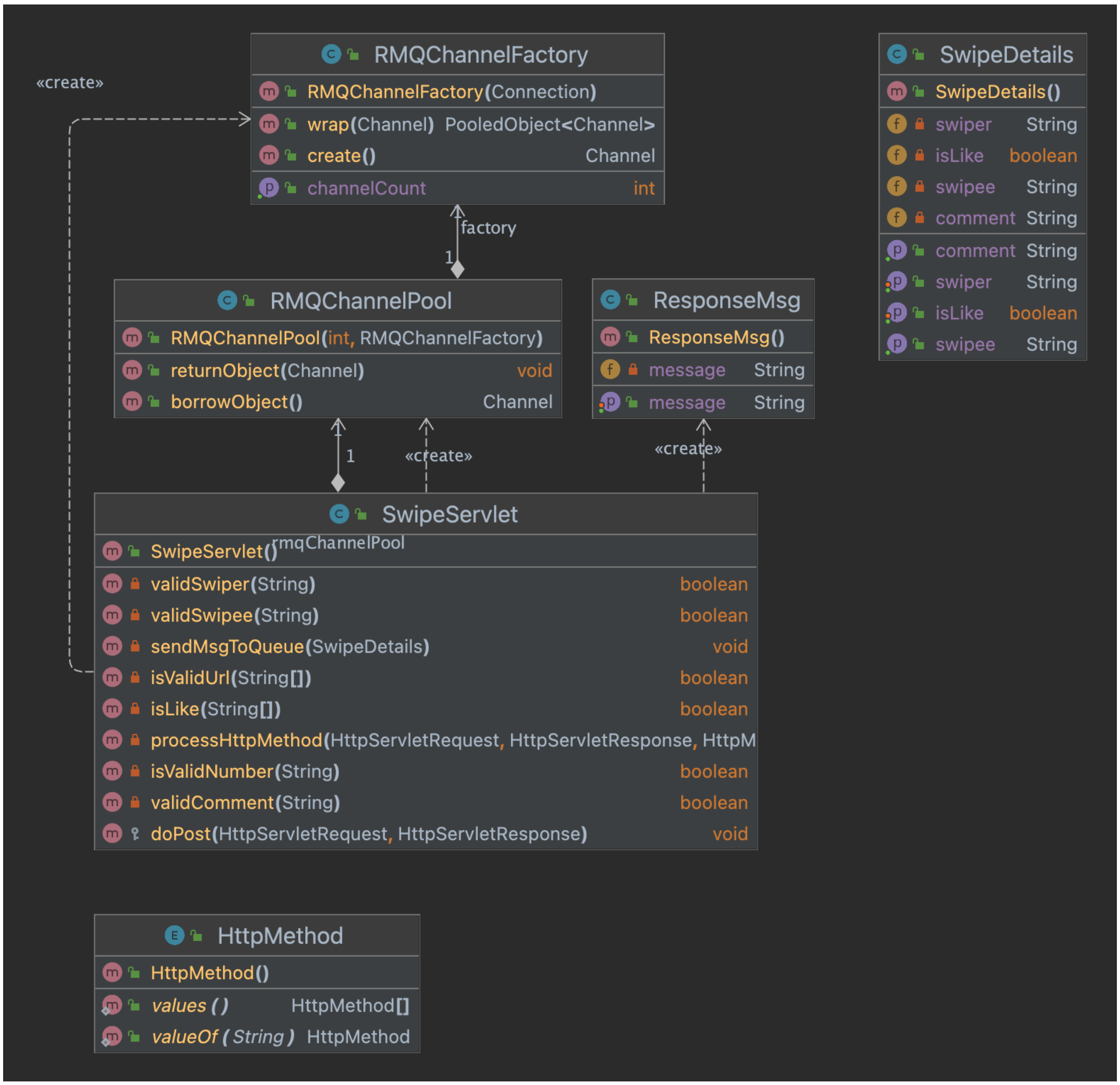
1. **Assignment 2 repo**

   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 indiciating 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.

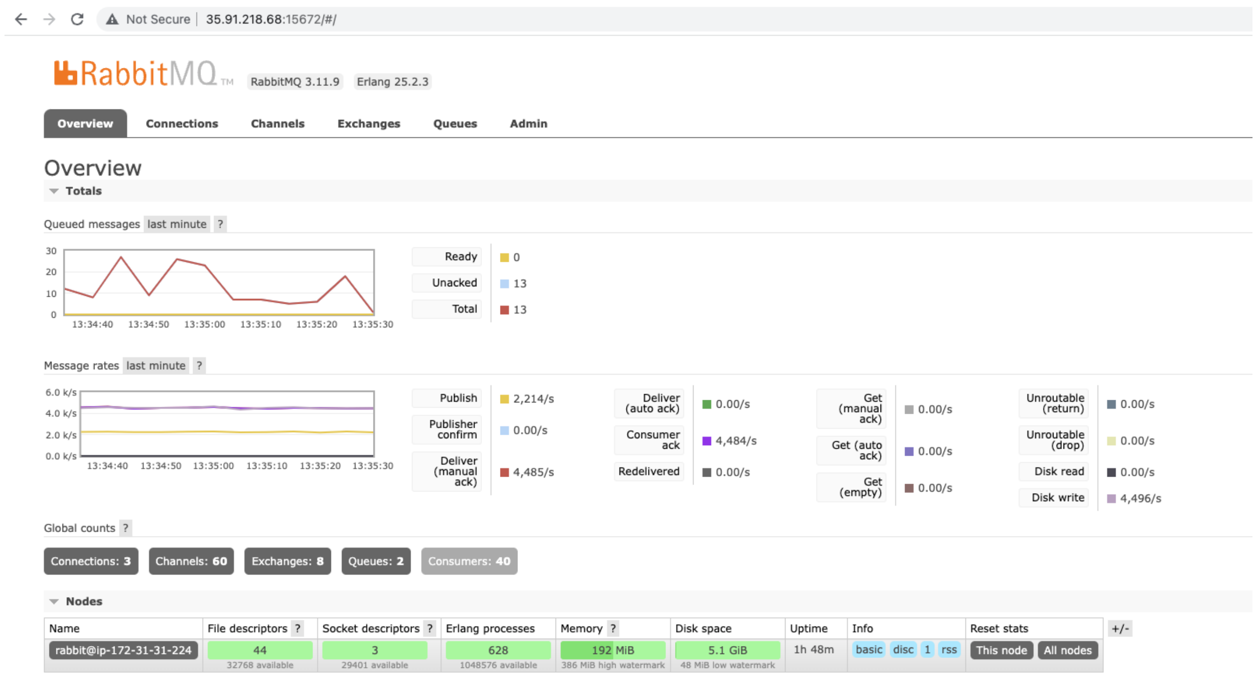
Throughput:

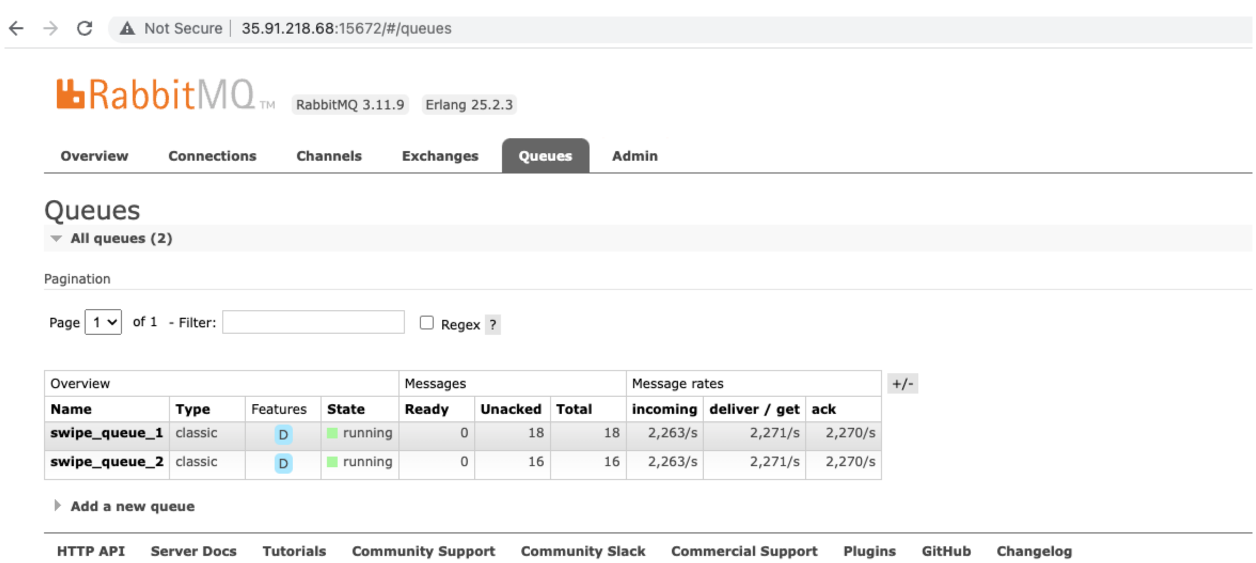
```
MultiThreadClient
/Library/Java/JavaVirtualMachines/jdk-11.0.11.jdk/Contents/Home/bin/java ...
Number of successful requests: 500000
Number of fail requests: 0
walltime: 231 seconds
Total throughput: 2164 req/s

Process finished with exit code 0
```

RMQ queue size, send/receive rates:

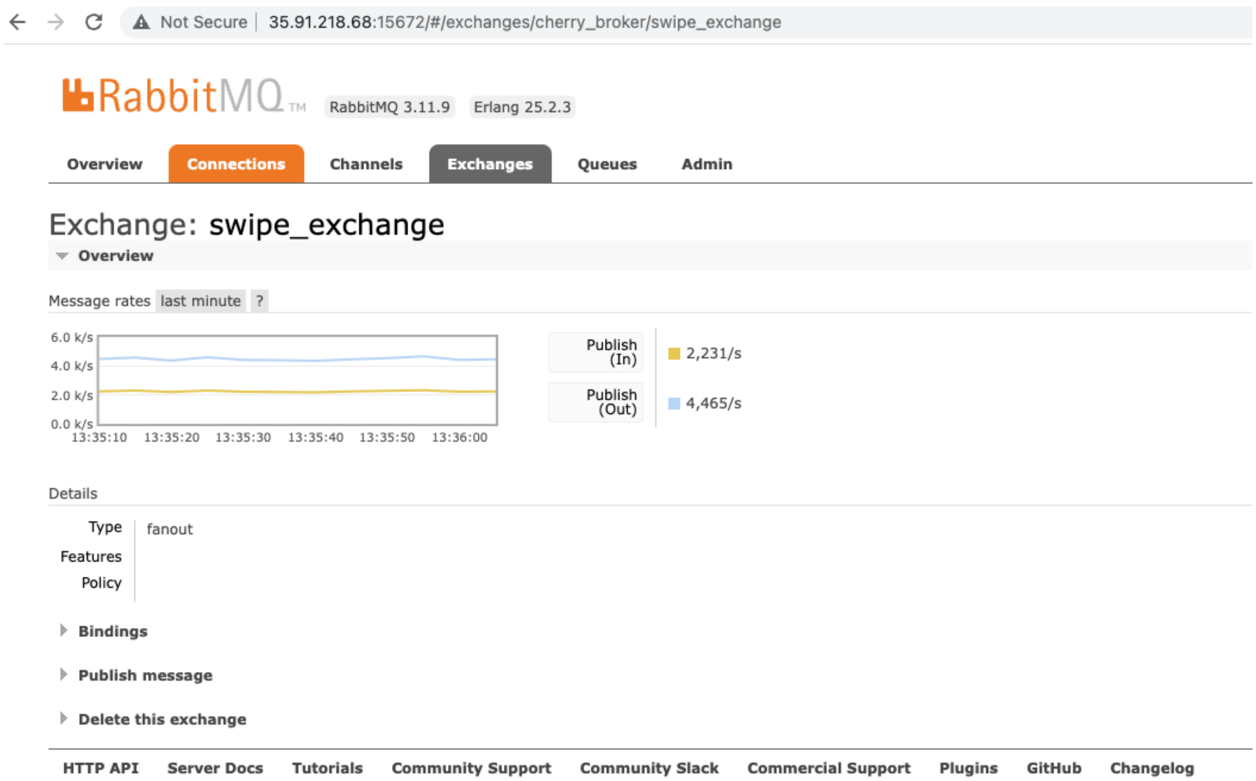
Queue 1: about 2271/s (deliver) , 2263/s(incoming)

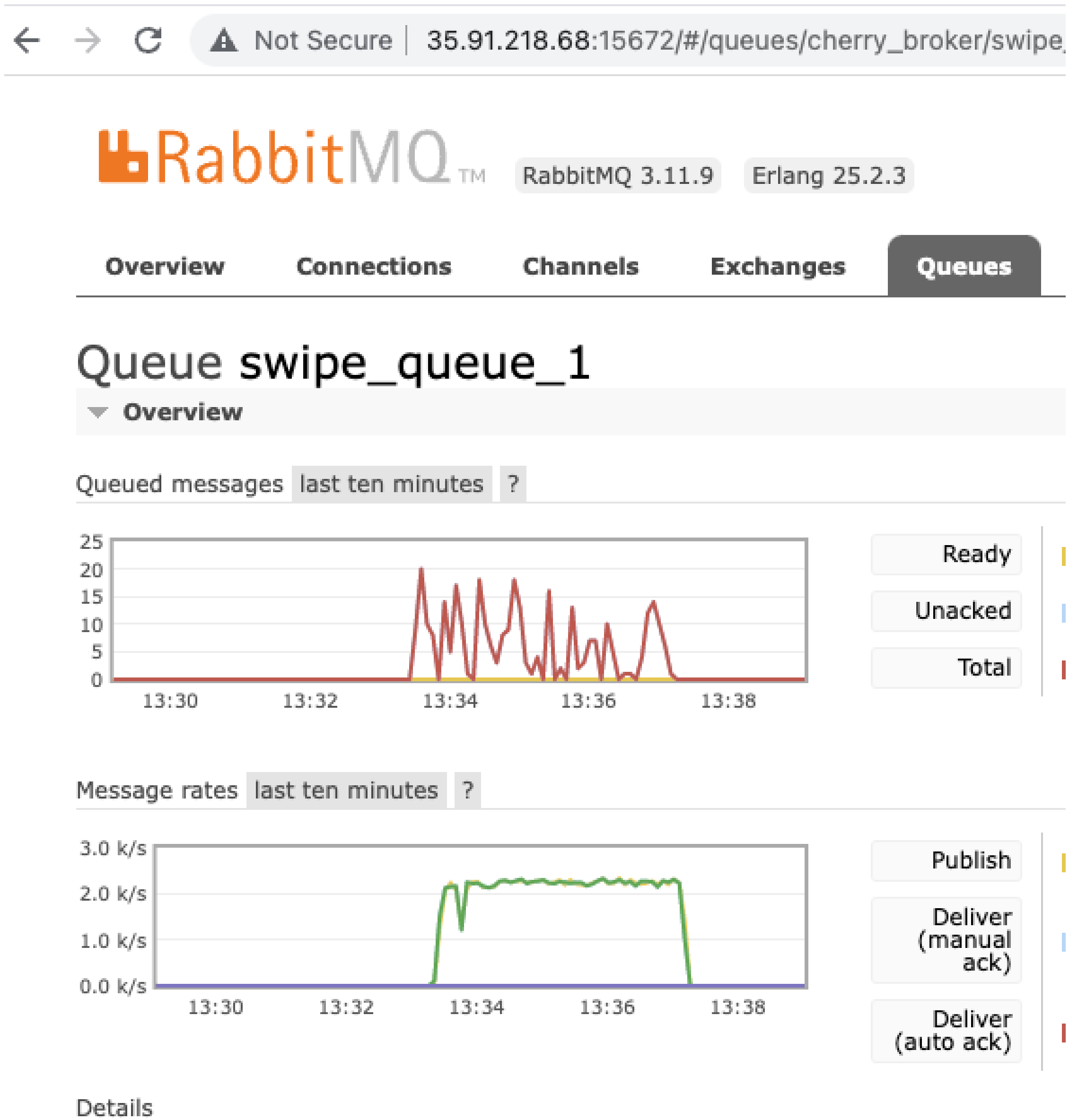
Queue2: about 2271/s (deliver) , 2263/s(incoming)

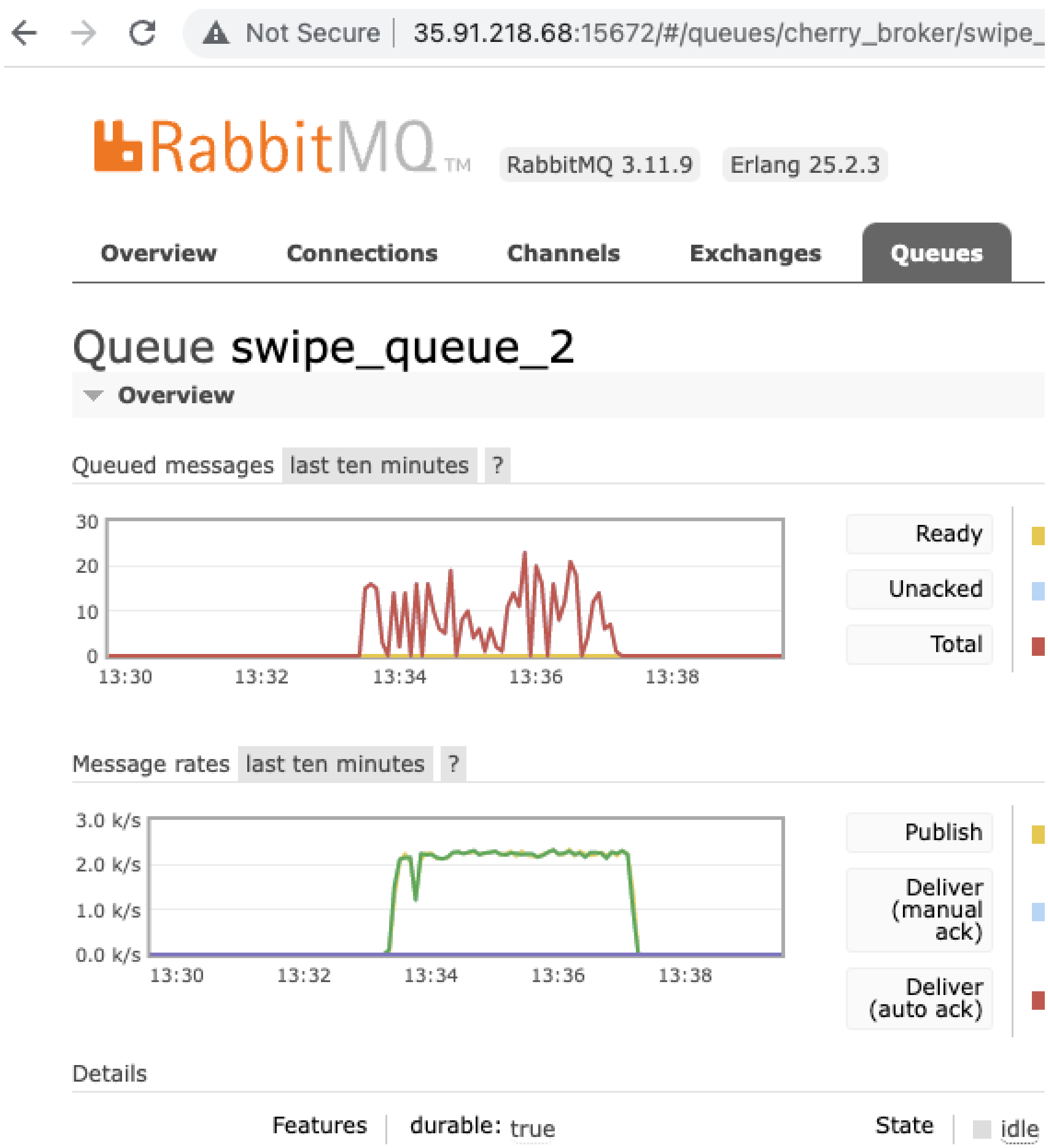




Exchange rate:

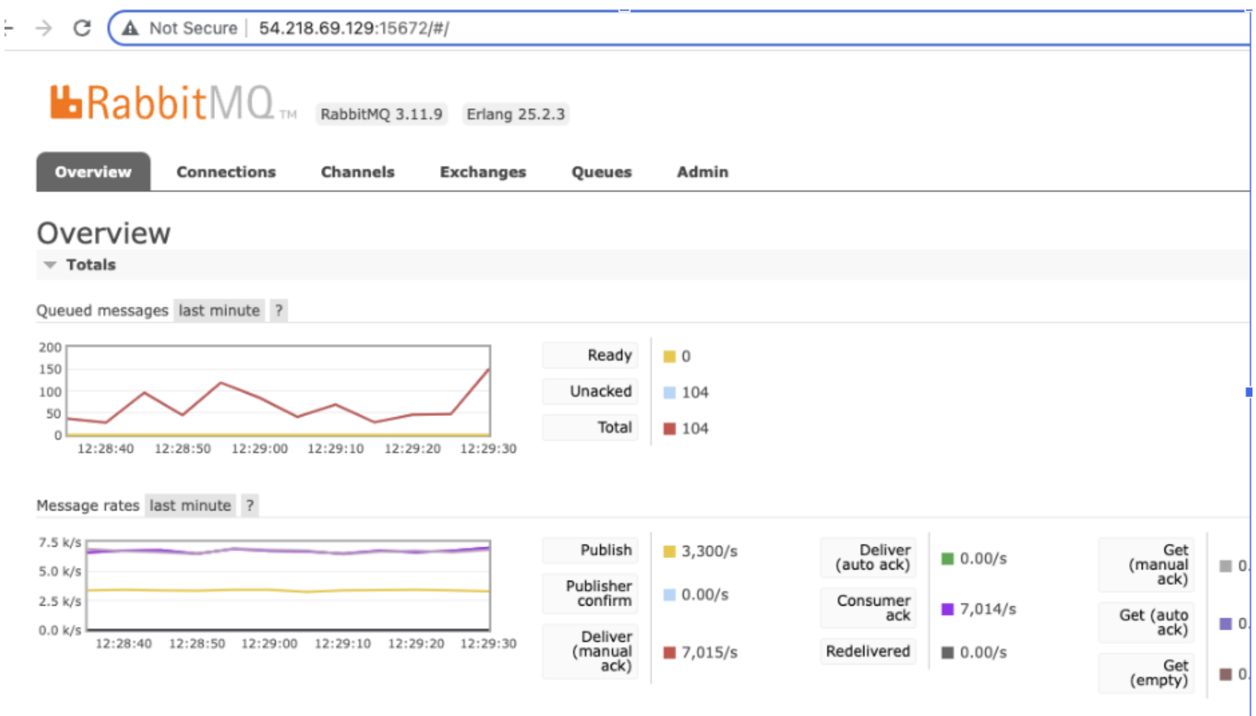


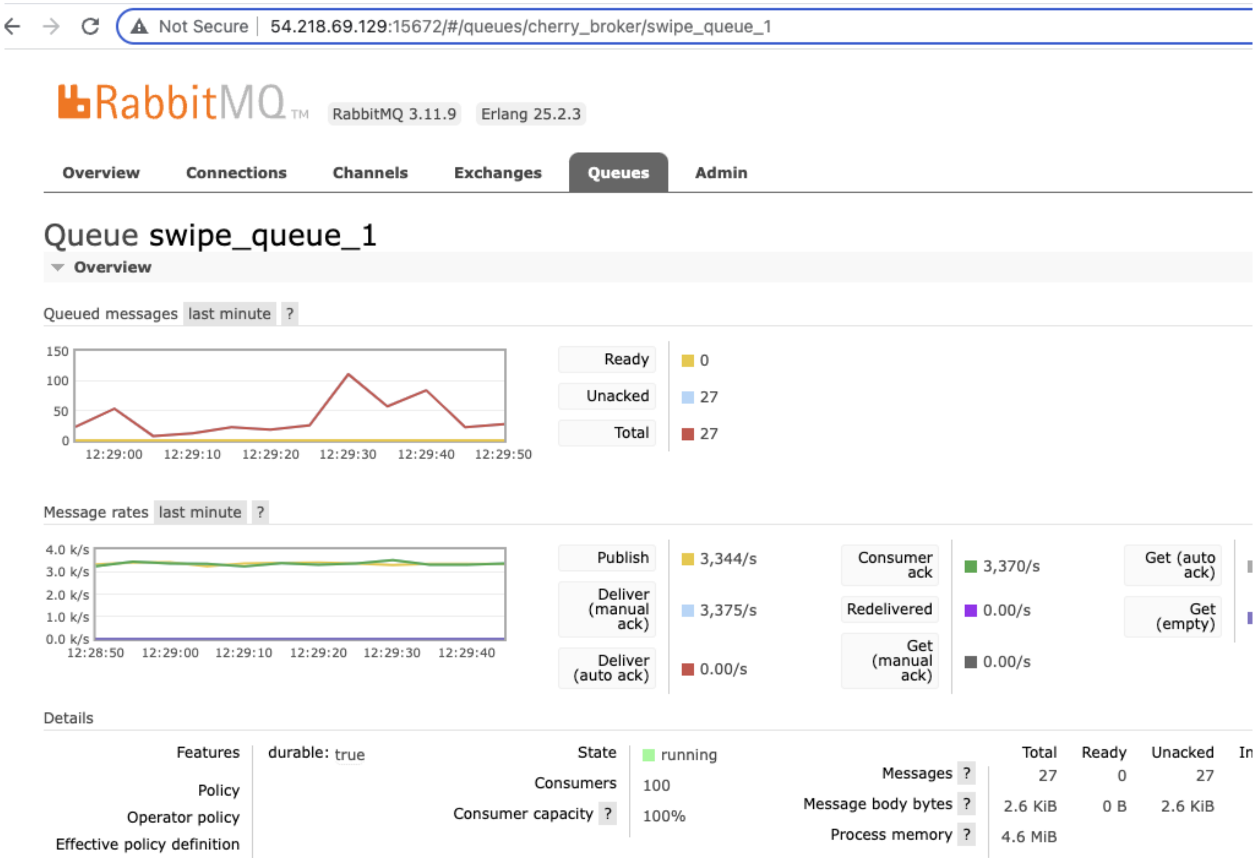

Not Secure
35.91.218.68:15672/#/queues/cherry_broker/swipe_
RabbitMQ TM
RabbitMQ 3.11.9
Erlang 25.2.3
Overview
Connections
Channels
Exchanges
Queues
Queue swipe_queue_1
Overview
Queued messages last ten minutes ?
25
20
15
10
5
0
13:30
13:32
13:34
13:36
13:38
Ready
Unacked
Total
Message rates last ten minutes ?
3.0 k/s
2.0 k/s
1.0 k/s
0.0 k/s
13:30
13:32
13:34
13:36
13:38
Publish
Deliver (manual ack)
Deliver (auto ack)
Details



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.

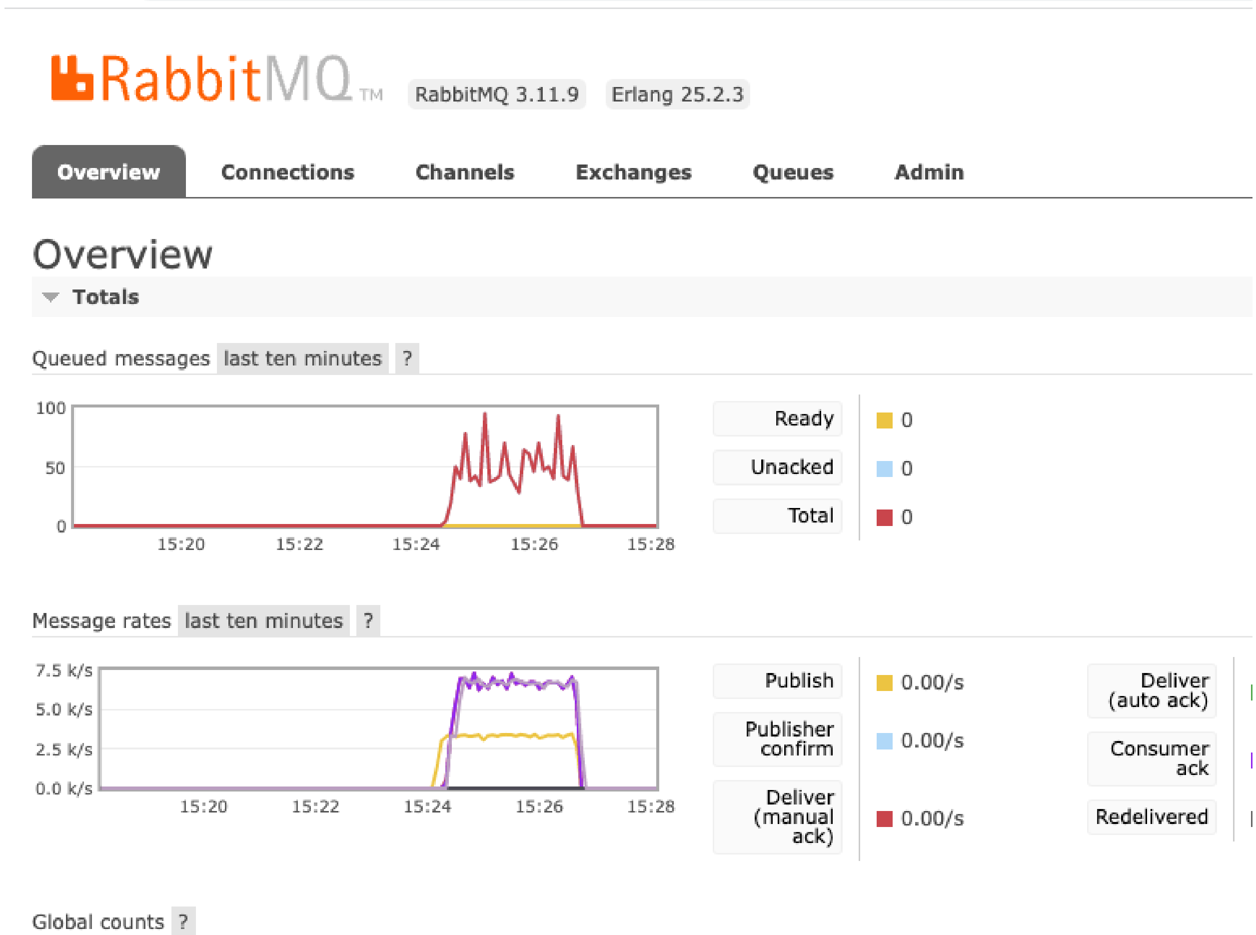

src
main
java
org.example
public class MultiThreadClient {
private static final int numOfThreads = 110;
MultiThreadClient
/Library/Java/JavaVirtualMachines/jdk-11.0.11.jdk/Contents/Home/bin/java ...
Number of successful requests: 500000
Number of fail requests: 0
walltime: 158 seconds
Total throughput: 3164 req/s
Process finished with exit code 0


Not Secure | 54.218.69.129:15672/#/
RabbitMQ
RabbitMQ 3.11.9
Erlang 25.2.3
Overview
Connections
Channels
Exchanges
Queues
Admin
Overview
Totals
Queued messages last minute ?
Ready 0
Unacked 104
Total 104
Message rates last minute ?
Publish 3,300/s
Publisher confirm 0.00/s
Deliver (manual ack) 7,015/s
Deliver (auto ack) 0.00/s
Consumer ack 7,014/s
Redelivered 0.00/s
Get (manual ack)
Get (auto ack)
Get (empty)



Not Secure | 54.218.69.129:15672/#/queues/cherry_broker/swipe_queue_1
RabbitMQ TM
RabbitMQ 3.11.9
Erlang 25.2.3
Overview
Connections
Channels
Exchanges
Queues
Admin
Queue swipe_queue_1
Overview
Queued messages last minute ?
150
100
50
0
12:29:00 12:29:10 12:29:20 12:29:30 12:29:40 12:29:50
Ready 0
Unacked 27
Total 27
Message rates last minute ?
4.0 k/s
3.0 k/s
2.0 k/s
1.0 k/s
0.0 k/s
12:28:50 12:29:00 12:29:10 12:29:20 12:29:30 12:29:40
Publish 3,344/s
Deliver (manual ack) 3,375/s
Deliver (auto ack) 0.00/s
Consumer ack 3,370/s
Redelivered 0.00/s
Get (manual ack) 0.00/s
Get (auto ack)
Get (empty)
Details
Features durable: true
Policy
Operator policy
Effective policy definition
State running
Consumers 100
Consumer capacity ? 100%
Total Ready Unacked In
Messages ? 27 0 27
Message body bytes ? 2.6 KiB 0 B 2.6 KiB
Process memory ? 4.6 MiB


Not Secure
54.218.69.129:15672/#/queues/cherry_broker/swipe_queue_2
RabbitMQ TM
RabbitMQ 3.11.9
Erlang 25.2.3
Overview
Connections
Channels
Exchanges
Queues
Admin
Queue swipe_queue_2
Overview
Queued messages last minute ?
75
50
25
0
12:29:20 12:29:30 12:29:40 12:29:50 12:30:00 12:30:10
Ready 0
Unacked 4
Total 4
Message rates last minute ?
4.0 k/s
3.0 k/s
2.0 k/s
1.0 k/s
0.0 k/s
12:29:20 12:29:30 12:29:40 12:29:50 12:30:00 12:30:10
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
Policy
Operator policy
State running
Consumers 100
Consumer capacity ? 100%
Total
Messages ? 4
Message body bytes ? 410 B


RabbitMQ™
RabbitMQ 3.11.9
Erlang 25.2.3
Overview
Connections
Channels
Exchanges
Queues
Admin
Overview
Totals
Queued messages last ten minutes ?
100
50
0
15:20
15:22
15:24
15:26
15:28
Ready 0
Unacked 0
Total 0
Message rates last ten minutes ?
7.5 k/s
5.0 k/s
2.5 k/s
0.0 k/s
15:20
15:22
15:24
15:26
15:28
Publish 0.00/s
Publisher confirm 0.00/s
Deliver (manual ack) 0.00/s
Deliver (auto ack)
Consumer ack
Redelivered
Global counts ?

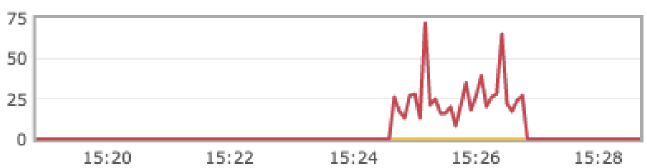
RabbitMQ™ RabbitMQ 3.11.9 Erlang 25.2.3

Overview | Connections | Channels | Exchanges | **Queues** | Admin

Queue swipe_queue_1

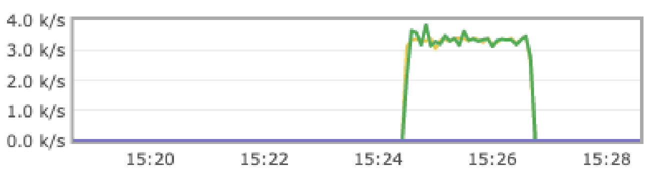
▼ Overview

Queued messages last ten minutes ?



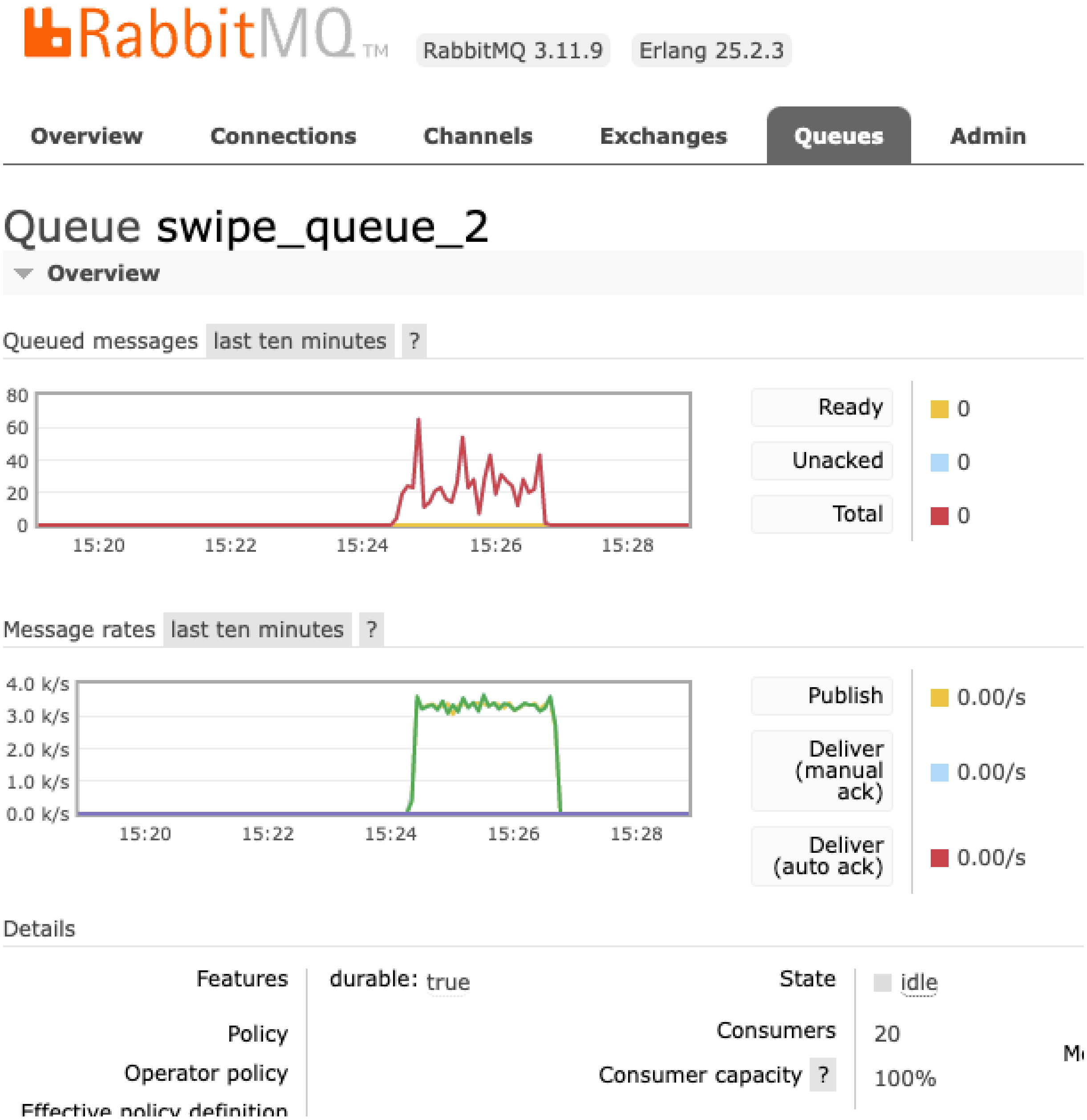
Ready	0
Unacked	0
Total	0

Message rates last ten minutes ?



Publish	0.00/s	Consumer ack
Deliver (manual ack)	0.00/s	Redelivered
Deliver (auto ack)	0.00/s	Get (manual ack)

Details



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.

It is essential to recognize that including an Application Load Balancer did not double the throughput, since various factors can affect performance aside from just adding a server. Moreover, even after sending 500,000 requests in single servlet mode, the CPU usage remained consistently low, around 20-30%. Notably, the CPU usage for each servlet instance also remained similar when operating under the ALB mode.

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.