Assignment 2 Write Up

CS 6650, Spring 2022
Sean Stevens

Github Repository URL

https://github.com/seanmstevens/CS6650-Assignment2

Server Design

The design for my server uses a shared and synchronized channel pool to reduce the overhead associated with creating RabbitMQ channels on the fly for each request, which would hinder performance significantly. The channel pool is implemented using a BlockingQueue, whereby the queue is populated upon servlet initiation, and threads can "borrow" a channel from the queue, publish their message, then return the channel back to the pool for reuse by other threads. By default, 20 channels are created for the Tomcat threads to share.

Major Classes

- ***ApiTest:*** A utility class to test the responses received from various endpoints defined in the servlets.
- ***Parameter:*** A POJO convenience class that stores information regarding the JSON parameters passed in through POST requests. The class has a validation method that is used to determine if the JSON payload provided by the client is valid.
- ***ResortsServlet:*** The servlet that handles all requests in the resorts family of endpoints. Provides dummy data for most requests.
- ***SkiersServlet:*** The primary servlet that handles all requests in the skiers family of endpoints. This servlet contains most of the business logic of the application. Here, the channel pool is declared upon servlet initialization and the Tomcat-managed threads publish messages to the queue whenever one is dispatched to handle a POST request.
- ***StatisticServlet:*** A servlet that handles statistics requests to the API. Provides dummy data for now.

Packages

The packages used in the server include the generated Swagger client, the JavaX Servlet package(s), and the RabbitMQ AMQP Client packages.

Consumer Design

The message consumer uses a channel-per-thread model whereby a thread will declare a channel to the message queue and, using the push model of message consumption, provides a callback method to the message broker. When a message comes through and a particular channel is selected, the callback is called and the message is processed. Once processing is complete, the consumer acknowledges the message, indicating to the broker that the message can be removed from the queue.

The hashmap that keeps track of the lift IDs for each skier is implemented using a ConcurrentHashMap to enable concurrent access to different regions of the map, with ConcurrentLinkedQueue values. The queues serve as thread-safe collections that can store the lift ID values in insertion order.

Major Classes

- ***Consumer:*** The main entry point class that initializes the fixed thread pool (using a predetermined number of threads) and creates the RabbitMQ connection and hashmap to store the lift ID records for each skier. By default, 20 threads are created.
- ***ConsumerRunnable:*** The task that is submitted to the ExecutorService thread pool. This runnable is responsible for creating the RabbitMQ channel given the passed in connection object, processing the message contents once it arrives from the broker, and updating the hashmap in a thread-safe manner. Once a message is processed, the runnable performs a manual acknowledgment of the message to the broker, informing it that the message can be dropped from the queue.

Packages

The two packages used in the consumer are the RabbitMQ AMQP Client package as well as the Gson JSON parsing package.

RabbitMQ Run Results

Through testing, I found that a single t2.micro EC2 instance was sufficient to handle request loads from a client with 128 threads and 20,000 skiers. The following runs show the results of 64, 128, and 256 thread runs with no load balancing on this instance type. For all the RabbitMQ charts shown in this section, the purple line represents the consumer's manual acknowledgment rate and the yellow line represents the producer's (server's) publishing rate. The red line shows the queue size.

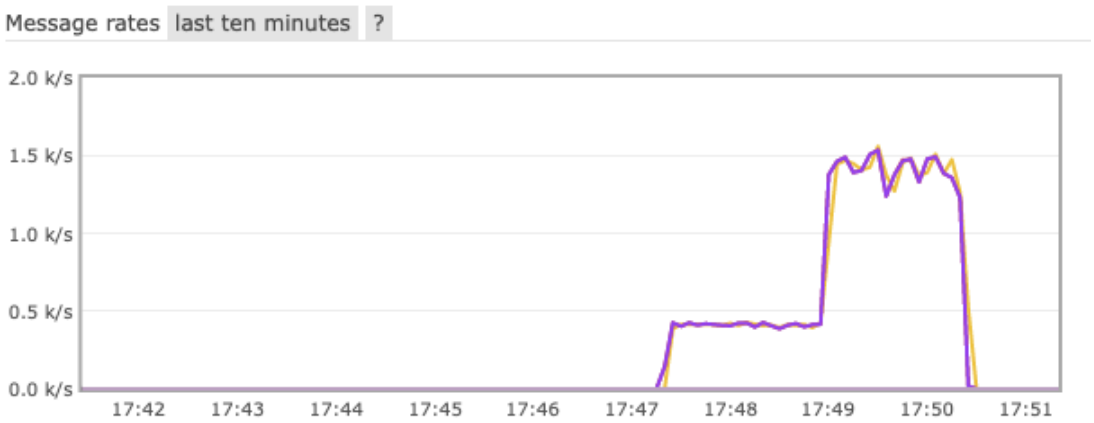
For load balancing, I replicated the single instance four times in three different availability zones and placed them behind a network load balancer. In addition, I increased the number of channels available in the pool for these servers from the default 20 to 100 each. So, there were 400 server-side producer channels connected to the RabbitMQ queue instead of just 20. Despite this, I did not see a huge jump in performance with high request loads. Fortunately, the load balanced instances saw an increase in message rate from 256 to 512 threads and there were far fewer socket timeouts overall, but it did not seem this was enough to handle the highest request loads in general.

Finally, I decided to test request load with a single t3.medium instance. Similar to the load balanced scheme, it seems that high request loads do not see a drop in performance, but the larger instance does not afford a performance jump in accordance with Little's Law predictions. The limiting factor seems to be the mean response time and socket timeouts.

Single t2.micro Instance Results

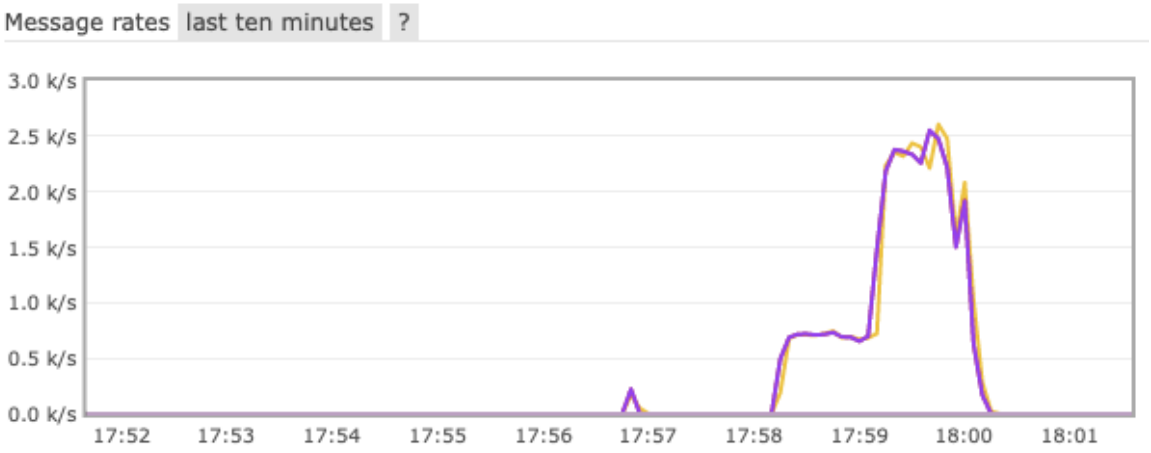
64 Threads (Single t2.micro Server Instance):

```
------ RUN STATISTICS ------

Total successes: 159814
Total failures: 0
Time elapsed (sec): 182.671
Total throughput (reqs/sec): 874.87335

Max response time: 648
Min response time: 25
Mean response time: 43.53917679302189
Median response time: 41
99th percentile response time: 91

Process finished with exit code 0
```



128 Threads (Single t2.micro Server Instance):

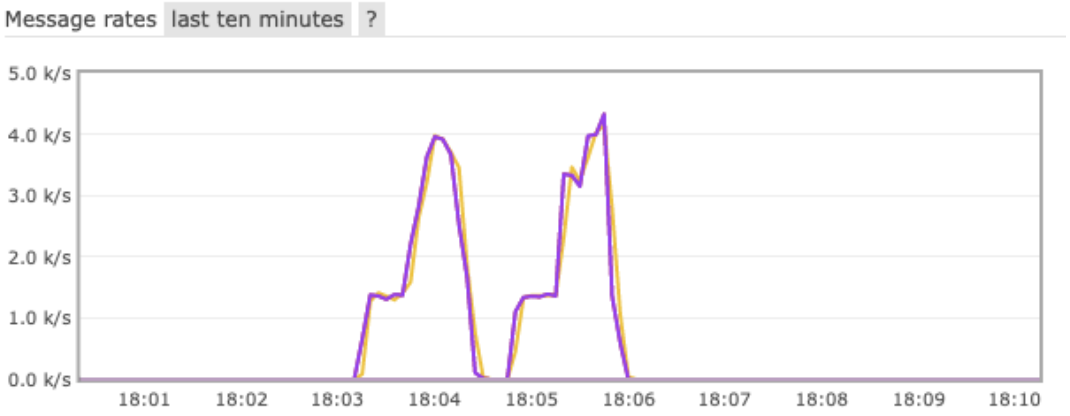
```
------ RUN STATISTICS ------

Total successes: 159820
Total failures: 0
Time elapsed (sec): 118.177
Total throughput (reqs/sec): 1352.3782

Max response time: 10017
Min response time: 26
Mean response time: 53.80333099754743
Median response time: 48
99th percentile response time: 167

Process finished with exit code 0
```



256 Threads (Single t2.micro Server Instance):

Here is where we can see the limits of the single t2.micro instance start to show themselves. During this run, there were multiple socket timeouts, which led to increased response times on average. There were also spikes where the queue would be rapidly populated, and then quickly drained.

```
------ RUN STATISTICS ------

Total successes: 159769
Total failures: 0
Time elapsed (sec): 67.522
Total throughput (reqs/sec): 2366.1768

Max response time: 10100
Min response time: 28
Mean response time: 62.59062552794759
Median response time: 55
99th percentile response time: 137

Process finished with exit code 0
```



512 Threads (Single t2.micro Server Instance):

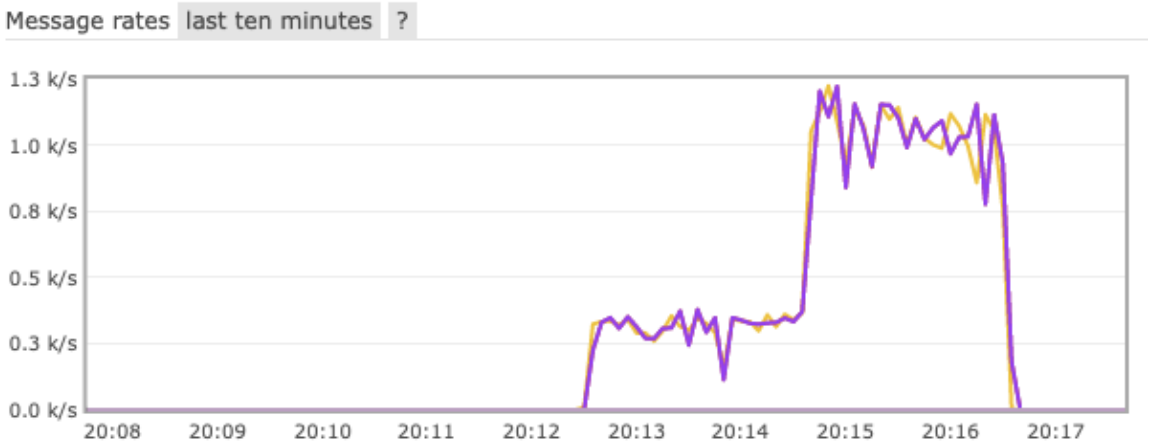
Again, the single instance is showing deficiencies. With a 512-thread run, the throughput ended up being significantly lower, with over double the average response times and 2 actual failures.

```
------ RUN STATISTICS ------

Total successes: 159793
Total failures: 2
Time elapsed (sec): 61.321
Total throughput (reqs/sec): 2605.8447

Max response time: 10190
Min response time: 31
Mean response time: 126.41659056507
Median response time: 103
99th percentile response time: 505

Process finished with exit code 0
```

Load Balanced Results

64 Threads (4 x t2.micro Instances):

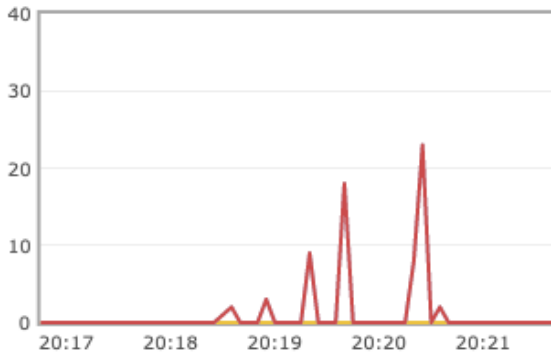
```
------ RUN STATISTICS ------

Total successes: 159814
Total failures: 0
Time elapsed (sec): 241.483
Total throughput (reqs/sec): 661.80225

Max response time: 1367
Min response time: 27
Mean response time: 58.344663170936215
Median response time: 51
99th percentile response time: 220

Process finished with exit code 0
```



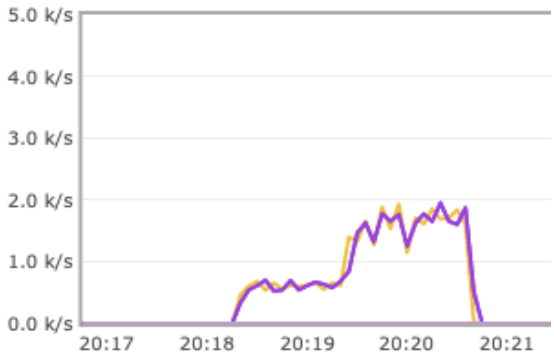
128 Threads (4 x t2.micro Instances):

```
------ RUN STATISTICS ------

Total successes: 159820
Total failures: 0
Time elapsed (sec): 140.604
Total throughput (reqs/sec): 1136.6675

Max response time: 1508
Min response time: 29
Mean response time: 72.1689338005256
Median response time: 61
99th percentile response time: 367

Process finished with exit code 0
```

Queued messages last ten minutes ?



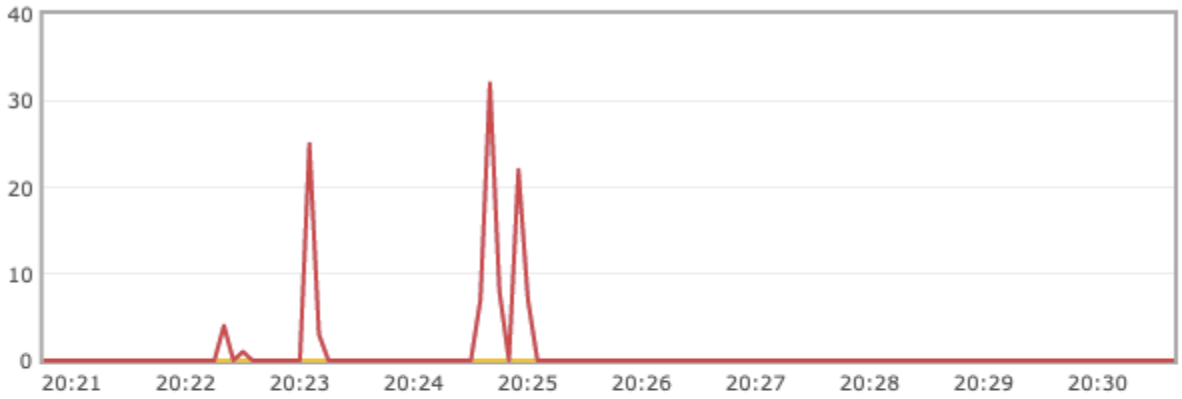
Message rates last ten minutes ?



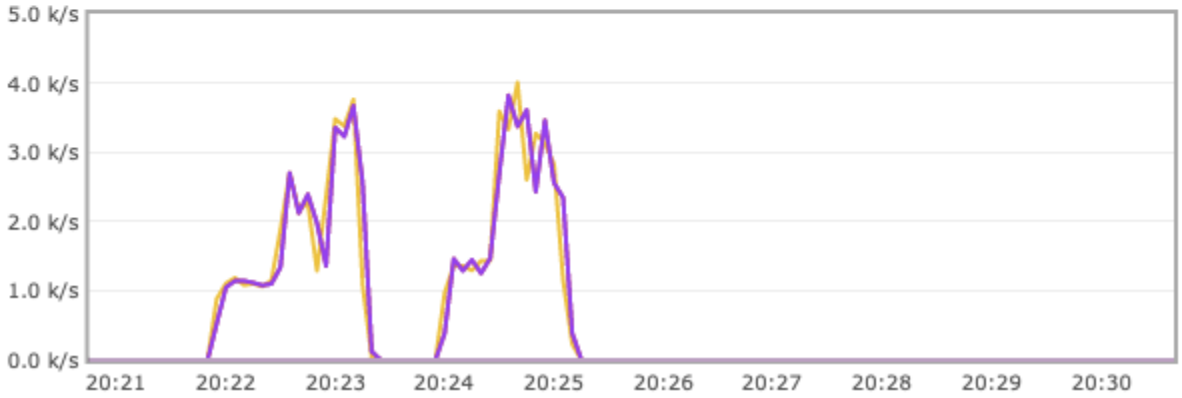
256 Threads (4 x t2.micro Instances):

```
------ RUN STATISTICS ------

Total successes: 159769
Total failures: 0
Time elapsed (sec): 72.331
Total throughput (reqs/sec): 2208.8591

Max response time: 10094
Min response time: 27
Mean response time: 70.22116455759578
Median response time: 58
99th percentile response time: 357

Process finished with exit code 0
```

Queued messages last ten minutes ?

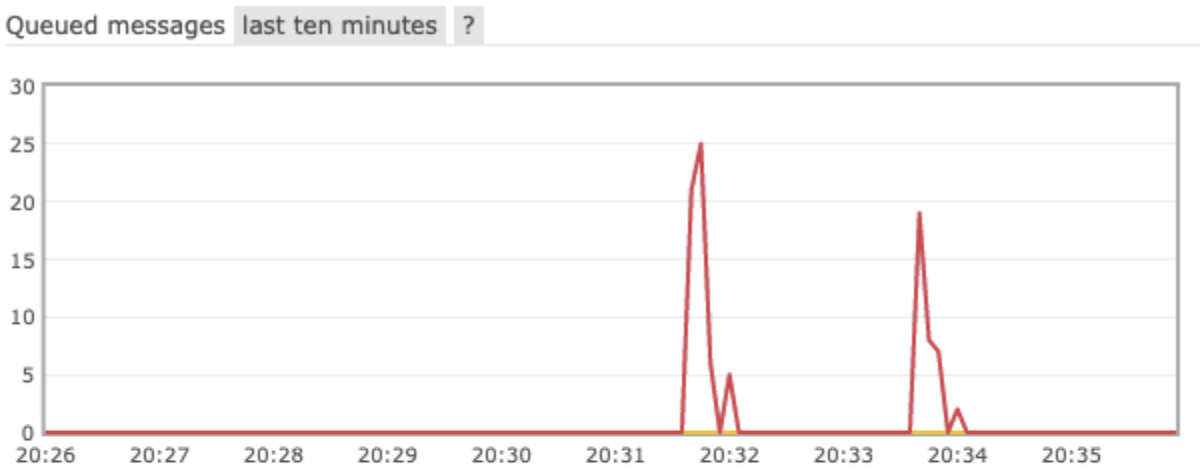


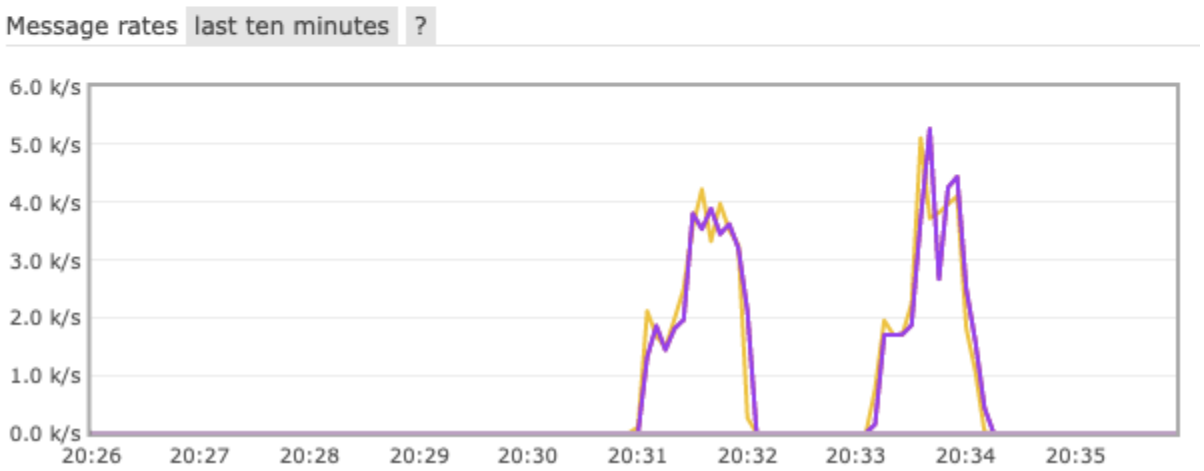
Message rates last ten minutes ?



512 Threads (4 x t2.micro Instances):

```
------ RUN STATISTICS ------

Total successes: 159794
Total failures: 1
Time elapsed (sec): 60.196
Total throughput (reqs/sec): 2654.5618

Max response time: 10155
Min response time: 30
Mean response time: 112.52279863395839
Median response time: 83
99th percentile response time: 681

Process finished with exit code 0
```

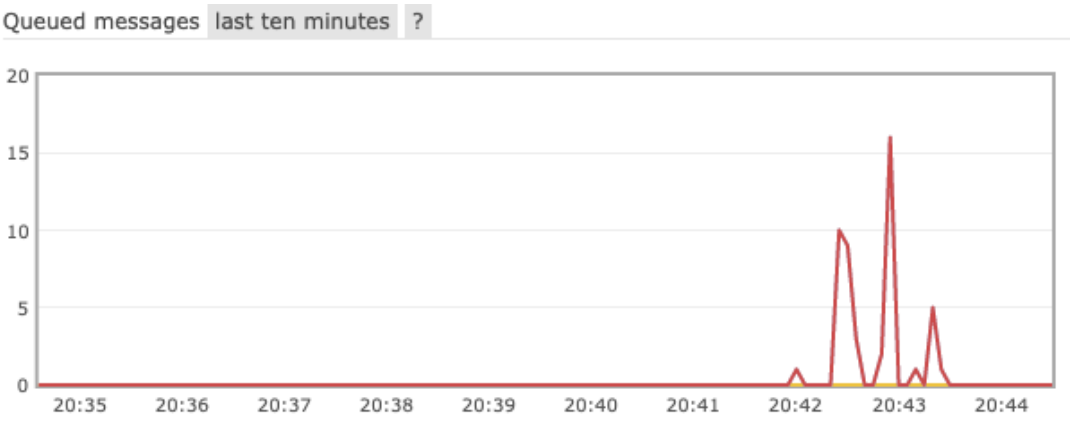


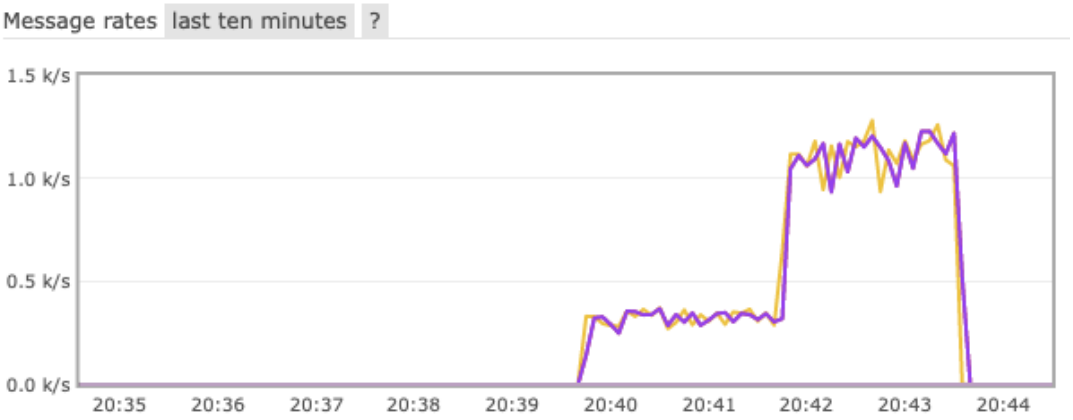


Single t3.medium Instance Results

64 Threads (Single t3.medium Instance):

```
------ RUN STATISTICS ------

Total successes: 159814
Total failures: 0
Time elapsed (sec): 231.095
Total throughput (reqs/sec): 691.5511

Max response time: 1002
Min response time: 28
Mean response time: 55.20718460209994
Median response time: 49
99th percentile response time: 202

Process finished with exit code 0
```

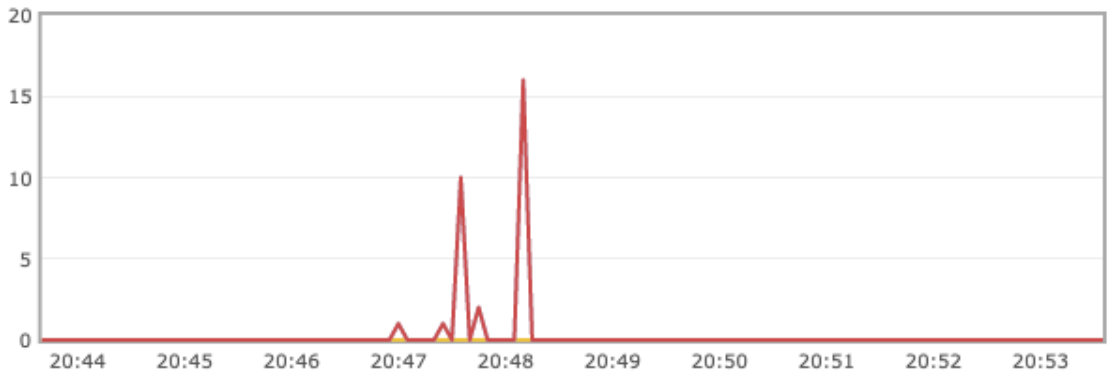




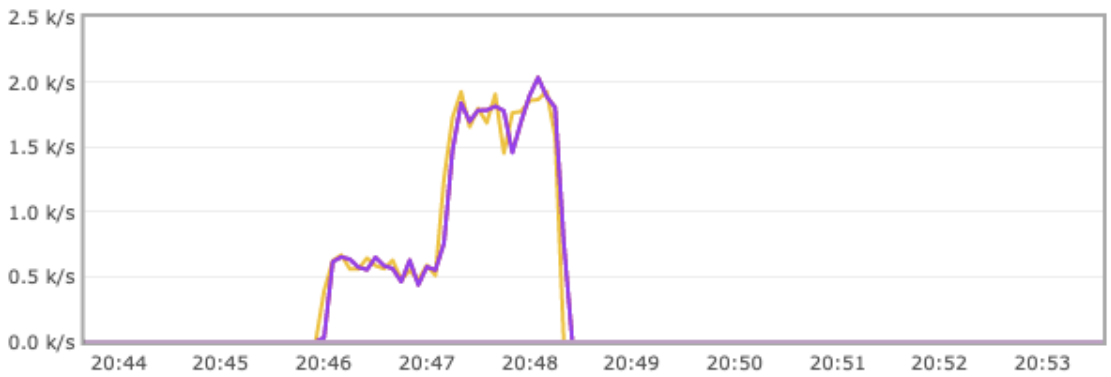
128 Threads (Single t3.medium Instance):

```
------ RUN STATISTICS ------

Total successes: 159820
Total failures: 0
Time elapsed (sec): 138.324
Total throughput (reqs/sec): 1155.4032

Max response time: 1015
Min response time: 29
Mean response time: 68.02796270804656
Median response time: 59
99th percentile response time: 297

Process finished with exit code 0
```

Queued messages last ten minutes ?



Message rates last ten minutes ?



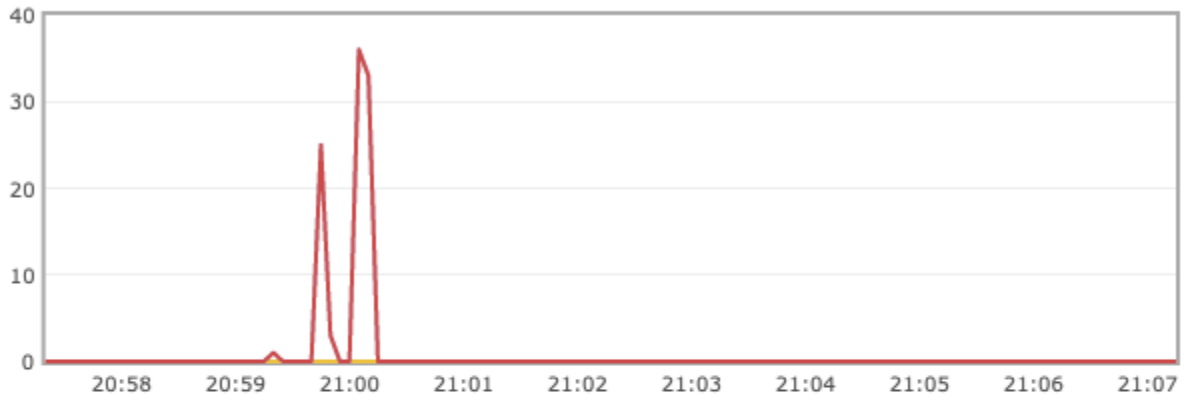
256 Threads (Single t3.medium Instance):

```
------ RUN STATISTICS ------

Total successes: 159769
Total failures: 0
Time elapsed (sec): 78.617
Total throughput (reqs/sec): 2032.245

Max response time: 1486
Min response time: 31
Mean response time: 80.66496003605205
Median response time: 67
99th percentile response time: 376

Process finished with exit code 0
```

512 Threads (Single t3.medium Instance):

```
------ RUN STATISTICS ------

Total successes: 159795
Total failures: 0
Time elapsed (sec): 62.901
Total throughput (reqs/sec): 2540.4207

Max response time: 10272
Min response time: 31
Mean response time: 123.95962579393635
Median response time: 106
99th percentile response time: 549

Process finished with exit code 0
```

Queued messages last ten minutes ?



Message rates last ten minutes ?

Conclusion

In my testing with several different schemes, I found that there was some performance gain in both the load balanced (scaled out) and the larger instance (scaled up) configurations. Although the jump in performance was not drastic, it is clear that load balancing or a larger instance type allowed the highest request loads to increase input, rather than bogging the server down due to memory or CPU limitations.

Although load balancing allowed more vCPUs to be available for processing requests, there was likely some additional overhead introduced by the load balancer itself. For the larger instance, there were two vCPUs available and 4 GB of memory on the t3.medium instance, so memory did not present as much of a bottleneck. Ultimately, it seems that the timeouts for failed requests were the true bottleneck, as I found 10,000 ms maximum response times in each of these configurations, so allowing the client to “fail fast” will likely be the best improvement to be made in order to increase throughput.