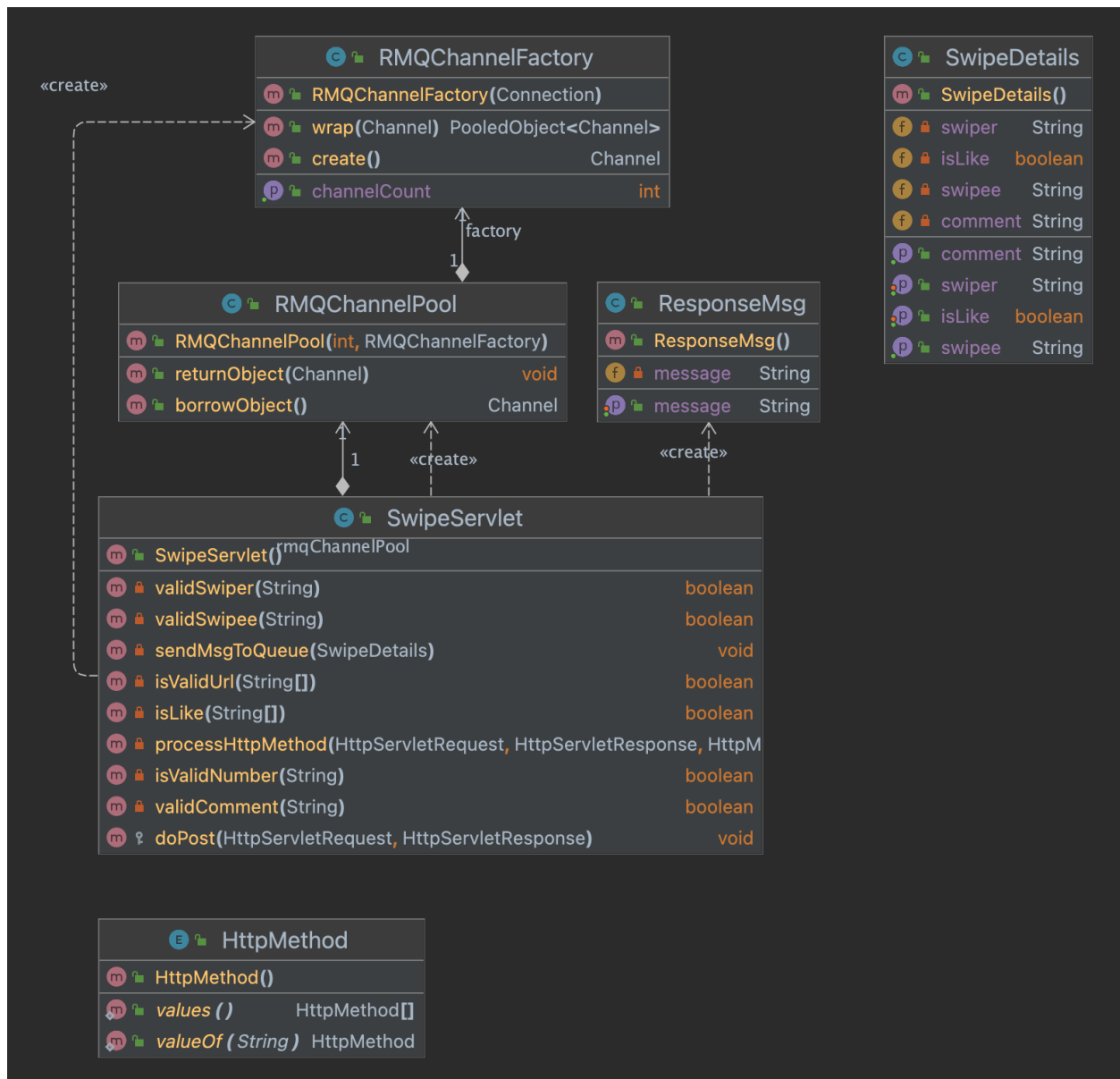


## 1. Assignment 2 repo

<https://github.com/Oliver1024/DistributedSystemDatingApp/tree/main/assignment2>

## 2. A 1-2 page description of your server design. Include major classes, packages, relationships, how messages get sent/received, etc



**RMQChannelFactory:**

The RMQChannelFactory class is a factory for creating Channel objects that are used to communicate with a RabbitMQ server, which implements the BasePooledObjectFactory interface provided by Apache Commons Pool 2, which defines a set of methods for creating, validating, and destroying objects that are managed by an object pool. And the the RMQChannelFactory class has a single constructor that takes a Connection object as a parameter. This Connection object represents a valid connection to a RabbitMQ server, and is used to create new Channel objects.

**RMQChannelPool:**

The RMQChannelPool class is designed to be used by multiple threads simultaneously, so it uses a BlockingQueue to ensure thread-safety. The borrowObject() method blocks until a channel is available, while the returnObject() method adds the channel back to the pool for other threads to use. The purpose of this class is to provide a simple way to manage a fixed number of RabbitMQ channels in a multi-threaded environment, which can improve performance and scalability of applications that use RabbitMQ.

**SwipeServlet:**

In my SwipeServlet class, the new feature I added is that the SwiperServlet class determines whether the swipe was a "like" or "dislike" and sends the swipe data to a RabbitMQ message queue using the RabbitMQ Java client library. In the constructor, a connection is established with a RabbitMQ server using the "ConnectionFactory" class. The RabbitMQ server's IP address, virtual host, and login credentials are set in the ConnectionFactory object. A new connection is established using the ConnectionFactory object, and a new channel factory is created using the Connection object. The RMQChannelPool object is then instantiated using the channel factory object and a POOL\_SIZE of 20.

The **sendMsgToQueue()** method sends the swipe data to the RabbitMQ queue. It retrieves a channel from the `RMQChannelPool` and declares an exchange named "swipe\_exchange" using the channel. It then serializes the `SwipeDetails` object to JSON and publishes it to the exchange using the channel. And this method is called after valid a body based on if swipes right or left which represents true or false respectively.

In my consumer side, a connection to the RabbitMQ server is established using the `Connection` class from the RabbitMQ client library. A channel is then created using the `createChannel()` method, which is used to interact with the RabbitMQ server. The `queueDeclare()` method is used to declare a queue for the consumer to consume messages from, and the `queueBind()` method is used to bind the queue to an exchange so that messages can be routed to the queue.

The `basicConsume()` method is used to start consuming messages from the queue. When a message is received, the `DeliverCallback` that was passed to the `basicConsume()` method is invoked with the delivery details and the message body. And then the message is deserialized from JSON to a `SwipeDetails` object using the `Gson` library and the `basicAck()` method is then called to acknowledge receipt of the message and indicating that can be removed it from the queue.

### 3. Test run results (command lines showing metrics, RMQ management windows showing queue size, send/receive rates) for a single servlet showing your best throughput.

Throughput:



```

n: MultithreadClient
/Library/Java/JavaVirtualMachines/jdk-11.0.11.jdk/Contents/Home/bin/java ...
Number of successful requests: 500000
Number of fail requests: 0
walltime: 231 seconds
Total throughput: 2164 req/s

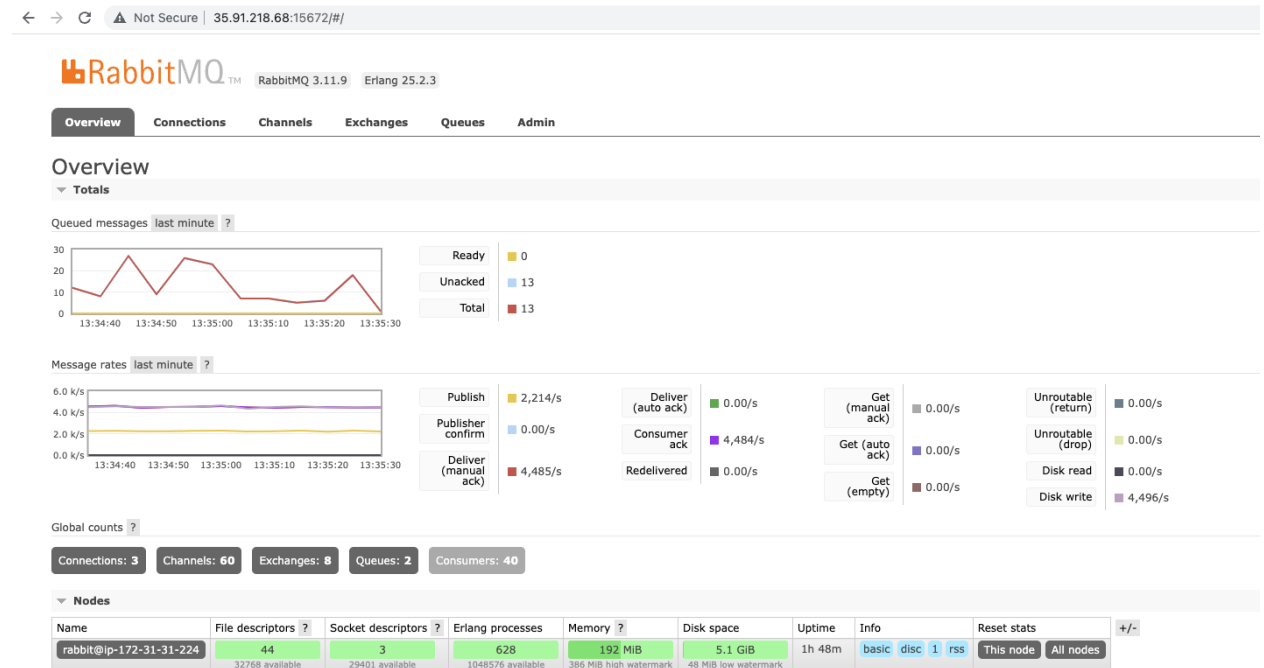
Process finished with exit code 0

```

RMQ queue size, send/receive rates:

Queue 1: about 2271/s (deliver) , 2263/s(incoming)

Queue2: about 2271/s (deliver) , 2263/s(incoming)



← → ↻ ⚠ Not Secure | 35.91.218.68:15672/#/queues

RabbitMQ™

RabbitMQ 3.11.9

Erlang 25.2.3

Overview

Connections

Channels

Exchanges

Queues

Admin

Queues

▼ All queues (2)

Pagination

Page 1 of 1 - Filter: ☐ Regex ?

Overview				Messages			Message rates			+/-
Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack	
swipe_queue_1	classic	D	running	0	18	18	2,263/s	2,271/s	2,270/s	
swipe_queue_2	classic	D	running	0	16	16	2,263/s	2,271/s	2,270/s	

► Add a new queue

HTTP API

Server Docs

Tutorials

Community Support

Community Slack

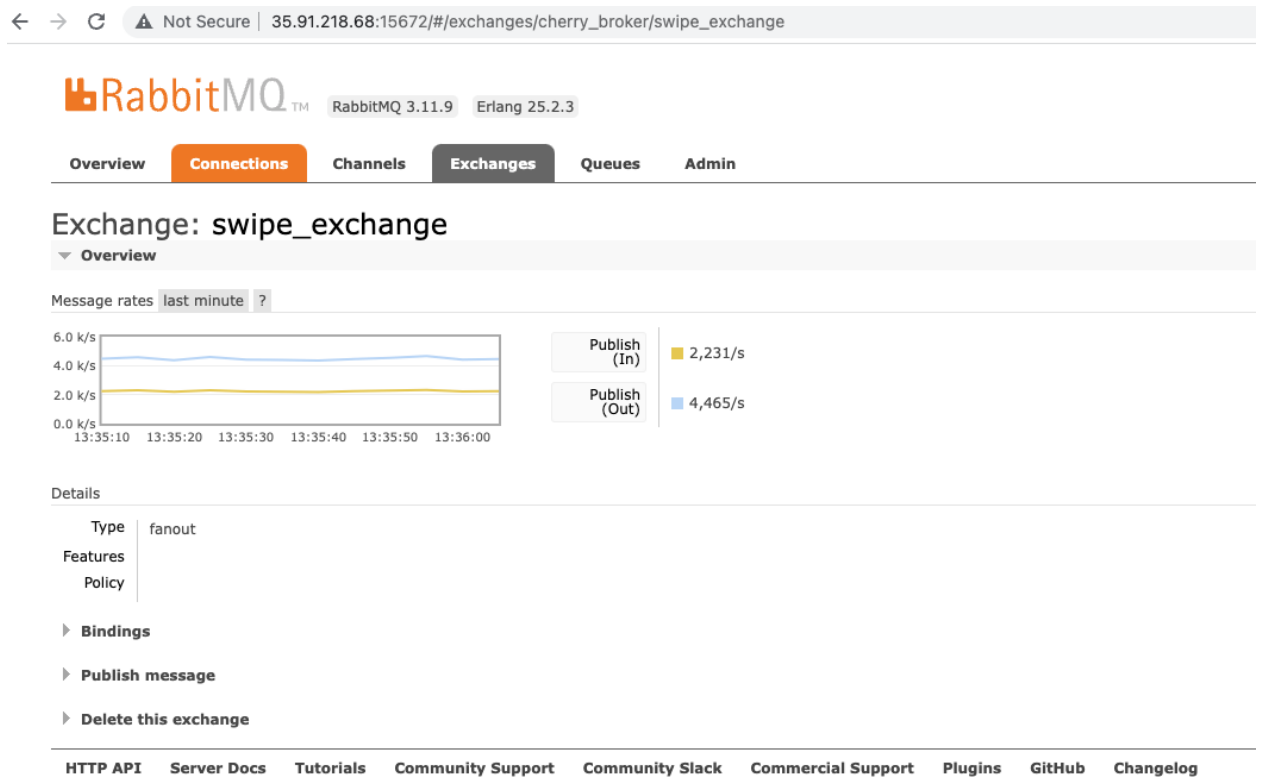
Commercial Support

Plugins

GitHub

Changelog

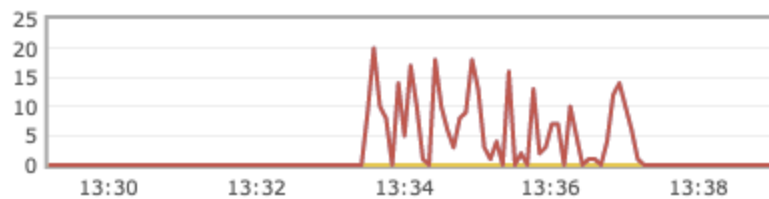
Exchange rate:



## Queue swipe\_queue\_1

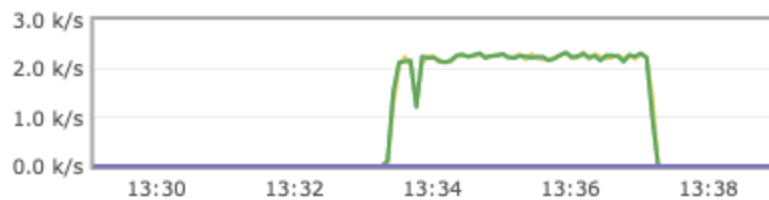
▼ Overview

Queued messages last ten minutes ?



Ready	
Unacked	
Total	

Message rates last ten minutes ?



Publish	
Deliver (manual ack)	
Deliver (auto ack)	

Details



RabbitMQ 3.11.9

Erlang 25.2.3

Overview

Connections

Channels

Exchanges

Queues

## Queue swipe\_queue\_2

▼ Overview

Queued messages last ten minutes ?

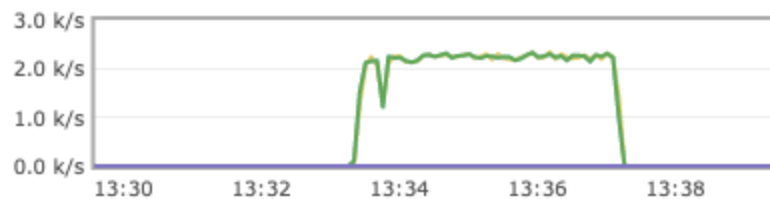


Ready

Unacked

Total

Message rates last ten minutes ?



Publish

Deliver  
(manual  
ack)

Deliver  
(auto ack)

Details

Features | durable: true

State | idle

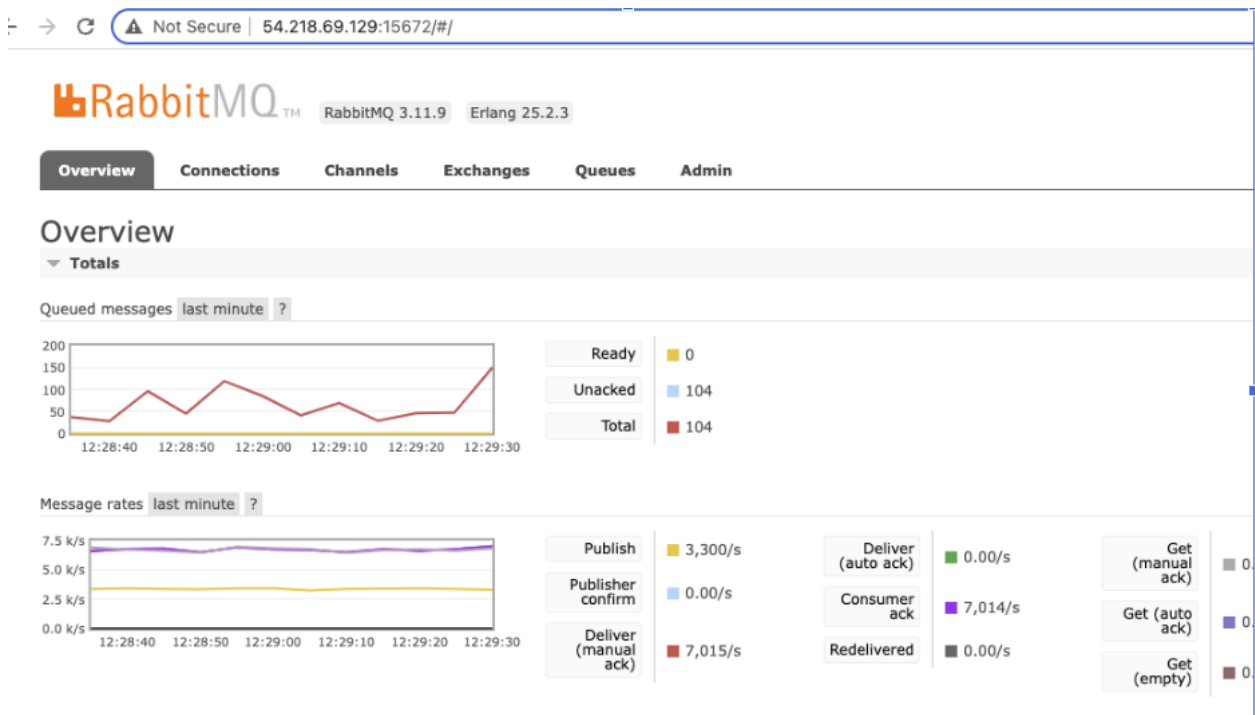
4. Test run results (command lines showing metrics, RMQ management windows showing queue size, send/receive rates) for a load balanced servlet showing your best throughput.

```
src
├── main
│   └── java
│       └── org.example
└── ...

MultiThreadClient.java
1 public class MultiThreadClient {
2     private static final int numOfThreads = 110;
3     private static AtomicInteger numSuccessfulRequests = new AtomicInteger(0);
4     private static AtomicInteger numFailedRequests = new AtomicInteger(0);
5
6     public static void main(String[] args) {
7         // ...
8     }
9 }

MultiThreadClient
/Library/Java/JavaVirtualMachines/jdk-11.0.11.jdk/Contents/Home/bin/java ...
Number of successful requests: 500000
Number of fail requests: 0
walltime: 158 seconds
Total throughput: 3164 req/s

Process finished with exit code 0
```





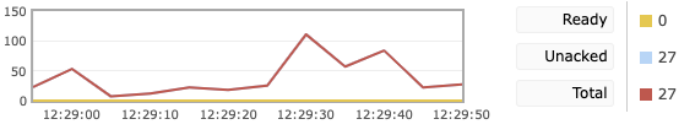
# Queue swipe\_queue\_1

▼ Overview

Queued messages

last minute

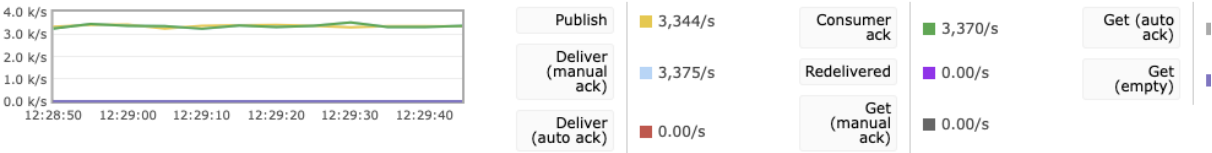
?



Message rates

last minute

?

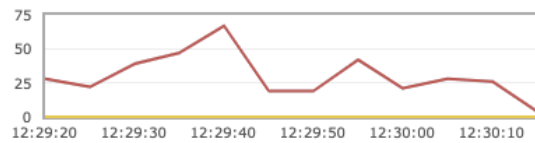


Details									
Features	durable: true	State	<div><div></div>running</div>			Total	Ready	Unacked	In
	Policy	Consumers	100	Messages ?		27	0	27	
	Operator policy	Consumer capacity ?	100%	Message body bytes ?		2.6 KiB	0 B	2.6 KiB	
	Effective policy definition			Process memory ?		4.6 MiB			

## Queue swipe\_queue\_2

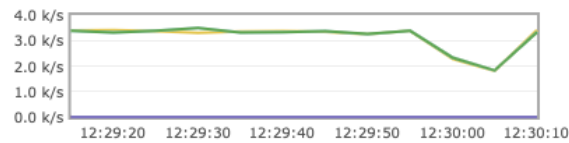
▼ Overview

Queued messages last minute ?



Ready 0  
Unacked 4  
Total 4

Message rates last minute ?



Publish 3,412/s  
Deliver (manual ack) 3,335/s  
Deliver (auto ack) 0.00/s

Consumer ack 3,324/s  
Redelivered 0.00/s  
Get (manual ack) 0.00/s

Details

Features	durable: true	State	running	Messages ?	Total	Read
Policy		Consumers	100		4	1
Operator policy		Consumer capacity ?	100%	Message body bytes ?	410 B	0 B

## Overview

### ▼ Totals

Queued messages last ten minutes ?



Ready

0

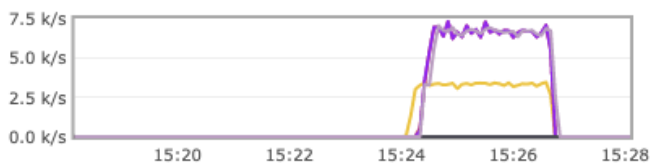
Unacked

0

Total

0

Message rates last ten minutes ?



Publish

0.00/s

Publisher confirm

0.00/s

Deliver (manual ack)

0.00/s

Deliver (auto ack)

Consumer ack

Redelivered

Global counts ?

## Queue swipe\_queue\_1

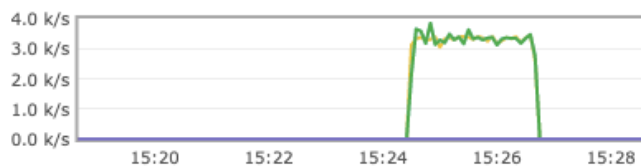
### ▼ Overview

Queued messages [last ten minutes](#) ?



Ready	0
Unacked	0
Total	0

Message rates [last ten minutes](#) ?



Publish	0.00/s
Deliver (manual ack)	0.00/s
Deliver (auto ack)	0.00/s

Consumer ack
Redelivered
Get (manual ack)

Details

## Queue swipe\_queue\_2

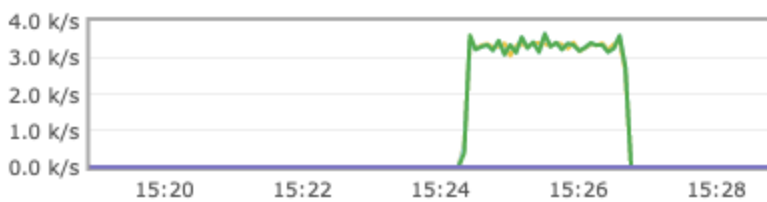
### Overview

Queued messages **last ten minutes** ?



Ready	0
Unacked	0
Total	0

Message rates **last ten minutes** ?



Publish	0.00/s
Deliver (manual ack)	0.00/s
Deliver (auto ack)	0.00/s

### Details

Features	durable: <u>true</u>	State	<u>idle</u>	
Policy		Consumers	20	
Operator policy		Consumer capacity ?	100%	
Effective policy definition				

The results clearly show that incorporating an Application Load Balancer has increased the throughput by approximately 50% compared to using a single Tomcat server. Additionally, the producer and consumer rates also increased by about 50% when compared to using a single servlet. The queue size remained consistently low in both the single servlet and ALB modes.

However, it's important to note that the throughput did not double after incorporating the ALB. This may be due to the fact that the CPU usage remained relatively low, at around 20-30% even when sending 500K requests from the client in single servlet mode. This remained consistent after incorporating the ALB, indicating that the bottleneck is not coming from the server side.

## **Tune the system & Loading test**

### ***How many client threads are optimal to maximize system throughput?***

To optimize system throughput, I conducted various tests on the number of client threads and queue consumer threads. Based on my observations, I found that around 110 client threads work best for maximizing throughput. Additionally, to keep the queue size as low as possible, I found that having 20 queue consumer threads was optimal. And the channel pool size for my producer is 30 working best.

### ***How many queue consumers threads are needed to keep the queue size as close to zero as possible?***

During the tuning process, I made adjustments to various configuration parameters such as the number of threads in the client, the channel pool size for producers, and the number of consumer threads. To accurately measure the performance, I performed tests at different times to account for external factors like AWS network fluctuation.

After experimenting with different configurations, I found the best throughput, producer, and consumer rates by adjusting one value at a time and monitoring the results. Overall, my testing helped me to identify the appropriate number of client threads, producer channel pool size, and consumer threads for optimal system performance.