CS6650 Spring 2023 Assignment 3 Submission Report

Name: Chin Shiang Jin

NEU ID: 002798503

NEU Email: chin.shi@northeastern.edu

Contents

URL of Git Repository

Main Repository

https://github.com/sjchin88/bsds-assignment3

Server (javax.servlet)

https://github.com/sjchin88/bsds-assignment3/tree/main/Server

Client

https://github.com/sjchin88/bsds-assignment3/tree/main/Client

Consumers program

https://github.com/sjchin88/bsds-assignment3/tree/main/Consumers

Solution architecture & Deployment Topologies

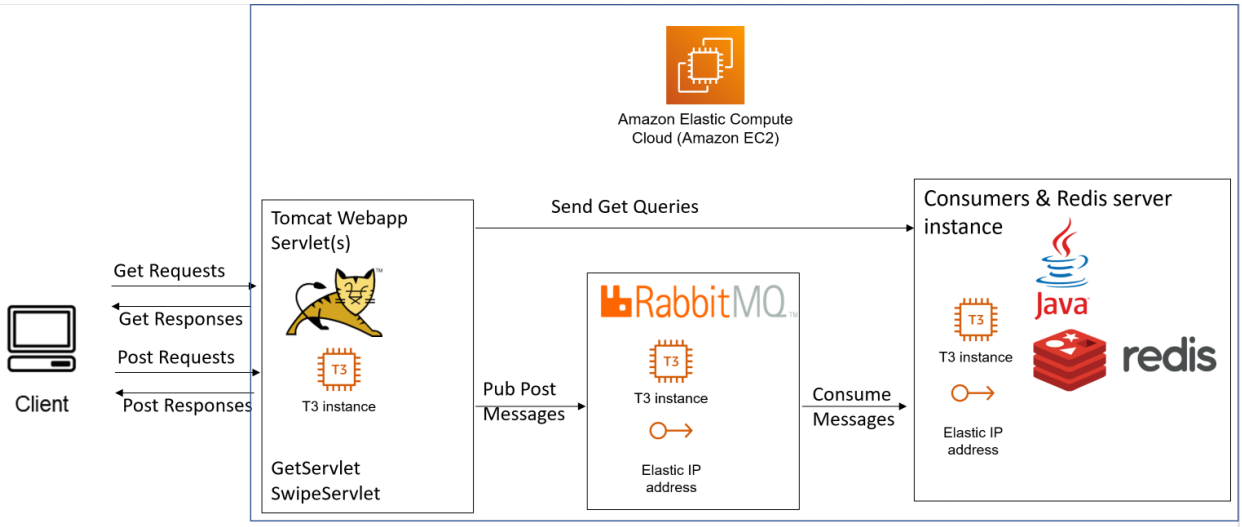

Figure 1: Solution architecture and Deployment Topologies

Figure 1 shows the overall solution architecture and deployment topologies for the design.

For the main server communicating with the client(s), an EC2 T3.micro instance is used to host the Tomcat server where the servlets are being deployed. The GetServlet serves all traffic for get requests (/stats/ and /matches/) and queries the SwipeData database (the Redis server) for the results. The SwipeServlet serves all traffic for the post requests as per previous assignments. The post requests will be processed, with key data sent to the RabbitMQ server (the TempStore).

For the TempStore, an EC2 T3.small instance is used to host the RabbitMQ server. The TempStore is ensured to be persistent by declaring all the exchanges, queues, and messages to be persistent. RabbitMQ is chosen because

- It is free and
- incur minimum learning curve since it was already been used in assignment 2.

For the consumer and database, an EC2 T3.small instance is used to host the java programs and the Redis server. The SwipeRecorder java program will consume the messages of the swiper:swipe pair from the RabbitMQ, and write it into the Redis database. The CountRecorder java program will consume the messages of the likes/dislikes by the swiper, and increment the corresponding count accordingly into the Redis database. Redis database is chosen because

- It is free (unlike AWS DynamoDB which still charges above free quotas)
- Minimum effort is required to set up the connections and the database using the Java Lettuce library
- Its storing scheme is simple enough for our use case. It is easy to write the client code to write to the database and queries from the database.
- Minimum latency incurred for the write operation as the Redis server is installed in the instance where the java consumer programs are run.

Database Design Descriptions

Redis is an in-memory database that persists on disk. The data model is key-value, but many different kinds of values are supported. For our use case scenario as seen thus far up to assignment 3, only two datatypes are needed for the values: string and sets(). With the swiper id as part of the key, the string value type will be used to store the integer representation of how many likes/dislikes the particular user has given. The sets() value type will be used to store all swipee ids where the user has swiped right on.

Redis does not need a formal schema to manage unstructured data. However, we can use the colon: inside the key as the namespace separator to group data together. For example in our use cases, to store the record of the likes count by the swiper together, we will use the following schema:
(key) Likes:${ userId } – (value) ${String representation of likes count integer}
Similarly, for the dislikes count, we will use
(key) Dislikes:${ userId } – (value) ${String representation of dislikes count integer}
and for the swipee id record, we will use
(key) Swiper:${ userId } – (value) set() containing all the swipee id the user has swiped right on.

For the likes and dislikes count addition event, we just need to call the INCR operation on the user's key, this will increment the integer value associated with the key by one and uses 0 as the initial value if the key doesn't exist yet. For the queries, we will use the GET to return the value of the key.

For the list of swipee IDs, SADD will be used to add a new swipee id to the list of swipee the user has swiped right on. This will also create the key with an empty set() first if the key does not yet exist. For queries, we can use SMEMBERS to return all members of the set, and SISMEMBER to check if another user is one of the swipee.

Best Performance and Comparison with Assignment 2

```
Test 4
Number of threads used: 350
Number of successful requests: 499800
Number of unsuccessful requests: 0
get thread exiting
Total run time (wall time) taken = 93761ms
Total Throughput in requests per second = 5332.707628971534
mean response time: 65.44000800319701
median response time: 63.0
99th percentile: 110.0
min response time: 10.0
max response time: 368.0
Get Request results:
mean response time: 68.53142857142856
median response time: 67.0
99th percentile: 112.98000000000002
min response time: 14.0
max response time: 126.0
```

Figure 2: Throughput statistic summary for the best run

Figure 2 shows the throughput statistic summary for the best run achieved. The best throughput for post requests is around 5333 per second, with a mean response time of 65.44 ms. For the get requests, the mean response time is about 68.53 ms which is about the same as the post requests.

Number of client threads used	200			350		
	HW2 Best	HW3 Best		HW2 Best	HW3 Run 1	
		Record	Variance(%) with HW2		Record	Variance(%) with HW2
Total run time (wall time) in ms	94621	95863	1.31%	95582	93761	-1.91%
Throughput per second	5,284.2	5215.8	-1.30%	5231.1	5,332.7	1.94%
mean response time (ms)	37.5	38.1	1.68%	66.6	65.4	-1.73%
median response time (ms)	35.0	37.0	5.71%	64.0	63.0	-1.56%
99th percentile (ms)	80.0	80.0	0.00%	117.0	110.0	-5.98%
min response time (ms)	10.0	8.0	-20.00%	10.0	10.0	0.00%
max response time (ms)	1,163.0	319.0	-72.57%	1210.0	368.0	-69.59%

Table 1: Explicit comparison with assignment 2 post results

Table 1 shows the comparison of the post requests statistic with the assignment 2 best results. It can be seen that for the same number of client threads used, the best performance achieved in assignment 3 is within +10% variance compared to assignment 2 best performance. Figure 3 shows the throughput comparisons between assignment 2 and assignment 3 for a different number of client threads used.

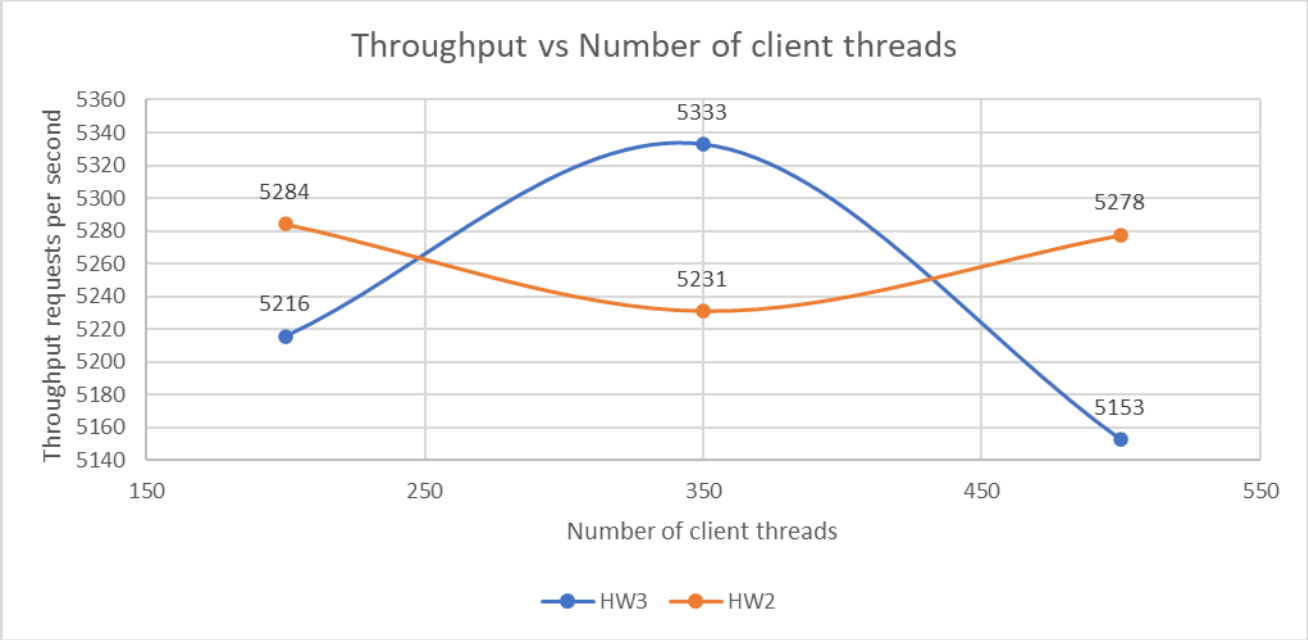

Figure 3: Throughput summary for different numbers of client threads between assignment 2 and assignment 3

The best throughput for this assignment was achieved using the following settings.

Setting for client-side:

- Using org.apache.hc.client5
- waiting for HTTP response asynchronously
- Actively manage the pool connection limit
- Use a single instance of HttpClient for all client threads.

Setting for server and RabbitMQ:

- Tomcat server max number of threads of 200
- 200 channel connections from the Tomcat server to RabbitMQ
- 20 threads each for both consumers' programs. As the queue size remains zero and flatline (see figure 4) this is unlikely to be the bottleneck here.
- The instance used for the server is T3.micro, and for RabbitMQ and consumers is T3.small.
    - Based on the stable profile seen on messages published and delivery rates on RabbitMQ (figure 4), and the CPU utilization not yet reached the maximum (figure 5), RabbitMQ is unlikely to be the bottleneck here.
    - Based on the CPU utilization report (figure 5), the server and consumer instances are unlikely to be at a bottleneck as well.

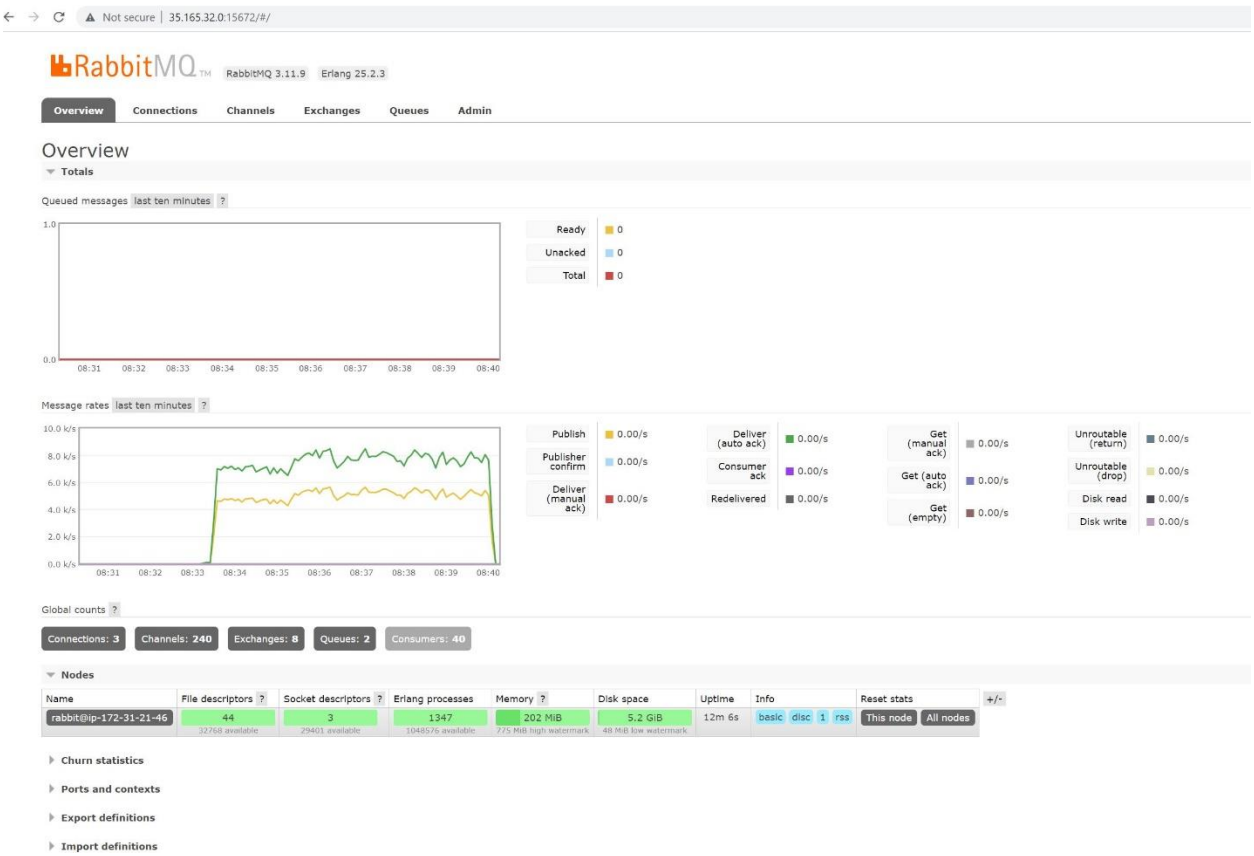

Figure 4: RabbitMQ overview

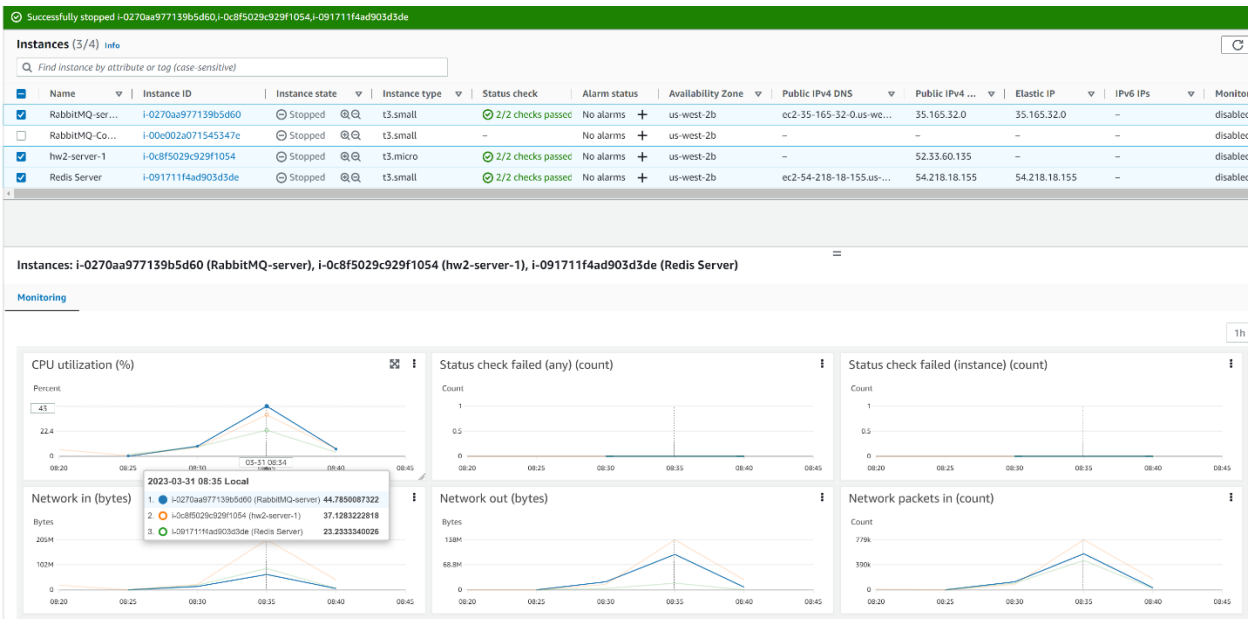

Figure 5: Instance CPU Utilization Report

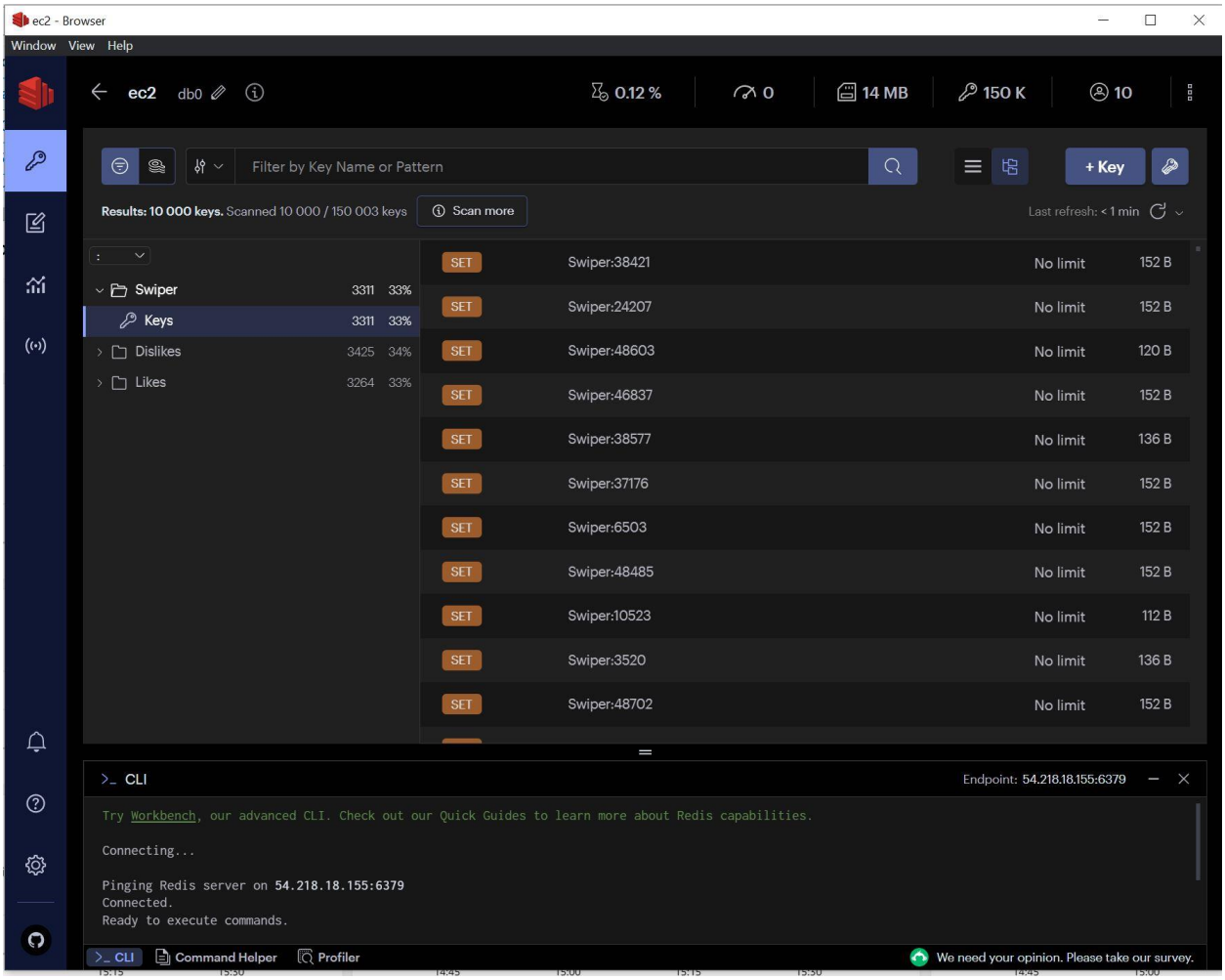

Figure 6: Overview of the Redis database state after all post requests