Assignment 2

GitHub Link:https://github.com/guojunseven/Building-Scalable-Distributed-System/tree/master/Assignment02

1. Client Side :

I keep the client side as default as the client from assignment1

2. Server Side:

For the server side, I implement the real business logic in the **WordCountServerice** class. And using a rabbit-mq channels pool to store and reuse the channels.

1) In the Init() method, I set up one single rabbitmq connection and use it for the furture publishing. And then I create a channel pool with 80 channel objects for every servlet thread to use. All of the channels declare and use the same specified queue (non persistent)

```
    @Override
    public void init() throws ServletException {
        super.init();
        // set mq connection
        setMqConnection();
        // set channels pool
        setChannelPool();
        functions.put("/wordcount", new WordCountService(channelObjectPool, queueName));
    }
private void setChannelPool() {
    this.channelObjectPool = new ChannelPool( poolSize: 80);
    for (int i = 0; i < 80; i++) {
        try {
            Channel channel = connection.createChannel();
            channel.queueDeclare(queueName, durable: false, exclusive: false, autoDelete: false, arguments: null);
            this.channelObjectPool.returnObject(channel);
        } catch (Exception e) {
            System.err.println("fail to create rabbitmq channel");
        }
    }
}
```

2) My **WordCountServerice** service class now accepts two parameters in construction : **queueName** for publishing and **channelsPool** to retrieve a channel for sending asynchronous messages.
3) For performance, I post the **map** storing word-frequency pairs directly, which means I only need to publish once for every single request from the client which I think is efficient. And to publish this message, I first need to borrow a channel object from the pool and return this intact channel into the pool after publishing the message. Finally, I just need to return the size of unique words to the client.

```
public WordCountService(ObjectPool<Channel> channelObjectPool, String queueName) {
    this.channelObjectPool = channelObjectPool;
    this.queueName = queueName;
}
```

```
service.WordCountService
private String queueName
```

```
@Override
public int apply(String target) throws Exception {
    HashMap<String, Integer> wordCount = new HashMap<~>();
    String[] words = target.split( regex: "\\s+");
    for (String word : words) {
        wordCount.put(word, wordCount.getOrDefault(word,  defaultValue: 0) + 1);
    }

    Channel channel = channelObjectPool.borrowObject();
    channel.basicPublish( exchange: "", queueName,  props: null, SerializationUtils.serialize(wordCount));
    // return the channel to the pool
    channelObjectPool.returnObject(channel);
    return wordCount.size();
}
```

3. Rabbitmq Instance and consumer

Because we are building a distributed system, then I decide to deploy my rabbitmq in a **t2.micro** instance which can be upgraded to higher specification later. And the consumer is deployed in another machine to retrieve the messages remotely.

My consumer is multithreaded and accept a **maxThreads** parameter through the command line.

```
public class MultiThreadedConsumer {
    private static String queueName = "wordCount";
    private static int maxThreads;
    private static  ConcurrentHashMap<String, Integer> map = new ConcurrentHashMap<>();

    public static void main(String[] args) throws IOException {
        if (args == null || args.length == 0) {
            System.err.println("must provide threads number");
        } else {
            maxThreads = Integer.parseInt(args[0]);
        }

        // set properties
        Properties properties = new Properties();
        properties.load(MultiThreadedConsumer.class.getClassLoader().getResourceAsStream( name: "application.properties"));

        ConnectionFactory factory = new ConnectionFactory();
        factory.setHost(properties.getProperty("hostname"));
        factory.setUsername(properties.getProperty("username"));
        factory.setPassword(properties.getProperty("password"));

        CyclicBarrier synk = new CyclicBarrier( parties: maxThreads + 1);
        try {
            Connection connection = factory.newConnection();
            for (int i = 0; i < maxThreads; i++) {
                Channel channel = connection.createChannel();
                channel.queueDeclare(queueName,  durable: false,  exclusive: false,  autoDelete: false,  arguments: null);
                new consumerHandler(map, channel, queueName, synk).start();
            }
            synk.await();
        } catch (IOException | TimeoutException e) {
            System.err.println("fail to create rabbitmq connection");
        } catch (InterruptedException e) {
            e.printStackTrace();
        } catch (BrokenBarrierException e) {
            e.printStackTrace();
        }
```

Every thread is a consumer handler i.e the consumer itself. It continuously retrieves messages from the specified queue and analyze it then add the frequency of corresponding word to the shared **concurrentHashMap** which is thread safe and can be accessed by multi-threads.

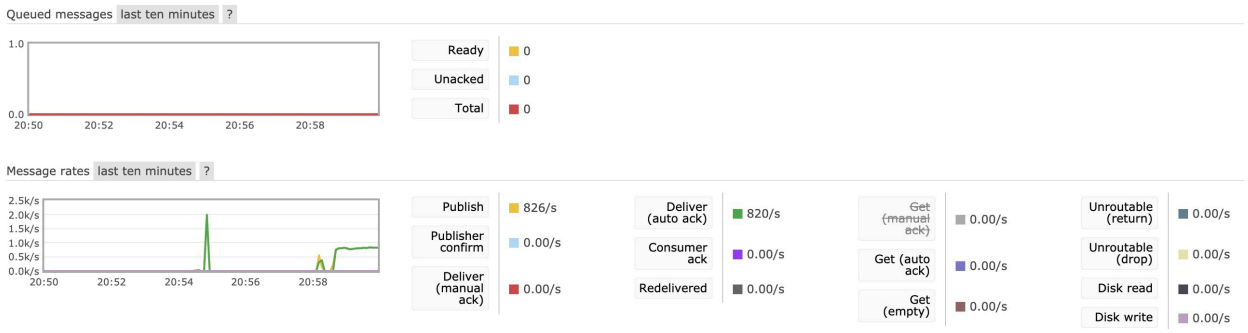
```java
public consumerHandler(
        ConcurrentHashMap<String, Integer> map, Channel channel, String queueName, CyclicBarrier synk) {
    this.map = map;
    this.channel = channel;
    this.queueName = queueName;
    this.synk = synk;
}

@Override
public void run() {
    DeliverCallback deliverCallback = (consumerTag, delivery) -> {
        Map<String, Integer> data = SerializationUtils.deserialize(delivery.getBody());
        Iterator<Map.Entry<String, Integer>> iter = data.entrySet().iterator();
        while (iter.hasNext()) {
            Map.Entry<String, Integer> entry = iter.next();
            String key = entry.getKey();
            map.put(key, map.getOrDefault(key, defaultValue: 0) + entry.getValue());
        }
    };
    try {
        channel.basicConsume(queueName, autoAck: true, deliverCallback, consumerTag -> { });
    } catch (IOException e) {
        System.err.println("fail to consume messages");
    }
    try {
        synk.await();
    } catch (InterruptedException | BrokenBarrierException e) {
        e.printStackTrace();
    }
}
```

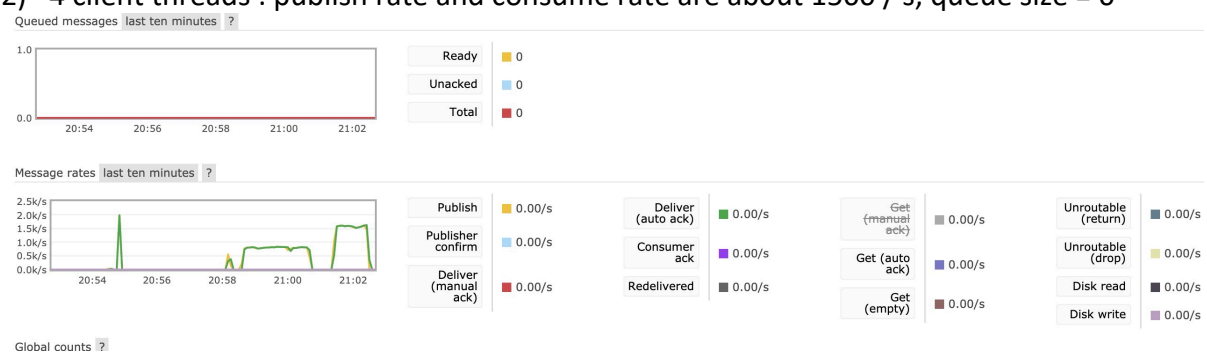
Results

1. Single ec2 t2.micro sever with t2.large client and 5 consumer threads (non-persistent queue)

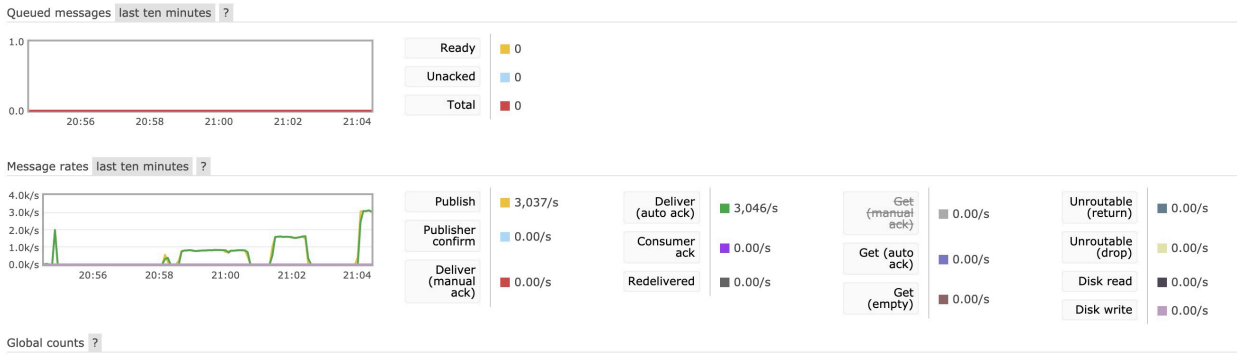
1) 2 client threads : publish rate and consume rate are about 800 / s, queue size = 0



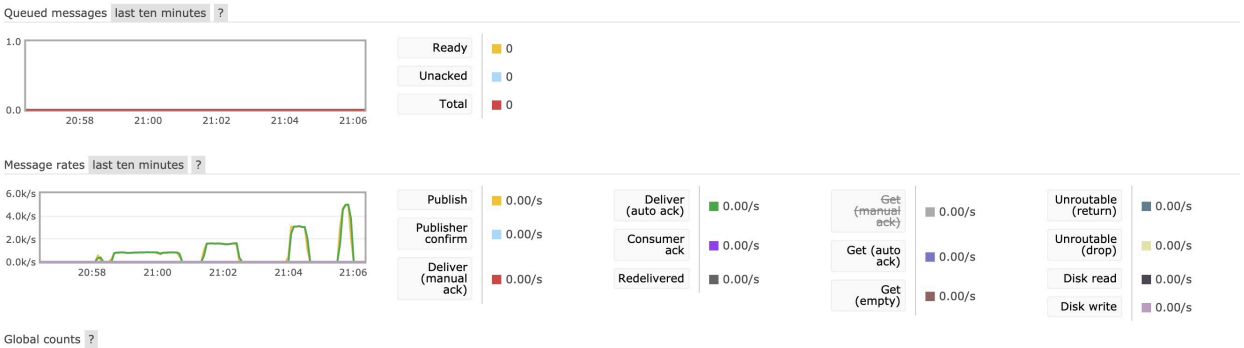
2) 4 client threads : publish rate and consume rate are about 1500 / s, queue size = 0



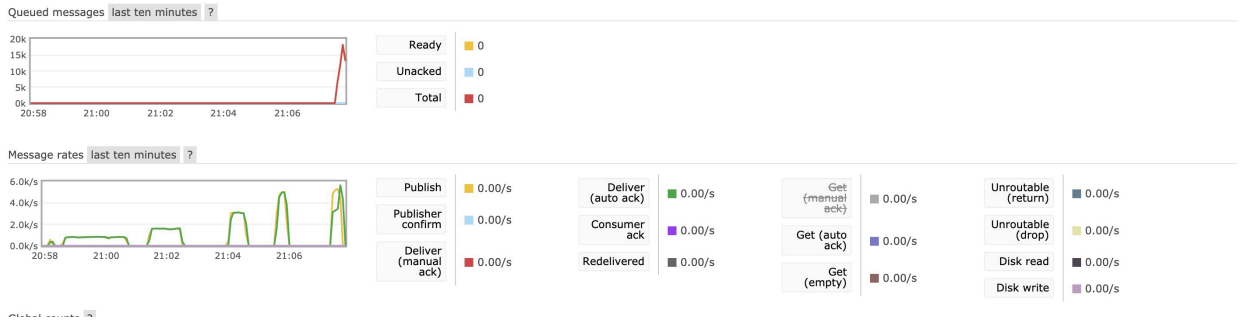
3) 8 client threads : publish rate and consume rate are about 3000 / s, queue size = 0



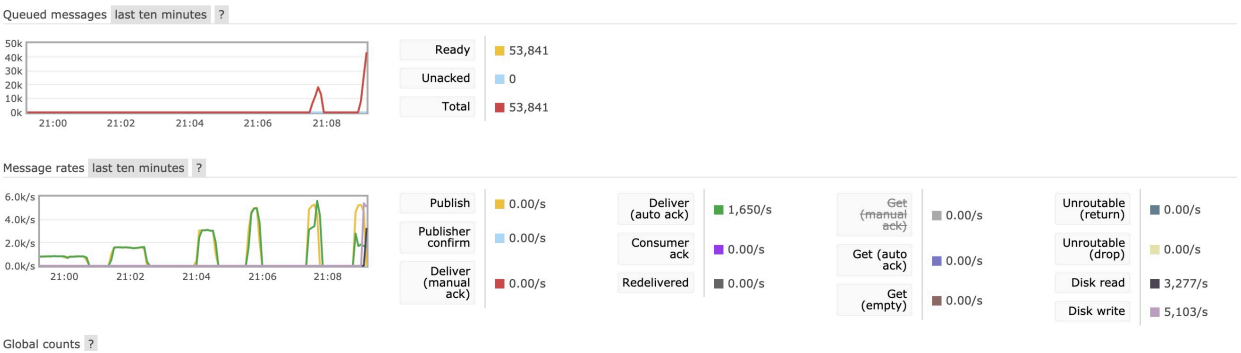
4) 16 client threads : publish rate and consume rate are about 5000 / s, queue size = 0



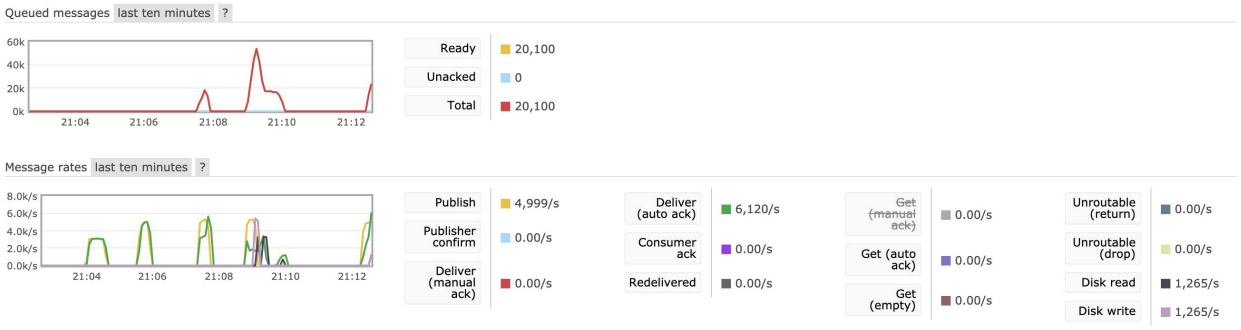
5) 32 client threads : publish rate is about 5500 / s and consume rate is slightly lower. queue size = 20k



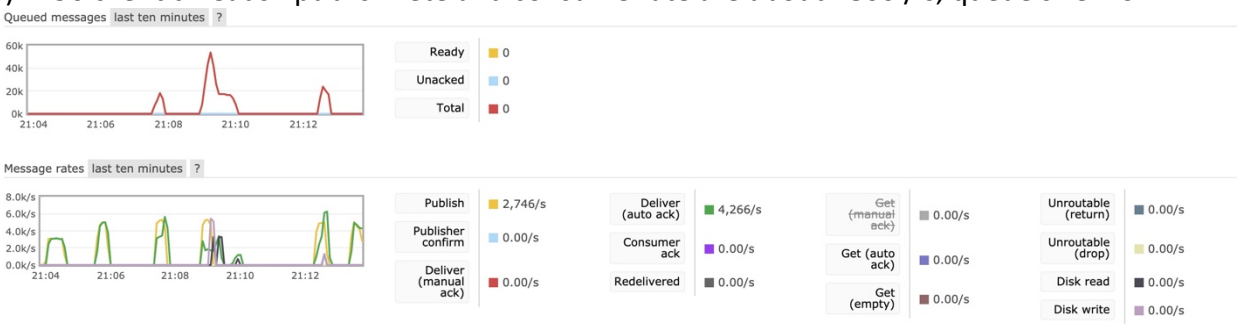
6) 64 client threads : publish rate is about 5500 / s and there is a fluctuation in consume rate in about 3000 / s, queue size = 55k



7) 128 client threads : publish rate and consume rate are about 5000 /s, queue size = 24k



8) 256 client threads : publish rete and consume rate are about 4800 / s, queue size = 0




```
^C[ec2-user@ip-172-31-95-38 assignment02]$ java -jar client.jar testdata.txt 2
All 2 threads processing completed!
-------------------------------------------------------
------------------------STATS------------------------
Total number of successful requests: 99300
Total number of unsuccessful requests: 0
Mean response time : 2.4894 ms
Median response time : 2.4186 ms
99th Percentile response time : 3.8263 ms
Max response time : 231.9107 ms
Total wall time: 124625 ms
Throughput: 796.79
-------------------------------------------------------
[ec2-user@ip-172-31-95-38 assignment02]$ java -jar client.jar testdata.txt 4
All 4 threads processing completed!
-------------------------------------------------------
------------------------STATS------------------------
Total number of successful requests: 99300
Total number of unsuccessful requests: 0
Mean response time : 2.5227 ms
Median response time : 2.4448 ms
99th Percentile response time : 3.8313 ms
Max response time : 230.8209 ms
Total wall time: 63187 ms
Throughput: 1571.53
-------------------------------------------------------
[ec2-user@ip-172-31-95-38 assignment02]$ java -jar client.jar testdata.txt 8
All 8 threads processing completed!
-------------------------------------------------------
------------------------STATS------------------------
Total number of successful requests: 99300
Total number of unsuccessful requests: 0
Mean response time : 2.6059 ms
Median response time : 2.4999 ms
99th Percentile response time : 3.9762 ms
Max response time : 237.8526 ms
Total wall time: 32696 ms
Throughput: 3037.07
-------------------------------------------------------
[ec2-user@ip-172-31-95-38 assignment02]$ java -jar client.jar testdata.txt 16
All 16 threads processing completed!
-------------------------------------------------------
------------------------STATS------------------------
Total number of successful requests: 99300
Total number of unsuccessful requests: 0
Mean response time : 3.3302 ms
Median response time : 3.1012 ms
99th Percentile response time : 7.1172 ms
Max response time : 233.4489 ms
Total wall time: 20914 ms
Throughput: 4748.02
-------------------------------------------------------
```

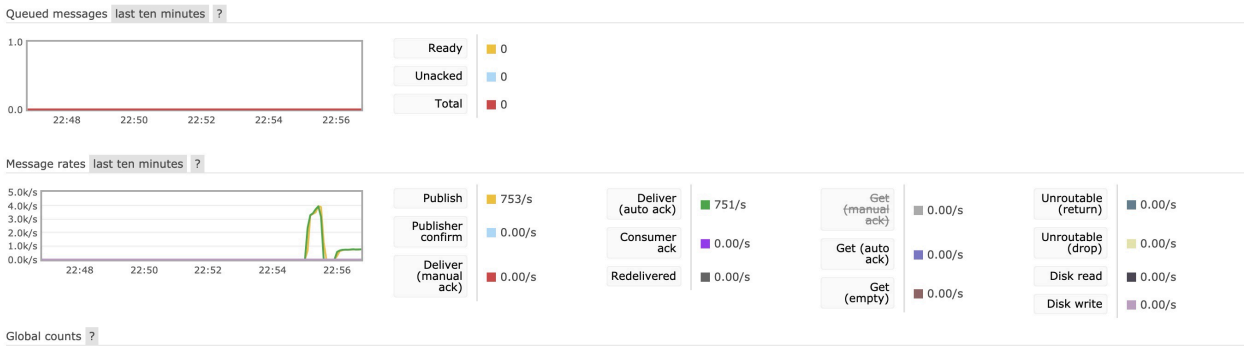
```
[ec2-user@ip-172-31-95-38 assignment02]$ java -jar client.jar testdata.txt 32
All 32 threads processing completed!
-------------------------------------------------------
------------------------STATS------------------------
Total number of successful requests: 99300
Total number of unsuccessful requests: 0
Mean response time : 6.1282 ms
Median response time : 5.4703 ms
99th Percentile response time : 18.4245 ms
Max response time : 262.2927 ms
Total wall time: 19205 ms
Throughput: 5170.53
-------------------------------------------------------
[ec2-user@ip-172-31-95-38 assignment02]$ java -jar client.jar testdata.txt 64
All 64 threads processing completed!
-------------------------------------------------------
------------------------STATS------------------------
Total number of successful requests: 99300
Total number of unsuccessful requests: 0
Mean response time : 12.2213 ms
Median response time : 10.2801 ms
99th Percentile response time : 22.3695 ms
Max response time : 290.1898 ms
Total wall time: 19137 ms
Throughput: 5188.90
-------------------------------------------------------
[ec2-user@ip-172-31-95-38 assignment02]$ java -jar client.jar testdata.txt 128
All 128 threads processing completed!
-------------------------------------------------------
------------------------STATS------------------------
Total number of successful requests: 99300
Total number of unsuccessful requests: 0
Mean response time : 25.7468 ms
Median response time : 23.4388 ms
99th Percentile response time : 52.9228 ms
Max response time : 309.8277 ms
Total wall time: 20175 ms
Throughput: 4921.93
-------------------------------------------------------
[ec2-user@ip-172-31-95-38 assignment02]$ java -jar client.jar testdata.txt 256
All 256 threads processing completed!
-------------------------------------------------------
------------------------STATS------------------------
Total number of successful requests: 99300
Total number of unsuccessful requests: 0
Mean response time : 56.2258 ms
Median response time : 53.6866 ms
```

Conclusion : The throughput is very high even with small number of threads because the client is in the same data center region as the server and the latency is very low. It reaches the peak in about 32 threads and then the throughput begins to drop slowly it means the server or the client is the bottleneck. The publish rate and consume rate is nearly the same.

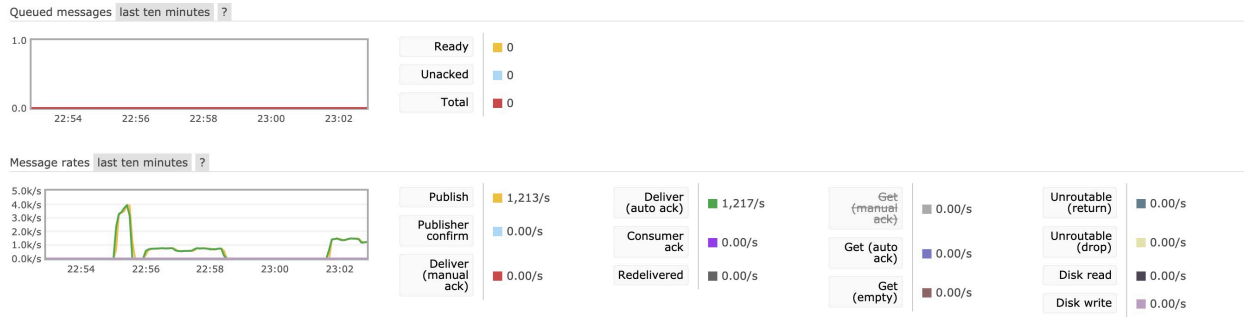
And we can see the the maximum consume rate is about 5000 / s. Thus the queue size will build up when publish rate exceeds 5000 / s.

2. 4 ec2 t2.micro sever with ELB and 5 consumer threads

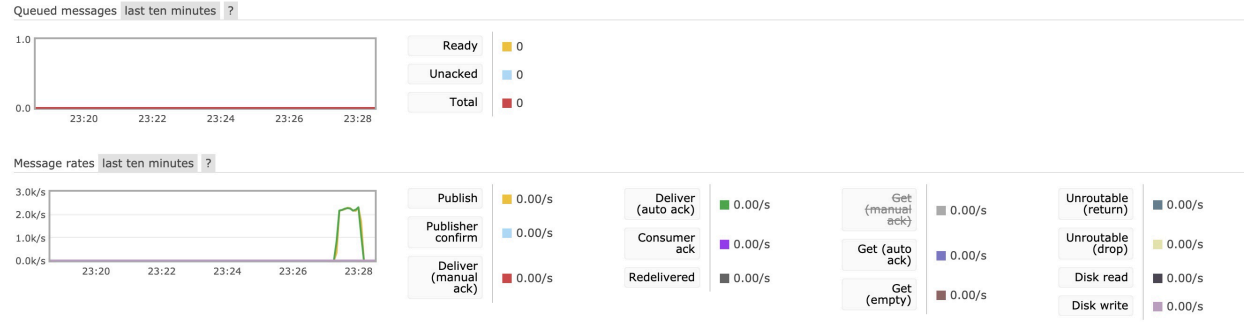
1. 2 client threads : publish rate and consume rate are about 750 / s, queue size = 0



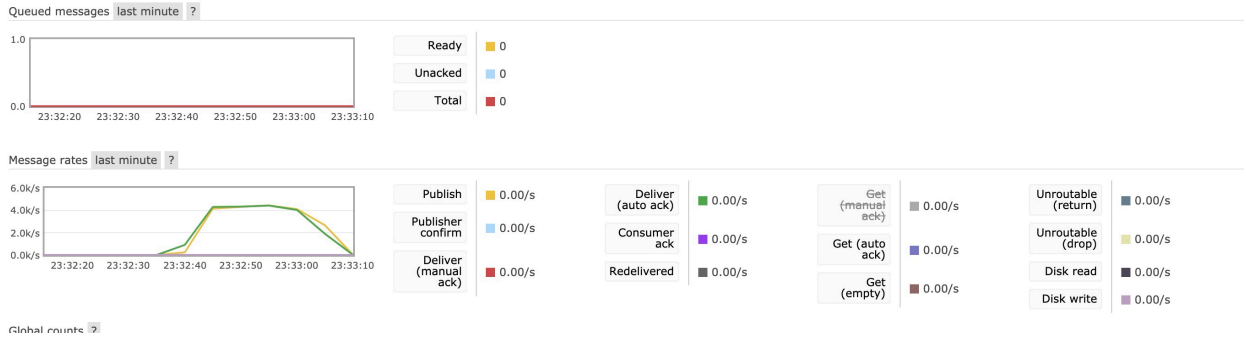
2. 4 client threads : publish rate and consume rate are about 1200 ~ 1400 / s, queue size = 0



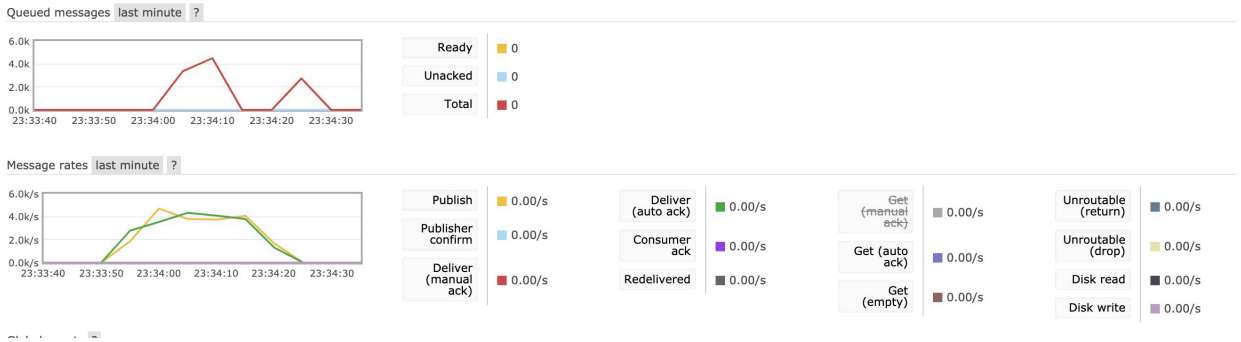
3. 8 client threads : publish rate and consume rate are about 2200 / s, queue size = 0



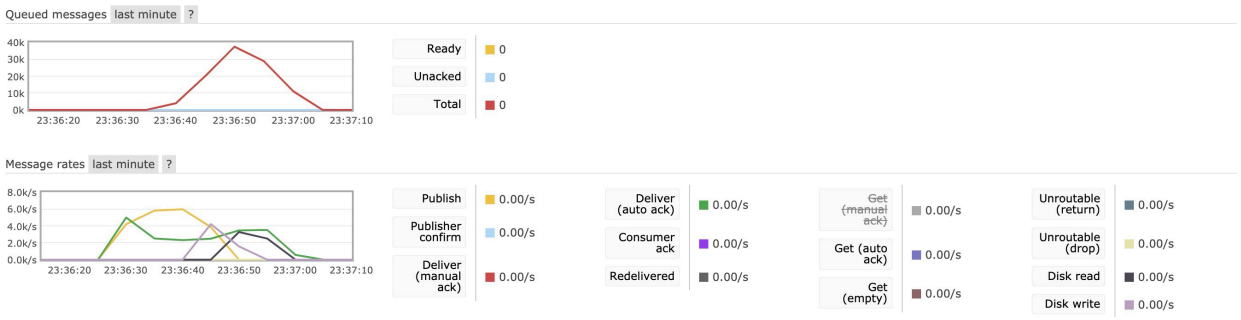
4. 16 client threads : publish rate and consume rate are about 4200/ s, queue size = 0



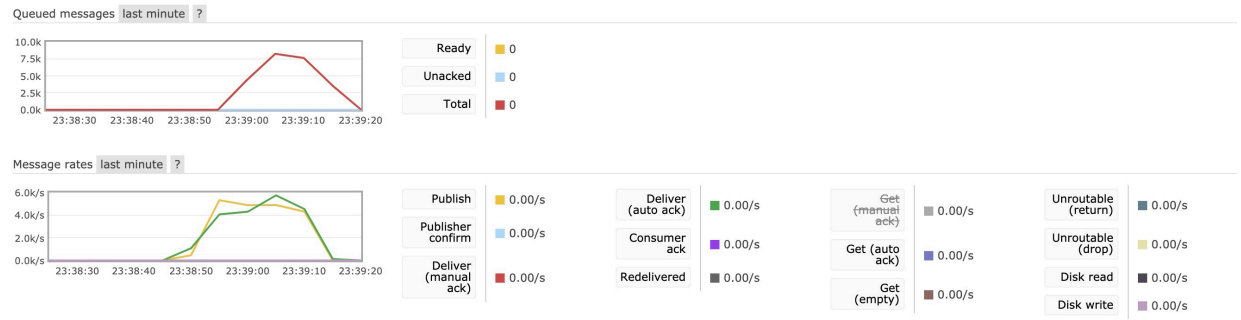
5. 32 client threads : publish rate is about 4500 / s and consume rate is slightly lower at the beginning but grows later to consume all messages queue size = 4k



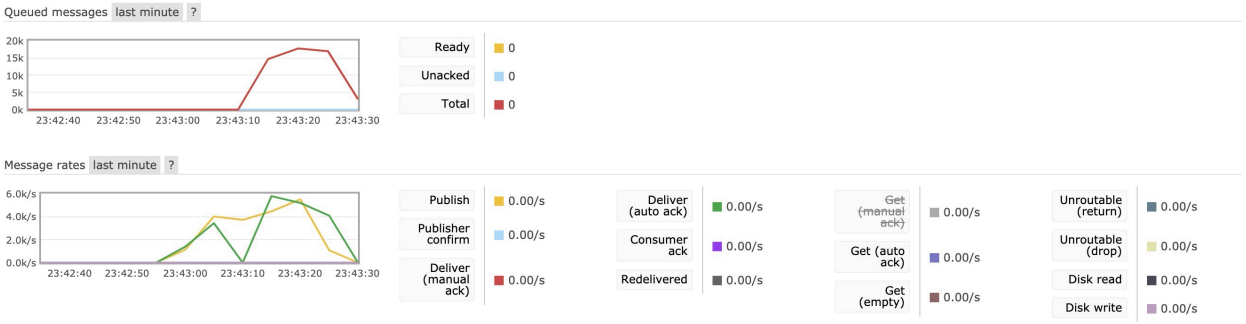
6. 64 client threads : publish rate is at most 6000 / s and there is a fluctuation in consume rate from 5000 to 3000 / s, queue size = 40k



7. 128 client threads : publish rate and consume rate are about 5000 /s, queue size = 8k



8. 256 client threads : publish rate and consume rate are fluctuated from 4000 - 6000/ s, queue size = 18k

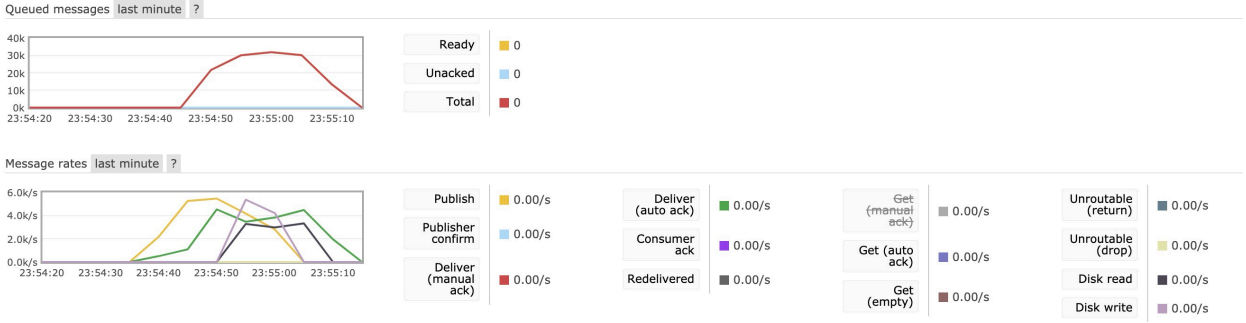


```
[ec2-user@ip-172-31-95-38 assignment02]$ java -jar client-lbs.jar testdata.txt 2
All 2 threads processing completed!
----------------------------------------------------
------------------------STATS-----------------------
Total number of successful requests: 99300
Total number of unsuccessful requests: 0
Mean response time : 2.9273 ms
Median response time : 2.8200 ms
99th Percentile response time : 4.9726 ms
Max response time : 237.4108 ms
Total wall time: 146374 ms
Throughput: 678.40
----------------------------------------------------
[ec2-user@ip-172-31-95-38 assignment02]$ java -jar client-lbs.jar testdata.txt 4
All 4 threads processing completed!
----------------------------------------------------
------------------------STATS-----------------------
Total number of successful requests: 99300
Total number of unsuccessful requests: 0
Mean response time : 2.9048 ms
Median response time : 2.7917 ms
99th Percentile response time : 4.8012 ms
Max response time : 245.6380 ms
Total wall time: 72688 ms
Throughput: 1366.11
----------------------------------------------------
[ec2-user@ip-172-31-95-38 assignment02]$ java -jar client-lbs.jar testdata.txt 8
All 8 threads processing completed!
----------------------------------------------------
------------------------STATS-----------------------
Total number of successful requests: 99300
Total number of unsuccessful requests: 0
Mean response time : 3.7513 ms
Median response time : 3.5733 ms
99th Percentile response time : 6.7277 ms
Max response time : 317.9946 ms
Total wall time: 46919 ms
Throughput: 2116.41
----------------------------------------------------
[ec2-user@ip-172-31-95-38 assignment02]$ java -jar client-lbs.jar testdata.txt 16
All 16 threads processing completed!
----------------------------------------------------
------------------------STATS-----------------------
Total number of successful requests: 99300
Total number of unsuccessful requests: 0
Mean response time : 3.8336 ms
Median response time : 3.6426 ms
99th Percentile response time : 7.8601 ms
Max response time : 246.5030 ms
Total wall time: 24030 ms
Throughput: 4132.33
----------------------------------------------------
[ec2-user@ip-172-31-95-38 assignment02]$ java -jar client-lbs.jar testdata.txt 32
All 32 threads processing completed!
----------------------------------------------------
------------------------STATS-----------------------
Total number of successful requests: 99300
Total number of unsuccessful requests: 0
Mean response time : 7.6752 ms
Median response time : 7.7545 ms
99th Percentile response time : 12.9062 ms
Max response time : 260.8114 ms
Total wall time: 24008 ms
Throughput: 4136.12
----------------------------------------------------
[ec2-user@ip-172-31-95-38 assignment02]$ java -jar client-lbs.jar testdata.txt 64
All 64 threads processing completed!
----------------------------------------------------
------------------------STATS-----------------------
Total number of successful requests: 99300
Total number of unsuccessful requests: 0
Mean response time : 11.1766 ms
Median response time : 10.5602 ms
99th Percentile response time : 23.4842 ms
Max response time : 335.2349 ms
Total wall time: 17521 ms
Throughput: 5667.48
----------------------------------------------------
[ec2-user@ip-172-31-95-38 assignment02]$ java -jar client-lbs.jar testdata.txt 128
All 128 threads processing completed!
----------------------------------------------------
------------------------STATS-----------------------
Total number of successful requests: 99300
Total number of unsuccessful requests: 0
Mean response time : 26.3463 ms
Median response time : 25.1965 ms
99th Percentile response time : 49.9433 ms
Max response time : 393.5224 ms
Total wall time: 20619 ms
Throughput: 4815.95
----------------------------------------------------
[ec2-user@ip-172-31-95-38 assignment02]$ java -jar client-lbs.jar testdata.txt 256
All 256 threads processing completed!
----------------------------------------------------
------------------------STATS-----------------------
Total number of successful requests: 99300
Total number of unsuccessful requests: 0
Mean response time : 58.4122 ms
Median response time : 59.0084 ms
99th Percentile response time : 80.7685 ms
Max response time : 481.2757 ms
Total wall time: 22891 ms
Throughput: 4337.95
----------------------------------------------------
```

Conclusion: As we can see using ELB with 4 free-tier ec2 instance doesn't improve our performance and it even decrease the throughput a little because the server is not the bottleneck and sending requests to ELB and letting it forward to servers just increase the latency. So this results is reasonable.

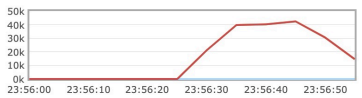
3. How many consumer threads to keep rabbit message queue size to 0 (32 threads client)

1. Consumer with 10 threads



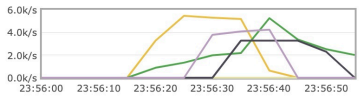
2. Consumer with 20 threads

Queued messages last minute ?



Ready	4,850
Unacked	0
Total	4,850

Message rates last minute ?



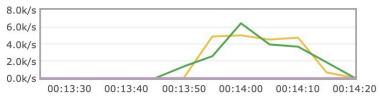
Publish	0.00/s	Deliver (auto ack)	2,010/s	~~Get (manual ack)~~	0.00/s	Unroutable (return)	0.00/s
Publisher confirm	0.00/s	Consumer ack	0.00/s	Get (auto ack)	0.00/s	Unroutable (drop)	0.00/s
Deliver (manual ack)	0.00/s	Redelivered	0.00/s	Get (empty)	0.00/s	Disk read	0.00/s
						Disk write	0.00/s

3. Consumer with 30 threads

Queued messages last minute ?

Ready	0
Unacked	0
Total	0

Message rates last minute ?

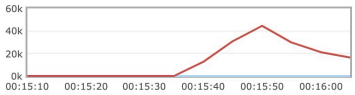


Publish	0.00/s	Deliver (auto ack)	0.00/s	~~Get (manual ack)~~	0.00/s	Unroutable (return)	0.00/s
Publisher confirm	0.00/s	Consumer ack	0.00/s	Get (auto ack)	0.00/s	Unroutable (drop)	0.00/s
Deliver (manual ack)	0.00/s	Redelivered	0.00/s	Get (empty)	0.00/s	Disk read	0.00/s
						Disk write	0.00/s

Global counts ?

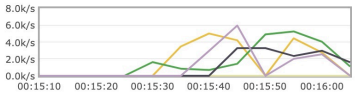
4. Consumer with 40 threads

Queued messages last minute ?



Ready	0
Unacked	0
Total	0

Message rates last minute ?



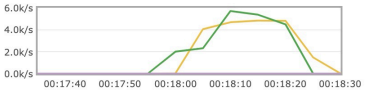
Publish	0.00/s	Deliver (auto ack)	1,100/s	~~Get (manual ack)~~	0.00/s	Unroutable (return)	0.00/s
Publisher confirm	0.00/s	Consumer ack	0.00/s	Get (auto ack)	0.00/s	Unroutable (drop)	0.00/s
Deliver (manual ack)	0.00/s	Redelivered	0.00/s	Get (empty)	0.00/s	Disk read	1,600/s
						Disk write	0.00/s

5. Consumer with 50 threads

Queued messages last minute ?

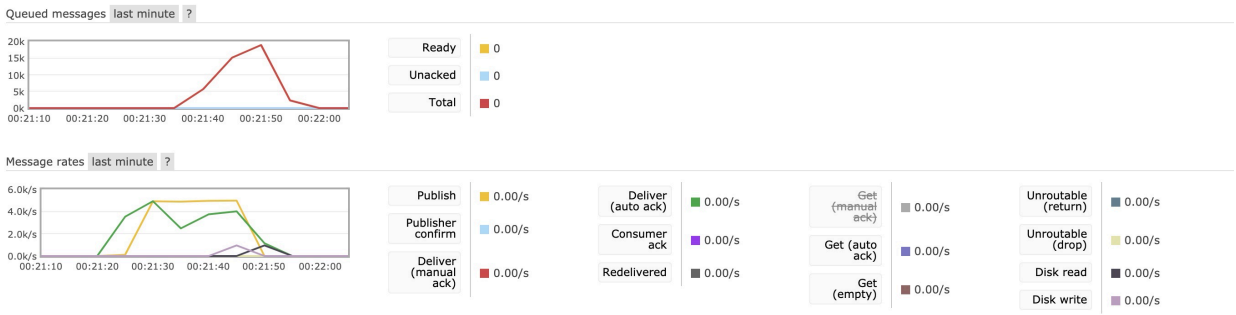
Ready	0
Unacked	0
Total	0

Message rates last minute ?



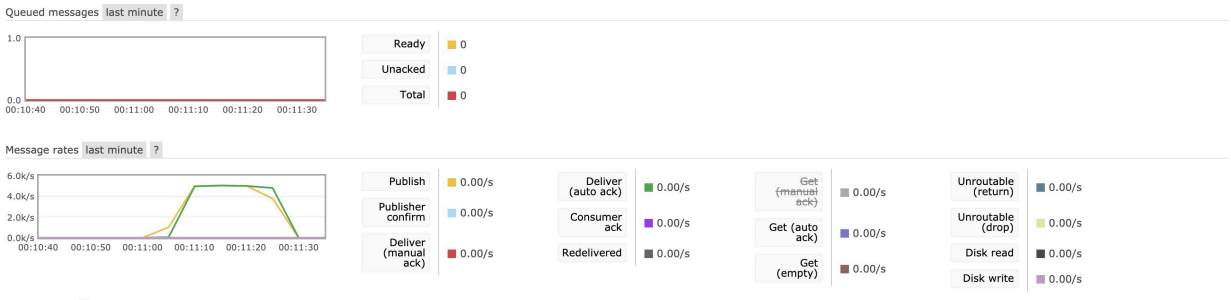
Publish	0.00/s	Deliver (auto ack)	0.00/s	~~Get (manual ack)~~	0.00/s	Unroutable (return)	0.00/s
Publisher confirm	0.00/s	Consumer ack	0.00/s	Get (auto ack)	0.00/s	Unroutable (drop)	0.00/s
Deliver (manual ack)	0.00/s	Redelivered	0.00/s	Get (empty)	0.00/s	Disk read	0.00/s
						Disk write	0.00/s

6. Consumer with 60 threads



7. Consumer with 70 threads

8. Consumer with 80 threads

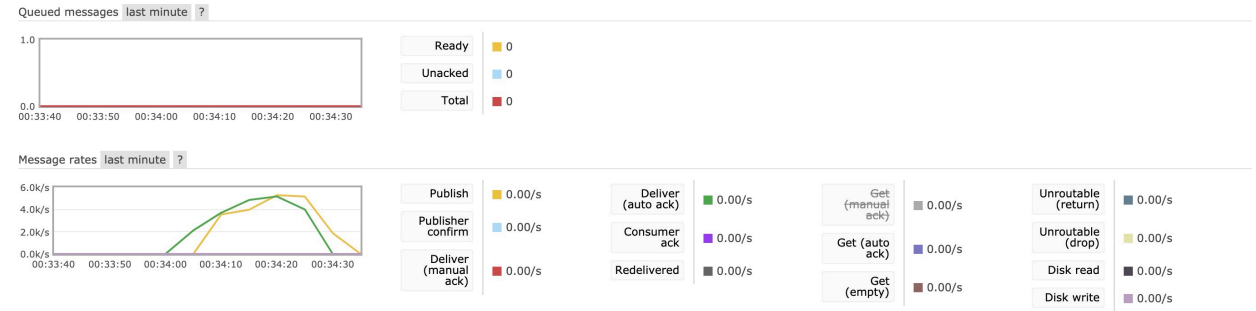



```
(base) junguo@GJs-MBP16 ~ % java -jar consumers-jar-with-dependencies.jar 10
^C%
(base) junguo@GJs-MBP16 ~ % java -jar consumers-jar-with-dependencies.jar 20
^C%
(base) junguo@GJs-MBP16 ~ % java -jar consumers-jar-with-dependencies.jar 40
^C%
(base) junguo@GJs-MBP16 ~ % java -jar consumers-jar-with-dependencies.jar 128
^C%
(base) junguo@GJs-MBP16 ~ % java -jar consumers-jar-with-dependencies.jar 30
^C%
(base) junguo@GJs-MBP16 ~ % java -jar consumers-jar-with-dependencies.jar 40
^[[A^C%
(base) junguo@GJs-MBP16 ~ % java -jar consumers-jar-with-dependencies.jar 50
^C%
(base) junguo@GJs-MBP16 ~ % java -jar consumers-jar-with-dependencies.jar 60
^C%
(base) junguo@GJs-MBP16 ~ % java -jar consumers-jar-with-dependencies.jar 70
^C%
(base) junguo@GJs-MBP16 ~ % java -jar consumers-jar-with-dependencies.jar 80
^C%
(base) junguo@GJs-MBP16 ~ %
```

It does seem that 80 threads is enough to keep queue size to 0. Why we need so many threads to ensure the queue size of 0. Why it is that? I think it is because I run consumers in my local machine which have like 80 ms latency to consume messages. And when messages are published to the queue, it takes at least 160 ms to consume the message. And during this period the queued messages has already built up.

If I try to deploy consumer on the instance in the same region with the rabbitmq, I get the following results.

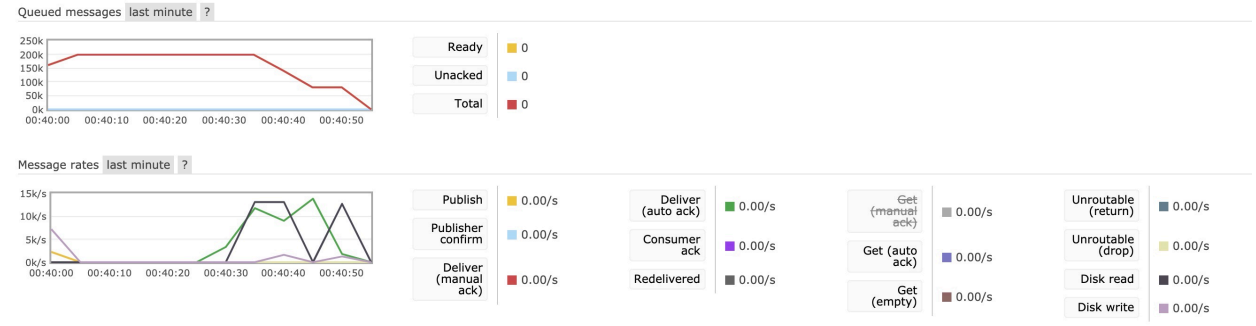
1. Consumer with 1 threads.




```
[ec2-user@ip-172-31-18-97 ~]$ java -jar consumer.jar 1
```

The consumer overwhelms the producer with only one single threads. It proves that reducing the physical latency can help increase throughput.

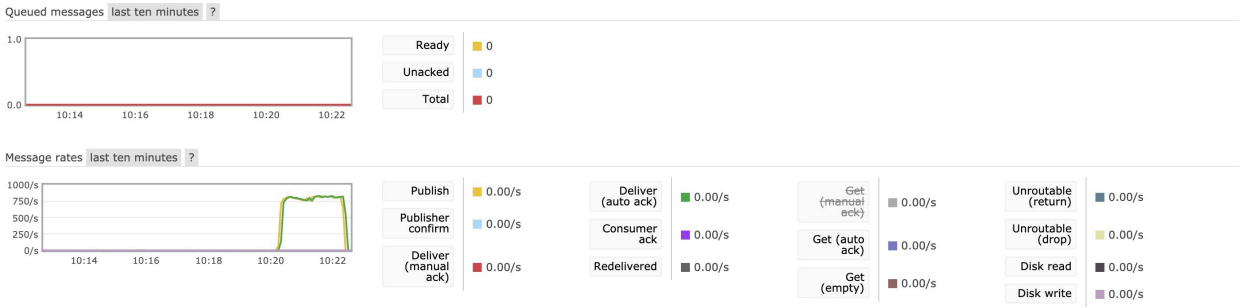
And then I stop the consumer to let the messages accumulated to see how fast the consumer can be. It may not be the real speed of consuming messages because our non-persistent queue can't hold so many messages in memory it will store these overwhelming messages to disk and let consumer consume from disk which is slower than retrieving from memory.



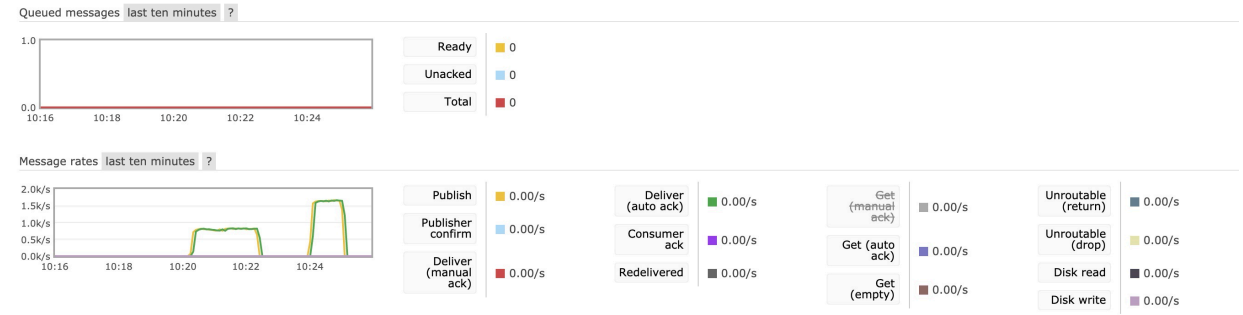
As you can see from the above graph, the single thread has a peak speed of nearly 15k / s which is pretty fast.

4.Single ec2 t2.micro sever with t2.large client and persistent rabbit message queue with 80 threads consumer

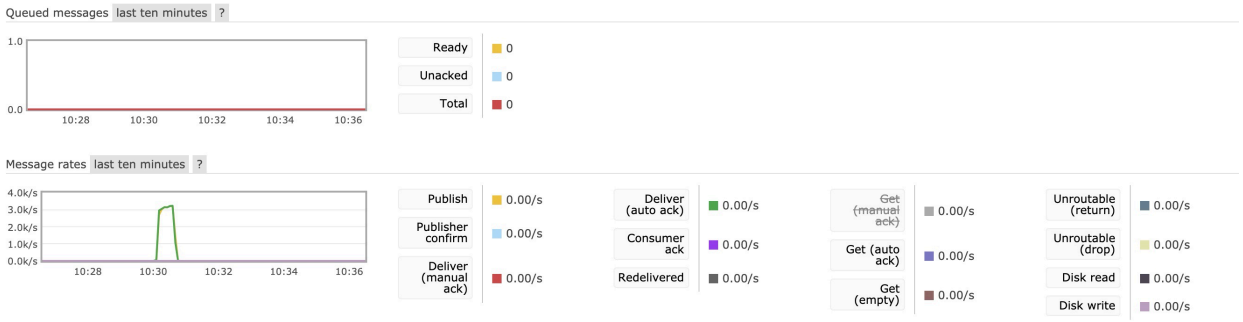
1. 2 client threads : publish rate and consume rate are about 800 / s, queue size = 0



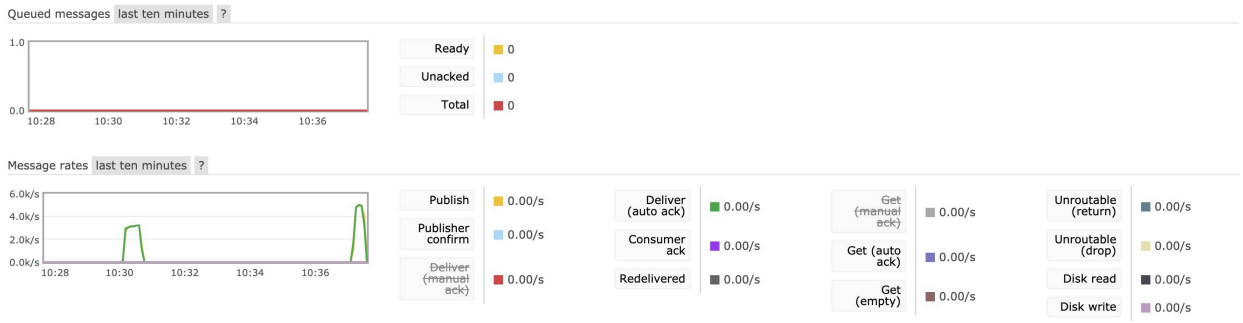
2. 4 client threads : publish rate and consume rate are about 1600/ s, queue size = 0



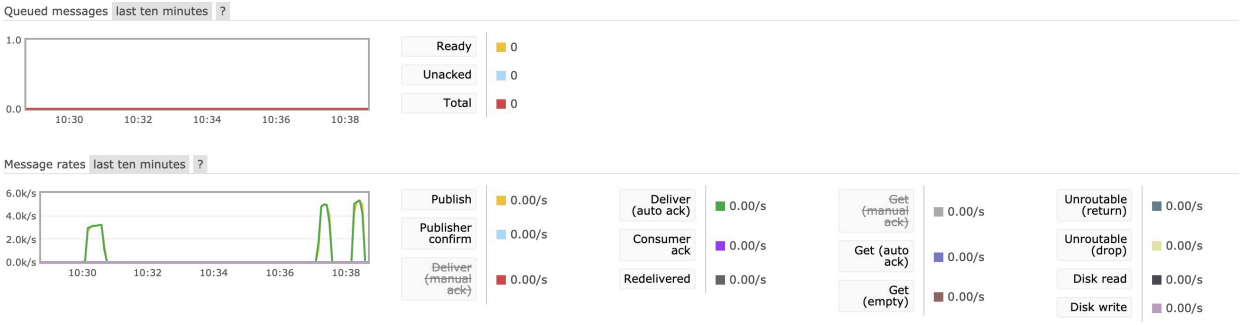
3. 8 client threads : publish rate and consume rate are about 3100 / s, queue size = 0



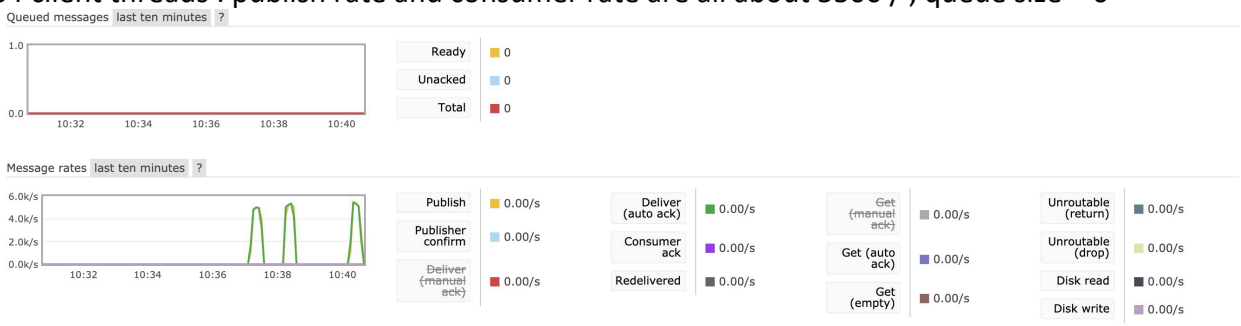
4. 16 client threads : publish rate and consume rate are about 5000/ s, queue size = 0



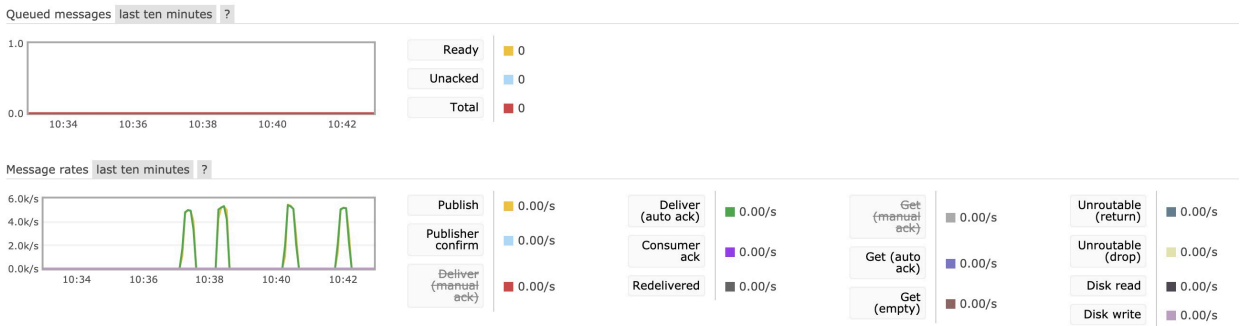
5. 32 client threads : publish rate and consumer rate are all about 5500 / , queue size = 0



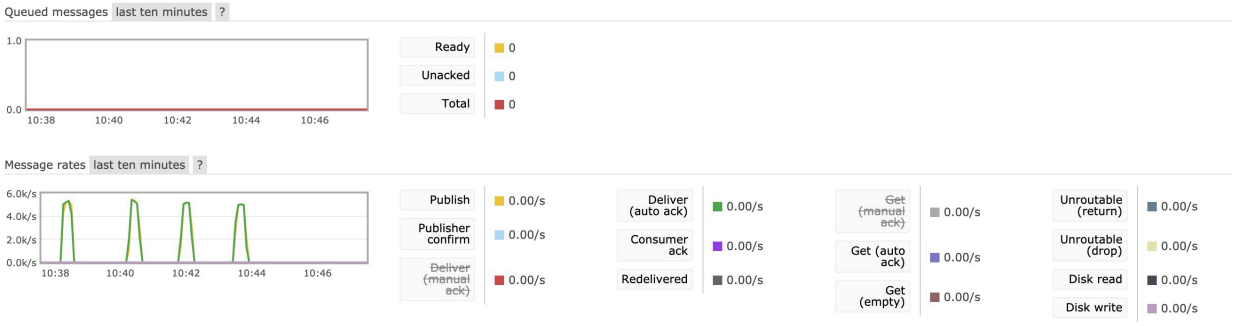
6. 64 client threads : publish rate and consumer rate are all about 5500 / , queue size = 0



7. 128 client threads : publish rate and consume rate are about 5000 /s, queue size = 0



8. 256 client threads : publish rate and consume rate are fluctuated from 5000 / s, queue size = 0



```
[ec2-user@ip-172-31-95-38 assignment02]$ java -jar client.jar testdata.txt 2
All 2 threads processing completed!
----------------------------------------------------
------------------------STATS-----------------------
Total number of successful requests: 99300
Total number of unsuccessful requests: 0
Mean response time : 2.5495 ms
Median response time : 2.4419 ms
99th Percentile response time : 4.6551 ms
Max response time : 231.0187 ms
Total wall time: 127594 ms
Throughput: 778.25
----------------------------------------------------
```

```
[ec2-user@ip-172-31-95-38 assignment02]$ java -jar client.jar testdata.txt 4
All 4 threads processing completed!
----------------------------------------------------
------------------------STATS-----------------------
Total number of successful requests: 99300
Total number of unsuccessful requests: 0
Mean response time : 2.4938 ms
Median response time : 2.4238 ms
99th Percentile response time : 3.5266 ms
Max response time : 233.8202 ms
Total wall time: 62475 ms
Throughput: 1589.44
----------------------------------------------------
```

```
[ec2-user@ip-172-31-95-38 assignment02]$ java -jar client.jar testdata.txt 8
All 8 threads processing completed!
----------------------------------------------------
------------------------STATS-----------------------
Total number of successful requests: 99300
Total number of unsuccessful requests: 0
Mean response time : 2.5705 ms
Median response time : 2.4653 ms
99th Percentile response time : 3.8868 ms
Max response time : 243.2054 ms
Total wall time: 32271 ms
Throughput: 3077.07
----------------------------------------------------
```

```
[ec2-user@ip-172-31-95-38 assignment02]$ java -jar client.jar testdata.txt 16
All 16 threads processing completed!
----------------------------------------------------
------------------------STATS-----------------------
Total number of successful requests: 99300
Total number of unsuccessful requests: 0
Mean response time : 3.3337 ms
Median response time : 3.1922 ms
99th Percentile response time : 5.9255 ms
Max response time : 241.8900 ms
Total wall time: 20944 ms
Throughput: 4741.21
----------------------------------------------------
```

```
[ec2-user@ip-172-31-95-38 assignment02]$ java -jar client.jar testdata.txt 32
All 32 threads processing completed!
----------------------------------------------------
------------------------STATS-----------------------
Total number of successful requests: 99300
Total number of unsuccessful requests: 0
Mean response time : 6.1805 ms
Median response time : 6.2655 ms
99th Percentile response time : 9.6213 ms
Max response time : 255.0060 ms
Total wall time: 19372 ms
Throughput: 5125.95
----------------------------------------------------
```

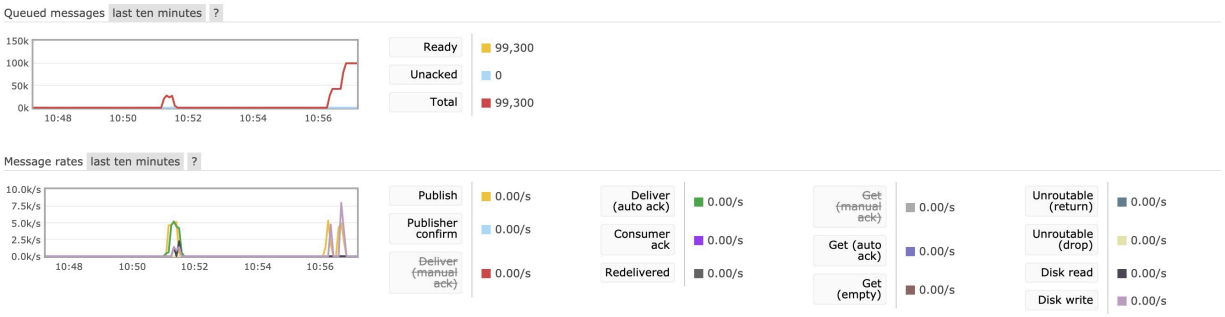
```
[ec2-user@ip-172-31-95-38 assignment02]$ java -jar client.jar testdata.txt 64
All 64 threads processing completed!
----------------------------------------------------
------------------------STATS-----------------------
Total number of successful requests: 99300
Total number of unsuccessful requests: 0
Mean response time : 12.2420 ms
Median response time : 10.8618 ms
99th Percentile response time : 21.4538 ms
Max response time : 272.1799 ms
Total wall time: 19174 ms
Throughput: 5178.89
----------------------------------------------------
```

```
[ec2-user@ip-172-31-95-38 assignment02]$ java -jar client.jar testdata.txt 128
All 128 threads processing completed!
----------------------------------------------------
------------------------STATS-----------------------
Total number of successful requests: 99300
Total number of unsuccessful requests: 0
Mean response time : 24.1067 ms
Median response time : 21.5340 ms
99th Percentile response time : 45.0051 ms
Max response time : 374.4581 ms
Total wall time: 18908 ms
Throughput: 5251.75
----------------------------------------------------
```

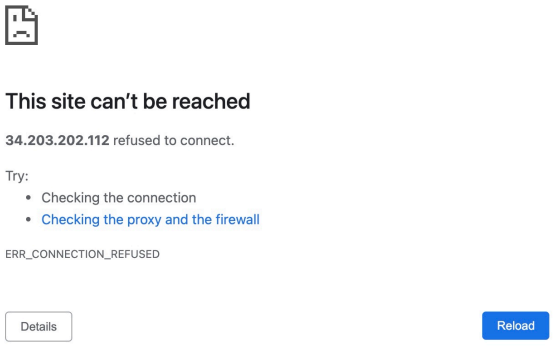
```
[ec2-user@ip-172-31-95-38 assignment02]$ java -jar client.jar testdata.txt 256
All 256 threads processing completed!
----------------------------------------------------
------------------------STATS-----------------------
Total number of successful requests: 99300
Total number of unsuccessful requests: 0
Mean response time : 51.5130 ms
Median response time : 49.2163 ms
99th Percentile response time : 102.5509 ms
Max response time : 391.8398 ms
Total wall time: 20237 ms
Throughput: 4906.85
----------------------------------------------------
```

Conclusion : I don’t see much difference in the throughputs between using non-persistent queue and persistent queue except the queue size is different because in the implementation of persistent queue, I use the optimized consumer threads of 80 to ensure that the consumer can overwhelm the producers. But when I shutdown the broker in rabbitmq server, the persistent-queue keep the ability to recover when restarting the broker.

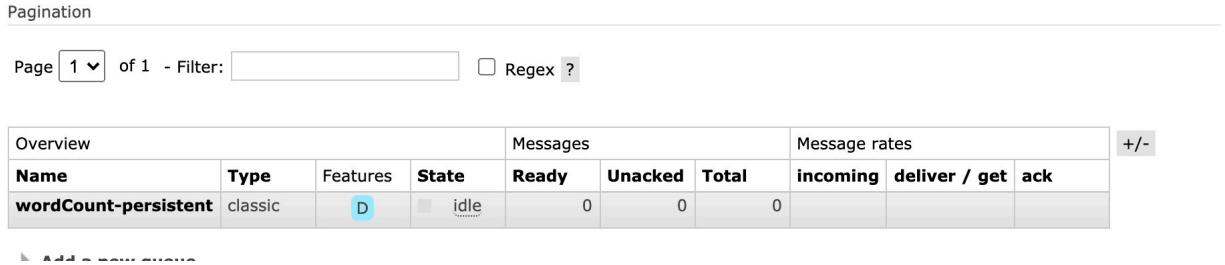
Before shutting down



Then shutting down



After restarting



Noticeably, the wordcount (non-persistent) queue is gone but wordcount-persistent is recreated automatically and alive. However, the messages are all gone. To recover all messages you need to set the publishing message as durable too.

Bonus: Run client with 512 threads

```
[ec2-user@ip-172-31-95-38 assignment02]$ java -jar client-lbs.jar testdata.txt 512
All 512 threads processing completed!
--------------------------------------------------
------------------------STATS-------------------------
Total number of successful requests: 99300
Total number of unsuccessful requests: 0
Mean response time : 109.7472 ms
Median response time : 99.0210 ms
99th Percentile response time : 173.9628 ms
Max response time : 632.1084 ms
Total wall time: 21609 ms
Throughput: 4595.31
--------------------------------------------------
```

Not surprisingly, the throughput is not going to increase because of the switch contention and it hit the bottleneck of the client. We need to upgrade our client. And the consumer is fast enough to consume the messages.