CS6650 FALL 2022: Assignment 3 – Adding Data Layer

Submitted by Srishti Hegde

GitHub Repo URL

The implementation of the server, client and consumer program can be found at
https://github.com/srishtih/CS6650Assignment3.git

Overview

The objective of this assignment was adding a data layer to the previously built ski server setup that wrote data received from incoming POST requests into a queue for further processing asynchronously.

In the previous assignment the data written into this queue was written into a HashMap, which was not persistent. Hence, in this assignment we implement the obvious next step, adding a database into the setup in place of the HashMap.

As far as database options, we were presented with 4: Redis (https://redis.io/), MySQL using AWS RDS, Postgres, or Aurora, DynamoDB and MongoDB. In this assignment, I've chosen to work with Redis for the sake of exploring this technology and due to its speed, reliability, and performance.

Database Design

When trying to setup Redis on an EC2 instance, the initial instance type was t2.micro. But the tests kept failing and when we migrated to a t2.medum, the tests passed and since we have used the same t2.medium type instance for hosting our Redis database.

Redis is an open source, in-memory, NoSQL key/value store that is primarily used as an application cache or quick-response database. As [per the assignment's ask, the database designed should enable queries like:

- "For skier N, how many days have they skied this season?"
- "For skier N, what are the vertical totals for each ski day?" (Calculate vertical as liftID*10)
- "For skier N, show me the lifts they rode on each ski day"
- "How many unique skiers visited resort X on day N?"

Key selection:

We use skier id as the central piece of data around which the key is built. In need of a unique identifier, we have decided to use a combination of skier id, season, time and day as the key as it should be technically impossible to have a collision with this key.

A couple of approaches were considered given these requirements. A brief overview of them is presented below and the thought process that went into it.

1. Direct translation of previous HashMap implementation: Creating a set for each skier id and adding the json string written into the remote queue by server.
   But for the purpose of making data more query-able, this has not been implemented.
2. Creating custom HashMap for queries needed: When the queries to be made from the database are limited and fixed, we could create sets of HashMaps with keys that are dependent on query requirements. For example, for query 1 : For skier N, how many days have they skied this season Here we could create a HashMap where the key is skier id: day id: season id and fields of skier id, day and season id. All HashMaps thus created can be added to some set on which querying might be done.

3. Building a set of HashMap of skier rides: Building on the ideas presented in the previous approach, we build a new key from the skier id, season, day and time.
   This HashMap stores all the values such as the resort id, lift id, season., day and time as field-values. Each of these HashMap entries created are added into a Redis set since they can be queried.
   Since wild cards cannot be used with HashMap keys, we can use SCAN with MATCH for filtering the keys of the HashMaps in the set.
   The implementation and trial of the possible queries with the current set up was limited without the usage of Redis stack's features.

   Another observation regarding the lift ride events was that the number of keys in the set might not always equal to the number of requests sent. This was due to the lift ride event being randomly generated, so there could be instances where events might be generated for the same time point for a skier which is practically not possible.

Performance Review

The system that we have now set up is that we have a client that sends a large number of requests, N (which replicates user generated POST requests (as though from end user)) to a remote server.

The server processes the requests to verify the values are valid and writes the data as a json string to a remote queue which send sit downstream for asynchronous processing. The remote queue we have used here is RabbitMQ. The data validation is done at server end, but it might be a good idea to do client-side validation.

The consumer code reads from a RabbitMQ queue and writes the data into the Redis database in the model we discussed above.

To test the performance of our setup, we can fine tune some of the aspects of this system, nalmely:

1. Server hosting EC2 instance type
2. RabbitMQ hosting EC2 instance type
3. Consumer and redis hosting EC2 instance type
4. Number of requests sent
5. Client being remote/local
6. Introduction of throttling with circuit breaker.

In terms of performance, our objective is to increase throughput and minimize queue length in RabbitMQ.

For all experimentation, we will be sending 500K requests from our local client.

The throughput and max queue length under different configurations are as follows:

				Threads config at client: 32-192 Requests sent: 500K Expected Throughput: 6656 req/sec
Throughtput and Max Queue Length variations based out of resources used				
Server type	RabbitMQ type	Consumer-Redis type	Throughput	Max Queue Length
t2.micro	t2.micro	t2.medium	4459.15 req/sec	86
t2.medium	t2.micro	t2.medium	4958.448 req/sec	62
t2.large	t2.micro	t2.medium	5087.298 req/sec	55
t2.large	t2.micro	t2.large	5044.5435 req/sec	49
t2.large	t2.medium	t2.large	4974.03 req/sec	75

As we see above, we see our ideal performance when server type is at least a t2.medium. The changes to RabbitMQ configuration had no real impact on performance. The EC2 instance on which consumer code runs and Redis runs was a t2.medium. Scaling this up did not bring about any major difference.

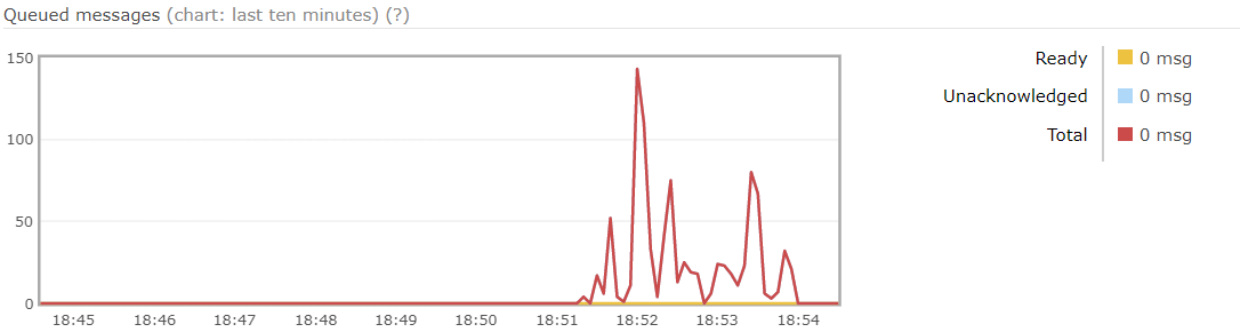
When cost is considered, the ideal configuration is when server is on t2.large, RabbitMQ on t2.micro and consumer on t2.medium

Screenshots:

Previously we observed that as the number of requests increases, the system stabilizes, and throughput improves. So, we will try sending 1 million requests and observe throughput and queue length

Throughput: 5821.9 req/sec	Max Queue Length: 131

Screenshots:



So, as the load hits large values, the system is still very stable with near-ideal performance and a queue length of 131 which is well within our acceptable queue length for a stable system.

Although the performance is optimal, we will now introduce throttling by introducing Circuit breaker at the client end.

Introducing Throttling:

The circuit breaker allows up to 1000 requests per minute before it interferes. When the load goes down again to 800 requests per second it switches back to state *closed*.

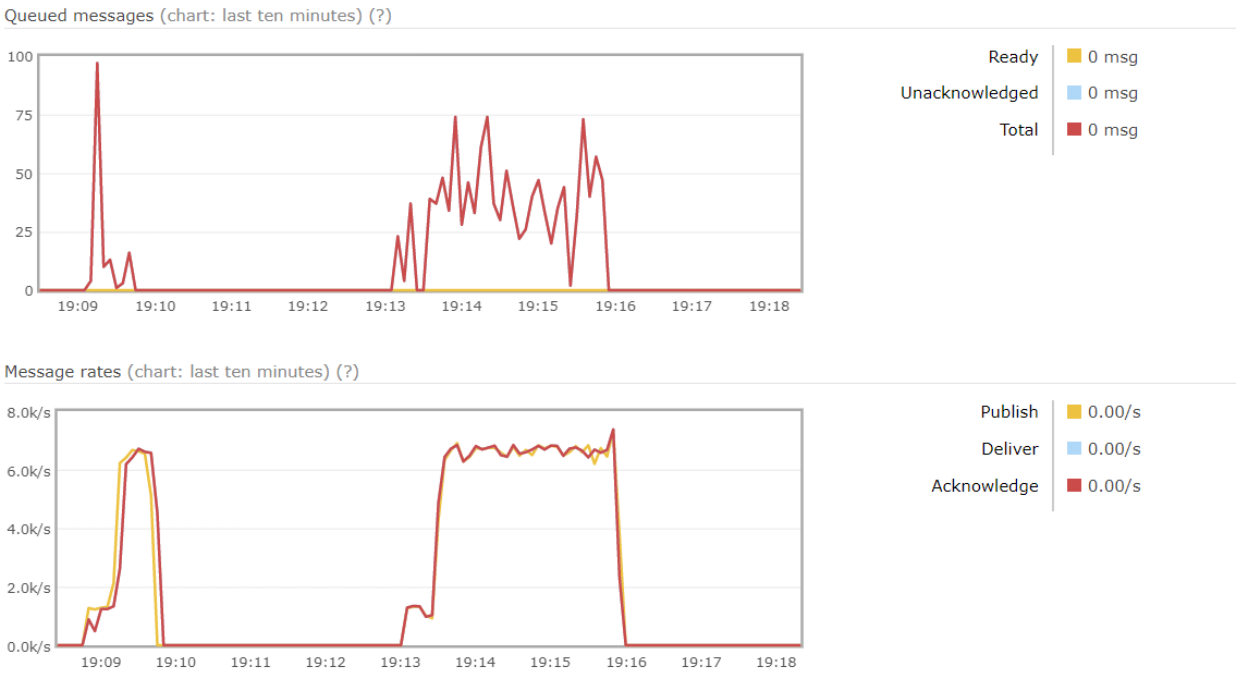
We send a million requests again with the above circuit breaker and see if this brings about any change to the Max queue length

Throughput: 5753.5 req/sec	Max Queue Length: 75

Screenshots:

```
C:\Users\srish\.jdks\corretto-17.0.4\bin\java.exe ...
Successful Requests: 1000000
Failed Requests: 0
Wall time(in milliseconds): 173805
Throughput observed: 5753.574 requests/sec
Throughput expected: 6656.000000000001 requests/sec

Process finished with exit code 0
```

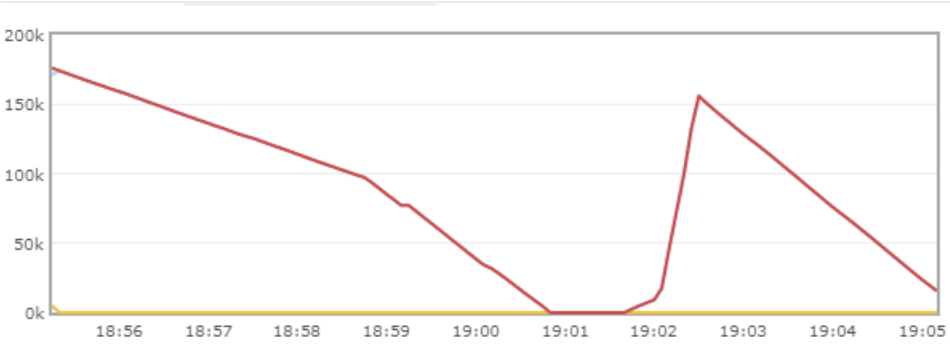


As seen above, there's a minor improvement to the queue length and a small dip in the throughput. Overall, there was not a very significant change in performance.

On having Consumer and Redis on different EC2 instances:

In this setup, we will now run consumer and Redis on different instances. The consumer will now run with 1000 threads and send 200K requests.

While throughput is not to be impacted, the queue length suffers. There is queue length of around 100K



Reflections

From the previous experimentation, we make the following observations. The resources that we use for running these programs run an important part in the performance of the system in terms of throughput.

The EC2 instance on which the servlet runs makes a huge impact on the throughput, where scaling up increases it. In all these cases, the queue length has always been within the small margin that can be ignored. So, there isn't much to optimize on that front. Had there been a need, using a circuit breaker would be a good approach. This might reduce throughput, but it overall stabilizes the system.

Also, the increase in the number of requests increases throughput because the system stabilizes.

However, when the consumer and Redis are on different instances, the performance is not very satisfactory. The acknowledgement rate did not go beyond around 2k req/sec, that too even when number of consumer threads is around 500. The instance was called up to a large and that dd not improve much.

We tried using a circuit breaker, but it took too long and was not a good solution we faced.

Finding insights on if hosting consumer and Redis on different instances is ever practical, and if so how to work on managing the queue length is the main takeaway from the assignment.