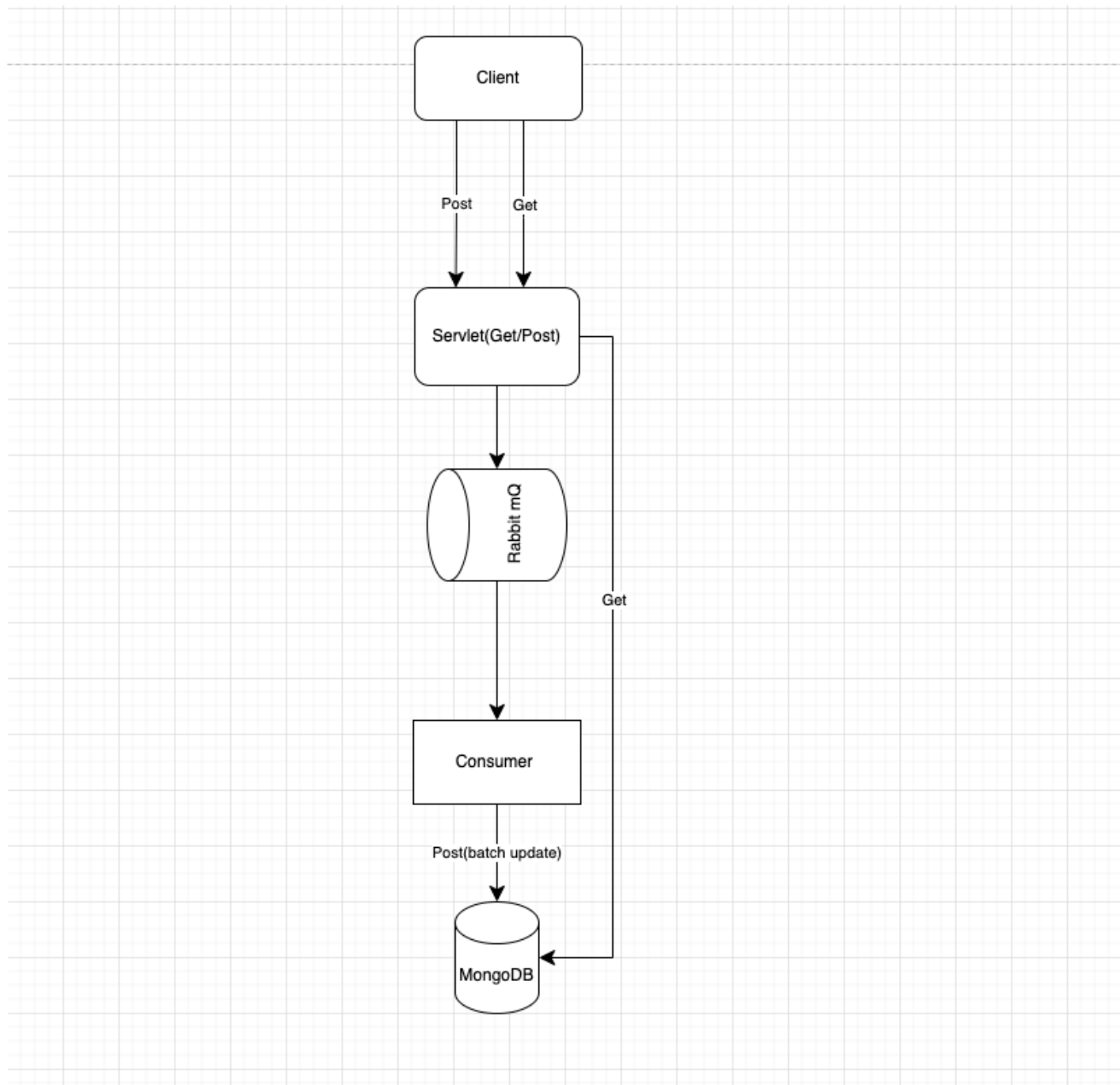
```
▼ {  
  "_id": 10763,  
  "numDislikes": 0,  
  "numLikes": 7  
}
```

AWS Services used:

EC2: To host the RabbitMQ message broker and the MultiThreadConsumer application and also tomcat server is running in EC2 instances. Also instilled MongoDB in EC2 instance.

VPC & Security Groups: To define network access and security rules for the RabbitMQ and MultiThreadConsumer instances, Tomcat serverMongoDB instance.

And here's my diagram for the solution architecture.



3. Compare:

A3 screenhosts:

```
CalculateLatency.MultiThreadClient x
===== GET requests results =====

Min latency: 13 ms
Mean latency: 46.14604810996563 ms
Max latency: 2826 ms

===== POST requests results =====

Total post throughput: 3289 req/s
Number of successful requests: 500000
Number of fail requests: 0
Mean latency time: 45 ms
Max latency time: 2910 ms
Min latency time: 12 ms

Process finished with exit code 0
```


Overview

▼ Totals

Queued messages **last ten minutes** ?



Ready

0

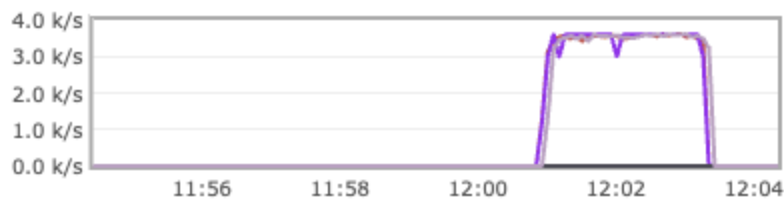
Unacked

2,000

Total

2,000

Message rates **last ten minutes** ?



Publish

0.00/s

Publisher
confirm

0.00/s

Deliver
(manual
ack)

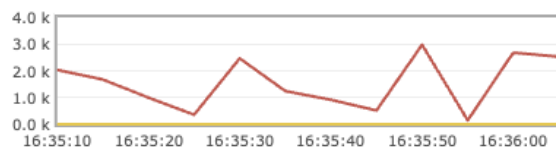
0.00/s

Global counts ?

Overview

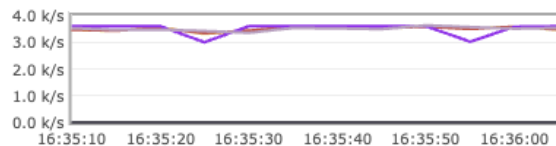
▼ Totals

Queued messages **last minute** ?



Ready	0
Unacked	2,211
Total	2,211

Message rates **last minute** ?



Publish	3,465/s	Deliver (auto ack)	0.00/s
Publisher confirm	0.00/s	Consumer ack	3,600/s
Deliver (manual ack)	3,477/s	Redelivered	0.00/s

Global counts ?

Connections: **31** Channels: **60** Exchanges: **7** Queues: **3** Consumers: **30**

A2 screenhosts:(From my A2 report)

```
src
├── main
│   └── java
│       └── org.example
└── MultiThreadClient.java
    5
    6 public class MultiThreadClient {
    7     private static final int numOfThreads = 110;
    8     private static AtomicInteger numOfSuccessReq = new AtomicInteger(0);

    /Library/Java/JavaVirtualMachines/jdk-11.0.11.jdk/Contents/Home/bin/java ...
    Number of successful requests: 500000
    Number of fail requests: 0
    walltime: 158 seconds
    Total throughput: 3164 req/s

    Process finished with exit code 0
```

Queue swipe_queue_1

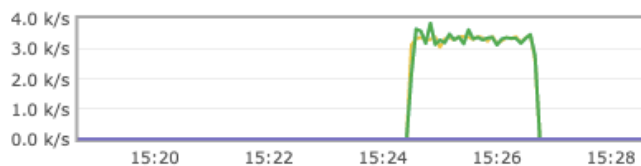
▼ Overview

Queued messages [last ten minutes](#) ?



Ready	0
Unacked	0
Total	0

Message rates [last ten minutes](#) ?



Publish	0.00/s
Deliver (manual ack)	0.00/s
Deliver (auto ack)	0.00/s

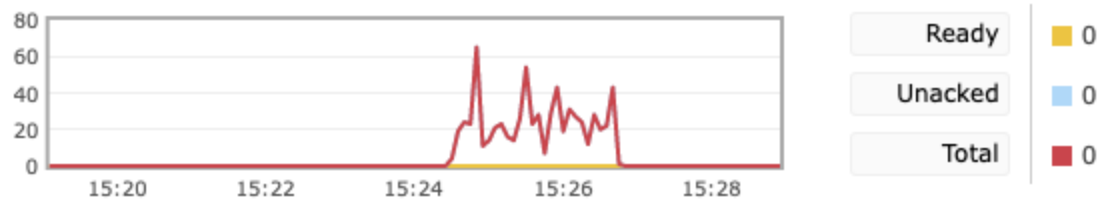
Consumer ack
Redelivered
Get (manual ack)

Details

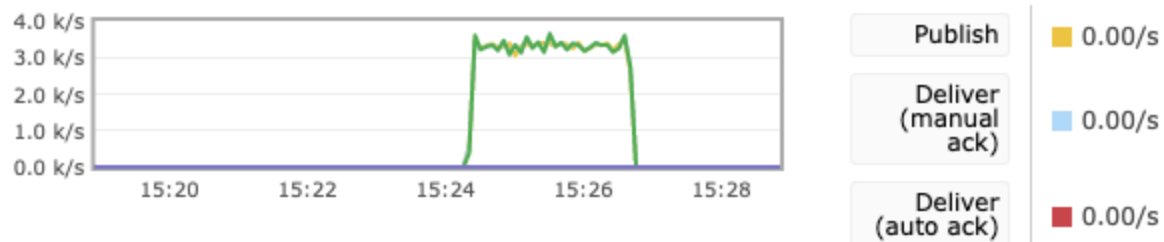
Queue swipe_queue_2

Overview

Queued messages **last ten minutes** ?



Message rates **last ten minutes** ?



Details

Features	durable: <u>true</u>	State	<u>idle</u>	
Policy		Consumers	20	
Operator policy		Consumer capacity ?	100%	M
Effective policy definition				

From the comparison, we can observe that the throughput for A2 and A3 are quite close. Similarly, the RabbitMQ screenshots show comparable results for the publish and consumer rate ack, with A2 around 3300 and A3 approximately 3250. For A2, I only printed the throughput and didn't record the latency. However, considering that the A3 latency is quite small and the throughput is very close to A2, it is reasonable to assume that the latencies for both A2 and A3 are nearly the same.

To achieve this level of performance, I conducted multiple trials with different configurations and varying numbers of threads for the client, Tomcat server, consumer, and database connections for the consumer. Initially, without batch writing, the data writing to the database was slow. However, after implementing

batch writing and adjusting the batch size, I was able to optimize the performance and achieve the current results.