Assignment 2

6/16/2021 - Report

Github repo URL

https://github.com/coffinated/bsds-a2

Server Design

For this assignment, I built up the simple Java servlet I used in the first assignment a bit to do the text processing required and send messages to a queue. The main new feature here is the use of the AWS Java SDK for Amazon SQS. When the servlet is initiated, it creates a new queue (if one with its name doesn't yet exist) and starts a connection via the AWS client. Then, when a POST request comes in, it reads the line of text, splits it by spaces, and sends each resulting word to the queue in batches of 10 (or fewer if it gets to the end of the line). The batches are a feature of SQS that I took advantage of in a couple of places in this assignment - it allows the client to send/receive more messages with a single interaction with the queue. When batching messages, you must provide a unique ID for each one, so I added a random UUID here. I don't store them anywhere since this assignment didn't require the servlet to get any response back once it sent messages to the queue. The body of each message is a string of the format "{word, 1}". After sending the messages to the queue, the servlet sends back a 200 response and that interaction is complete. The only other thing I added to the servlet code is a destroy() method that closes the AWS client connection when the server is destroyed.

I chose to use an SQS standard queue in this system. This is a distributed queue that offers at-least-once delivery with high throughput, but no guaranteed ordering. AWS also offers a FIFO queue option, but for this assignment the standard queue would suffice since order wasn't important.

Consumer design

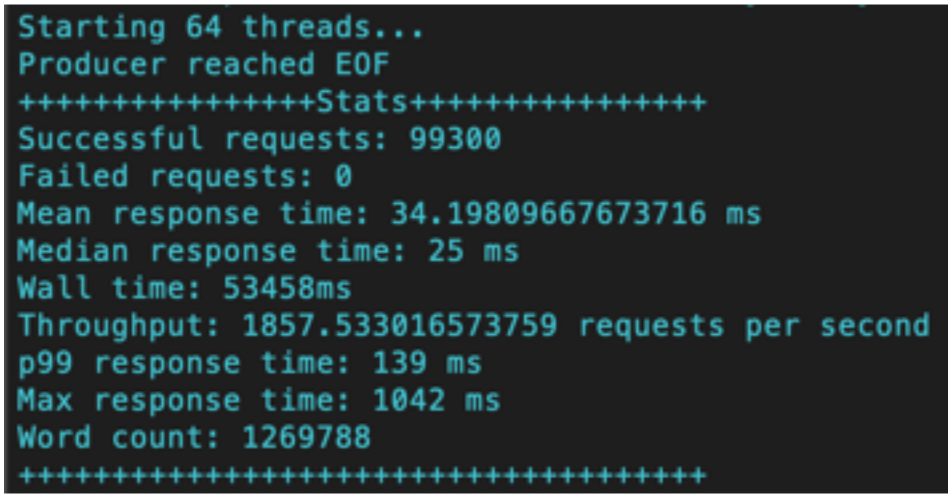
My consumer program gets access to the same queue in the SQS service. It requests the URL using the queue's name (and my access credentials which are provided behind the scenes by the SDK), then sends requests through the AWS client with that URL. It accepts a command line argument for the number of threads to run or defaults to 20. For my tests, I used 25 consumer threads. These receive messages from the queue in batches of (up to) 10, waiting up to 5 seconds for messages to reduce the number of requests being rapidly sent to the queue if it's empty.

Once a message has been received, its visibility timeout is 30 seconds by default in the SQS setup, and I kept that timeout as it seemed to work fine in my tests. The consumer doesn't need too much time for processing each request, and deletes each message after processing. If a message is received and not deleted within 30 seconds, it is visible again for other consumers to process. Processing includes updating a ConcurrentHashMap of each word's frequency in the message queue. I used this to check that my system was working as intended, by checking the resulting frequency of a few words against what a word processor's find-all command returned, and for the few words I tested, the results were as expected. This program runs until manually terminated, polling for new messages from the queue indefinitely. I originally had the consumer threads break out of the loop as soon as they received a response with 0 messages from the queue, then had the program close cleanly - however, I found it best to have my consumer running before the client started sending messages to the queue, so I changed it to just keep running regardless.

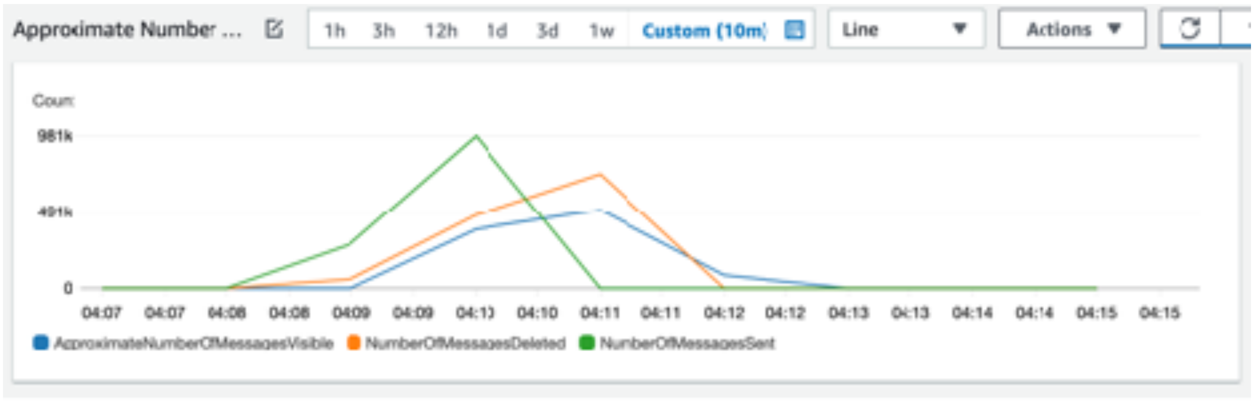
With the design sketched out, I will now proceed to the test runs. The architecture for my implementation included 4 EC2 instances: 2 servlet machines (both t2.medium) running with an AWS ELB distributing work to them, another t2.medium serving as the client, and yet another t2.medium running the consumer program. These were all in the same region to minimize latency. I tried running this with just one servlet machine and found that the number of client threads made very little difference to its throughput, so that's why I scaled out to 2 servlet machines with a load balancer.

Test run screenshots

With 64 client threads, I immediately noticed my throughput was maybe twice as high as on my assignment 1 tests. This was expected since I scaled my servlet both up and out, scaled my client up, and reduced the latency by running the client on an EC2 instance instead of from my own machine.

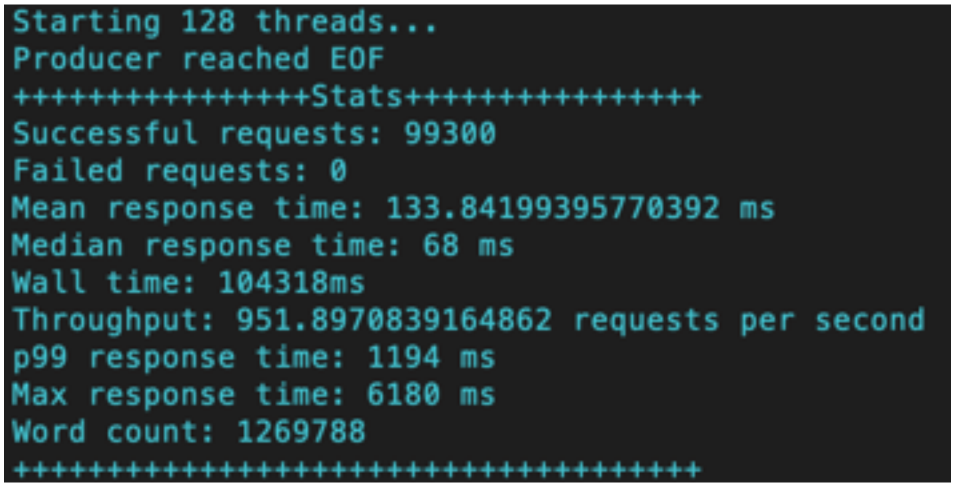


```
Starting 64 threads...
Producer reached EOF
+++++++++++++++Stats+++++++++++++++
Successful requests: 99300
Failed requests: 0
Mean response time: 34.19809667673716 ms
Median response time: 25 ms
Wall time: 53458ms
Throughput: 1857.533016573759 requests per second
p99 response time: 139 ms
Max response time: 1042 ms
Word count: 1269788
+++++++++++++++++++++++++++++++++++
```



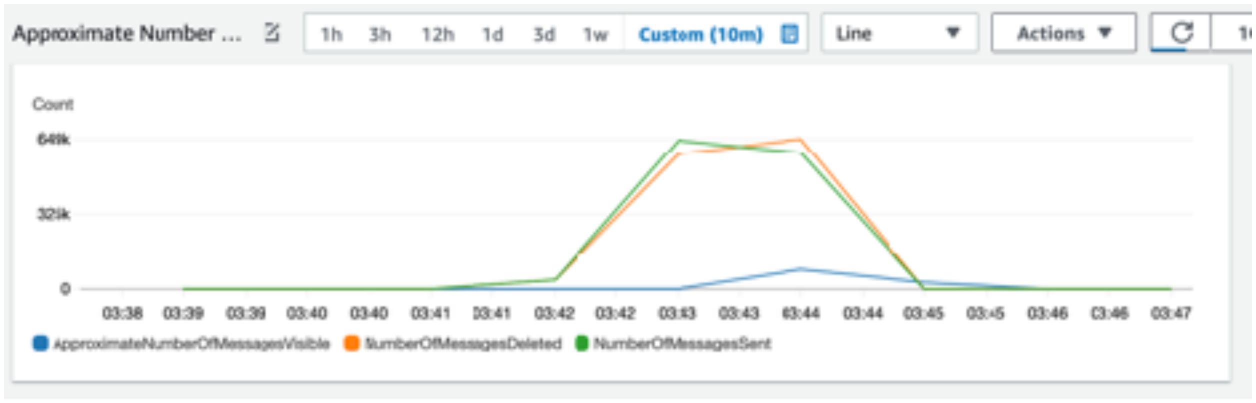
With 64 client threads, the consumer was able to process and delete messages from the queue more quickly than the AWS CloudWatch dashboard (screenshot above) could determine they were even in the queue in the first place, at least for the first few moments. The 'approximate number of messages visible' in the queue peaked at around 490k, which is a good deal of messages, but still less than half of the ~118 million that were sent in total.

128 client threads:

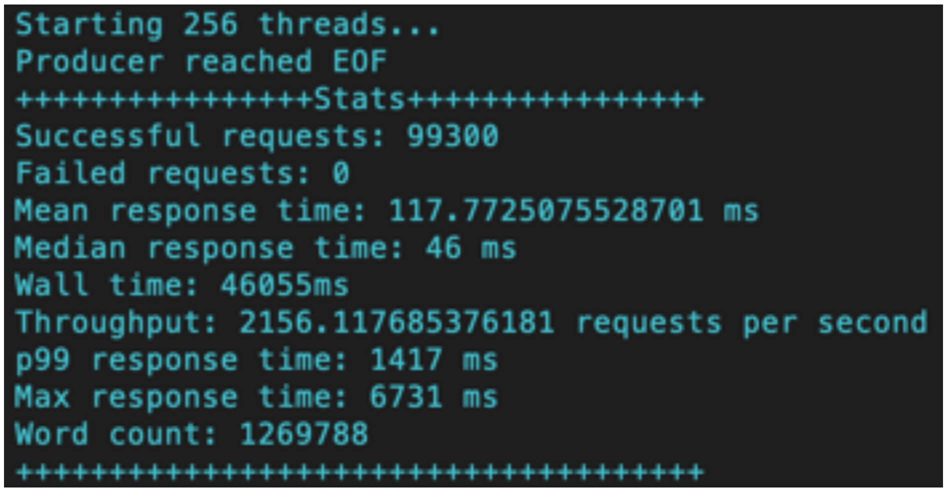


```
Starting 128 threads...
Producer reached EOF
+++++++++++++++Stats+++++++++++++++
Successful requests: 99300
Failed requests: 0
Mean response time: 133.84199395770392 ms
Median response time: 68 ms
Wall time: 104318ms
Throughput: 951.8970839164862 requests per second
p99 response time: 1194 ms
Max response time: 6180 ms
Word count: 1269788
+++++++++++++++++++++++++++++++++++++
```

My response time was not as great for this test run as for the 64-thread one, but as a result, the consumer was able to keep pace with the messages arriving on the queue much better. For this test the maximum 'approx. visible messages' reached about 85k in the queue.

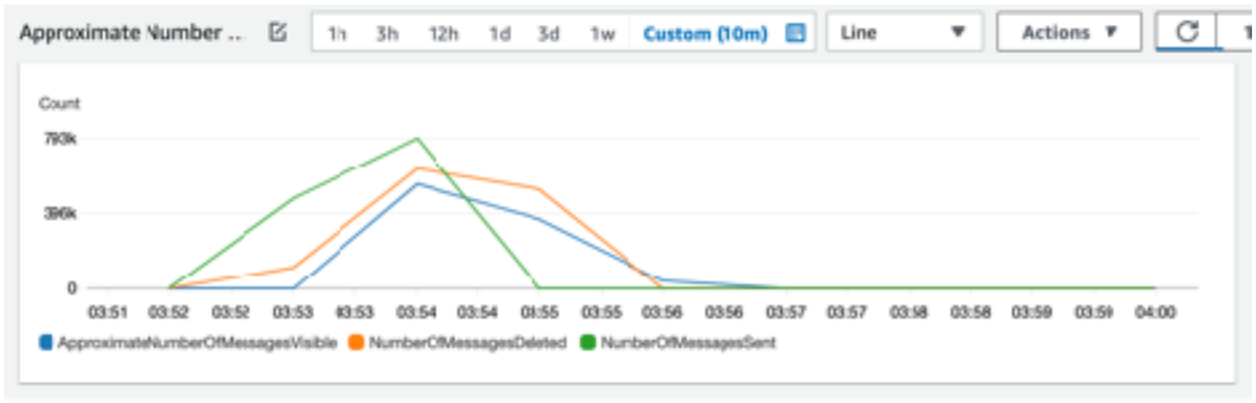


256 client threads:

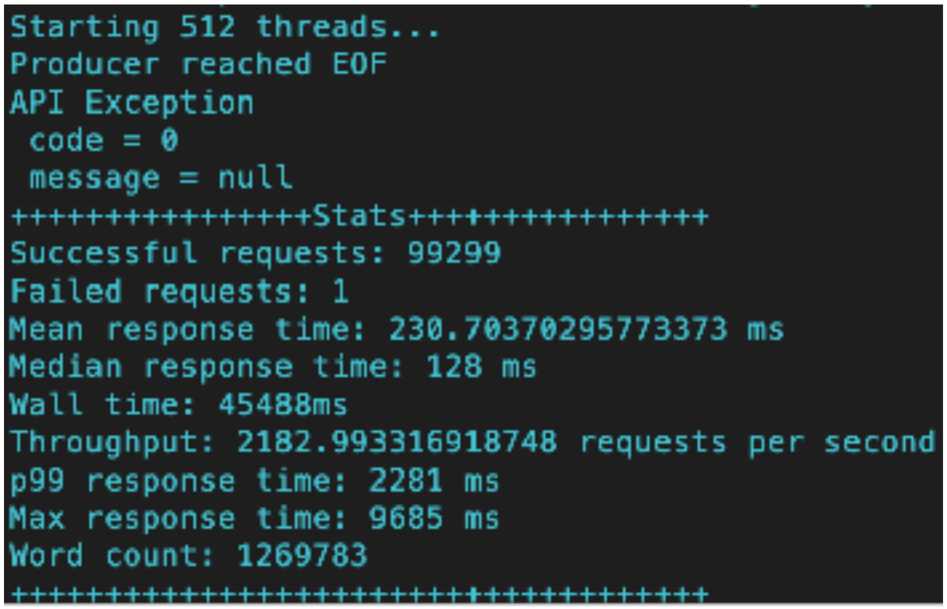


```
Starting 256 threads...
Producer reached EOF
+++++++++++++++Stats++++++++++++++++
Successful requests: 99300
Failed requests: 0
Mean response time: 117.7725075528701 ms
Median response time: 46 ms
Wall time: 46055ms
Throughput: 2156.117685376181 requests per second
p99 response time: 1417 ms
Max response time: 6731 ms
Word count: 1269788
++++++++++++++++++++++++++++++++++++
```

This test had good throughput, and a shorter wall time due to the higher number of client threads running. The queue got up to around 500k messages at its highest estimate, so once again the consumer was able to keep a decent pace.

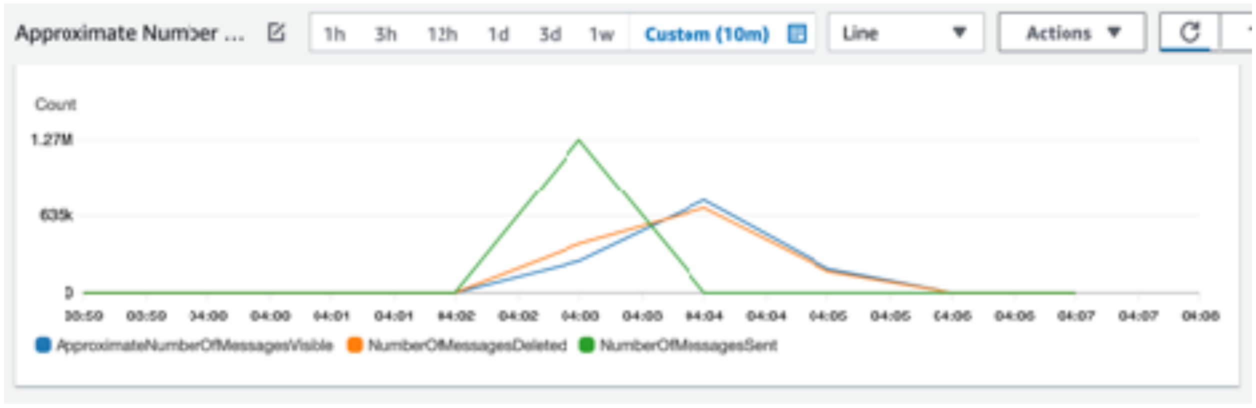


512 client threads:



```
Starting 512 threads...
Producer reached EOF
API Exception
 code = 0
 message = null
+++++++++++++++Stats+++++++++++++++
Successful requests: 99299
Failed requests: 1
Mean response time: 230.70370295773373 ms
Median response time: 128 ms
Wall time: 45488ms
Throughput: 2182.993316918748 requests per second
p99 response time: 2281 ms
Max response time: 9685 ms
Word count: 1269783
+++++++++++++++++++++++++++++++++++++
```

In a test run of 512 threads, I had one failed request where the server was too busy to respond, so clearly 2 servlet machines were not quite enough to handle this much traffic coming from the client. Still, the messages got put on the queue in record time, and the consumer couldn't receive them fast enough to quite keep up with its pace from the other test runs. The maximum approximate visible messages reached around 650k.



Conclusion

Using SQS for this assignment was a bit of a struggle since I had never used it before and it required quite a bit of setup and Googling how to do things using the AWS Educate access. I was glad to get some experience with it, but frustrated at the end with the metrics available after all that effort. I'd have liked slightly more fine-grained, accurate monitoring of my queue than what I found. The tests seem to indicate that my consumer, as it's written, can't process messages quickly enough to keep the queue near 0 messages for the kind of throughput the client was getting. It processed all the messages in usually about the same amount of time it took the client and servlet to publish them all to the queue, except in the 512-client-thread case. For the next assignment, I may need to refine the consumer or find a way to fine-tune the monitoring through AWS CloudWatch.