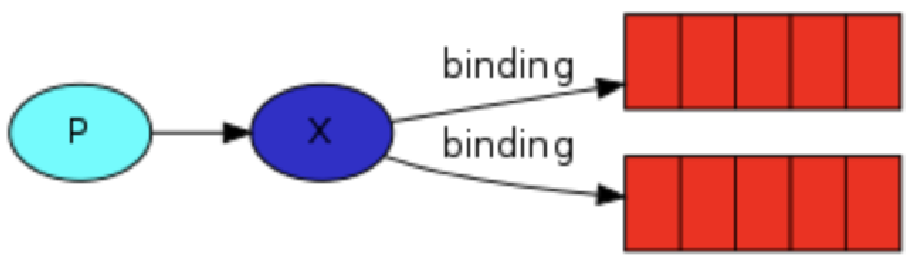
HW 3 Report - Linni Cai

Github URL:

https://github.com/linni-cai-lc/CS6650_Distributed_System/tree/main/hw3

Design:



In this assignment, since we have two consumers, one for skiers, one for resorts, but they share the same data from the server, so I chose RMQ publish/subscribe pattern, the server publishes, two consumers subscribe. The whole workflow is that client serves as producer, it sends plenty of posts to server, the server delivers results to consumers.

Database design is based on Redis key/value structure, each consumer stores the data in its instance's Redis storage.

- Skier Consumer: I created skierId + liftTime for skier consumer's uuid, since we know that a skier can only ski one time at the same liftTime, so this key combo will be unique, and doesn't overlap with other results.
- Resort Consumer: I created dayId + skierId + liftTime for resort consumer's uuid, similarly, the latter two can be unique, since we might need index later, dayId will be necessary to search as index keyword, so I add it into the key combo.
- The value for both is the same, it is a LiftRide JSON format string object, it can be easily converted back to the object for later usage.

Process:

- created 4 EC2 instance
    - 1 Linux instance running the server
        - provides with the skier API functionality
        - connect to load balancer
        - send messages to the queue
    - 2 Linux instance running the consumer

        - 1 for skier consumer
        - 1 for resort consumer
        - receive messages from the queue
    - 1 Ubuntu instance running the RabbitMQ server
        - owns the queue and store messages

Name	Instance ID
Linux (Server)	i-0e965884ba592ab77
Ubuntu (RabbitMQ)	i-066a6081de2958826
Linux (Consumer_Resort)	i-0c1485ce96d70d50a
Linux (Consumer_Skier)	i-0285065fef21ceb1c

Results:

The experiments are based on 20000 skiers, 40 lifts. Overall the queue size is pretty small, since consumption/production rate is nearly 1, so no backlog exists when I applied mitigation strategy.

Mitigation strategy is to add circuit breaker, I added it in Client side. I started my experiment with a smaller number of skiers such as 10 and 100, it worked fine without a circuit breaker, however when I increased it to 1000, there were a lot of backlogs, and API calls failed a lot, which hit the upper bound 5 failures for each POST in phase 1, and no more successful POST. Then I added the circuit breaker inside Client POST generation, it will hold to POST when there are specific number of POST sending to server already, and restart to send POST when previous POST finish, finally send all required number of POST. The effect is obvious, since I can run experiments with 20000 skiers, and the whole process is pretty quick. Compared to 256 threads, 128 threads experiment is faster, with smaller response time as well.

Step 1 Skiers

- Command window, use **java -jar consumer_skier.jar**

```
[*] Waiting for messages. To exit press CTRL+C
[*] Waiting for messages. To exit press CTRL+C
[*] Waiting for messages. To exit press CTRL+C
[*] Waiting for messages. To exit press CTRL+C
[*] Waiting for messages. To exit press CTRL+C
[*] Waiting for messages. To exit press CTRL+C
[*] Waiting for messages. To exit press CTRL+C
[*] Waiting for messages. To exit press CTRL+C
[*] Waiting for messages. To exit press CTRL+C
[*] Waiting for messages. To exit press CTRL+C
[*] Waiting for messages. To exit press CTRL+C
[*] Waiting for messages. To exit press CTRL+C
[*] Waiting for messages. To exit press CTRL+C
[*] Waiting for messages. To exit press CTRL+C
[*] Waiting for messages. To exit press CTRL+C
[*] Waiting for messages. To exit press CTRL+C
```

i-0285065fef21ceb1c (Linux (Consumer_Skier))

Public IPs: 34.222.68.109 Private IPs: 172.31.7.147

- RMQ management window for queue size

Queues

All queues (1)

Pagination

Page 1 of 1 - Filter: Regex ?

Overview				Messages			Message rates		
Name	**Type**	Features	**State**	**Ready**	**Unacked**	**Total**	**incoming**	**deliver / get**	**ack**
server_queue	classic	D	running	0	0	0	957/s	953/s	0.00/s

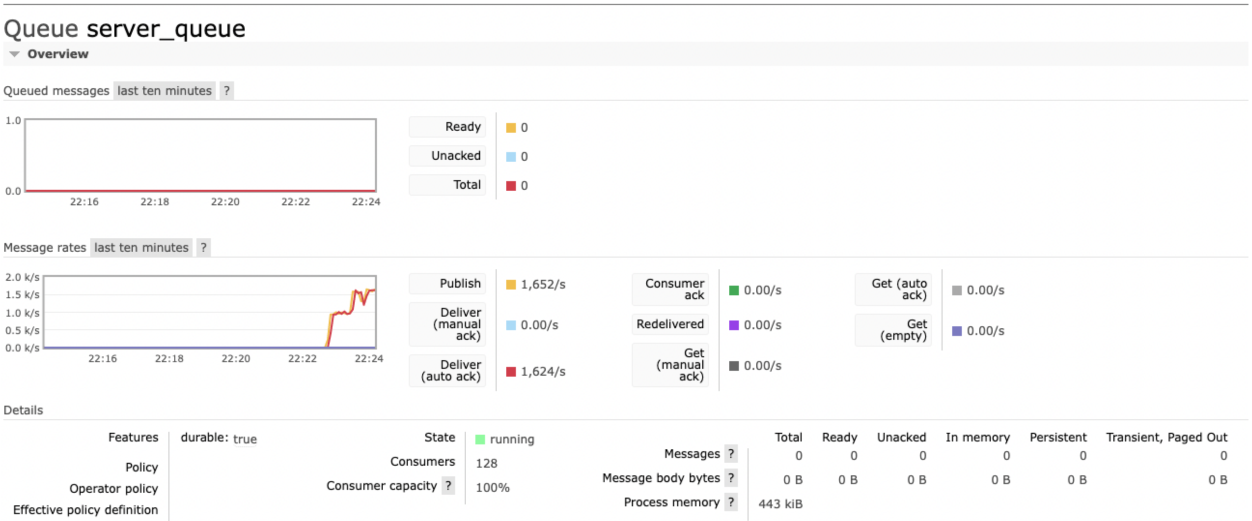
+/-

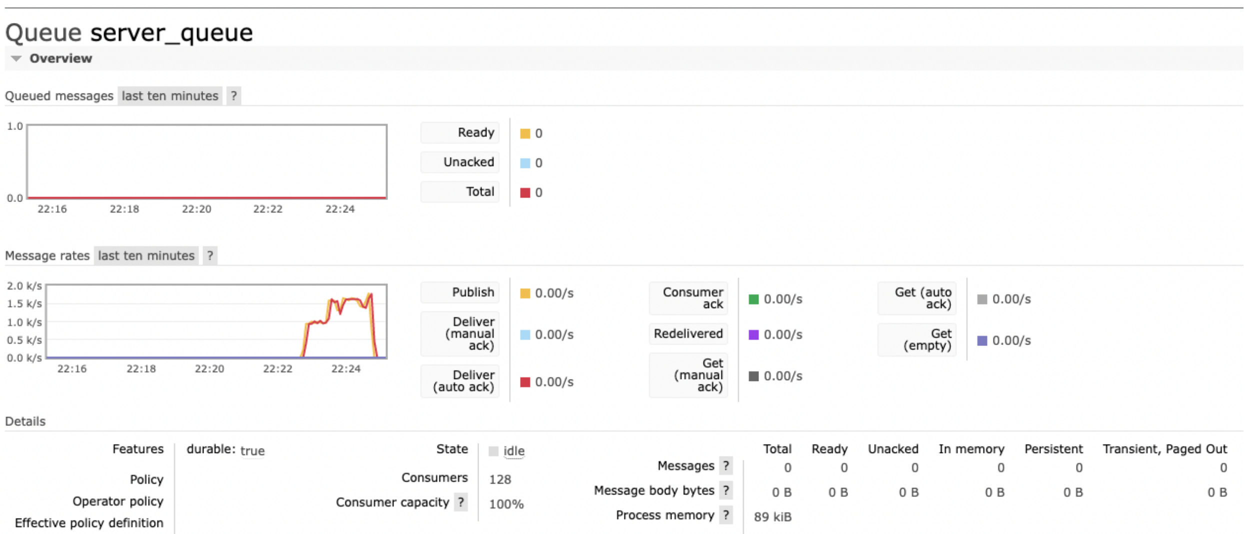
- 128 client threads

```
------ PART 1 ------
number of successful requests sent: 159977
number of unsuccessful requests: 0
the total run time for all phases to complete: 124753
the total throughput in requests per second: 1000
```

```
------  PART 2  ------
mean response time (millisecs): 71
median response time (millisecs): 59
throughput: 1000
p99 (99th percentile) response time: 271
min response time (millisecs): 11
max response time (millisecs): 987
```

- The queue size range is 0 - 1, message rate is send/receive = 1652 / 1624 = 1.02



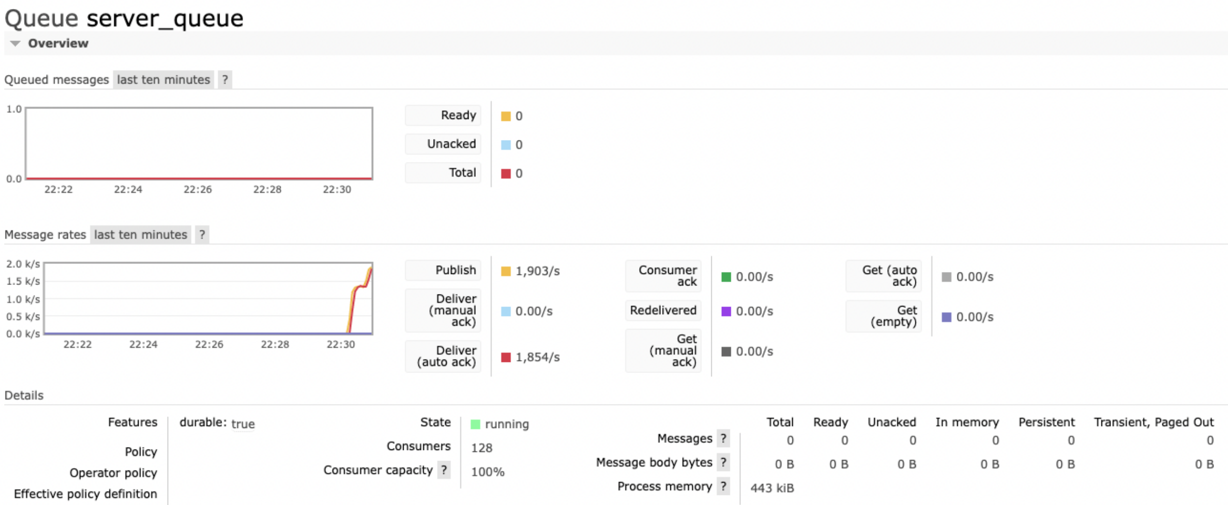


- 256 client threads

```
------  PART 1  ------
number of successful requests sent: 160420
number of unsuccessful requests: 0
```

```
the total run time for all phases to complete: 105108
the total throughput in requests per second: 1000

------  PART 2  ------
mean response time (millisecs): 119
median response time (millisecs): 83
throughput: 1000
p99 (99th percentile) response time: 542
min response time (millisecs): 12
max response time (millisecs): 1443
```

- The queue size range is 0 - 1, message rate is send/receive = 1903 / 1854 = 1.03



Step 2 Resorts

- Command window, use **java -jar consumer_resort.jar**

```
[*] Waiting for messages. To exit press CTRL+C
[*] Waiting for messages. To exit press CTRL+C
[*] Waiting for messages. To exit press CTRL+C
[*] Waiting for messages. To exit press CTRL+C
[*] Waiting for messages. To exit press CTRL+C
[*] Waiting for messages. To exit press CTRL+C
[*] Waiting for messages. To exit press CTRL+C
[*] Waiting for messages. To exit press CTRL+C
[*] Waiting for messages. To exit press CTRL+C
[*] Waiting for messages. To exit press CTRL+C
[*] Waiting for messages. To exit press CTRL+C
[*] Waiting for messages. To exit press CTRL+C
[*] Waiting for messages. To exit press CTRL+C
[*] Waiting for messages. To exit press CTRL+C
```

i-0c1485ce96d70d50a (Linux (Consumer_Resort))

Public IPs: 54.201.233.174   Private IPs: 172.31.28.191

- RMQ management window for queue size

Queues

All queues (1)

Pagination

Page 1 of 1 - Filter: Regex ?

Overview				Messages			Message rates		
Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
server_queue	classic	D	running	0	0	0	991/s	950/s	0.00/s

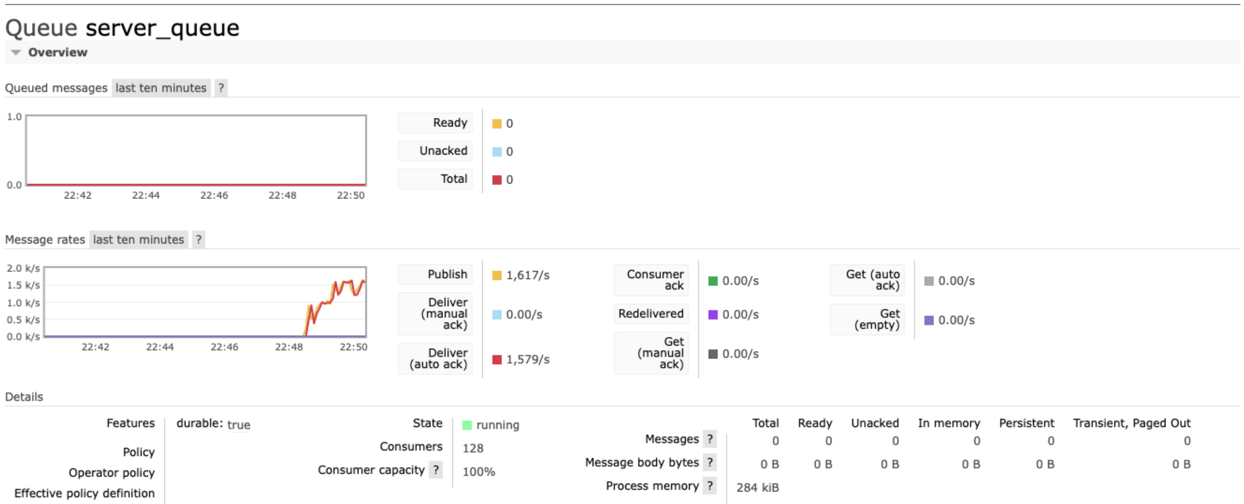
+/-

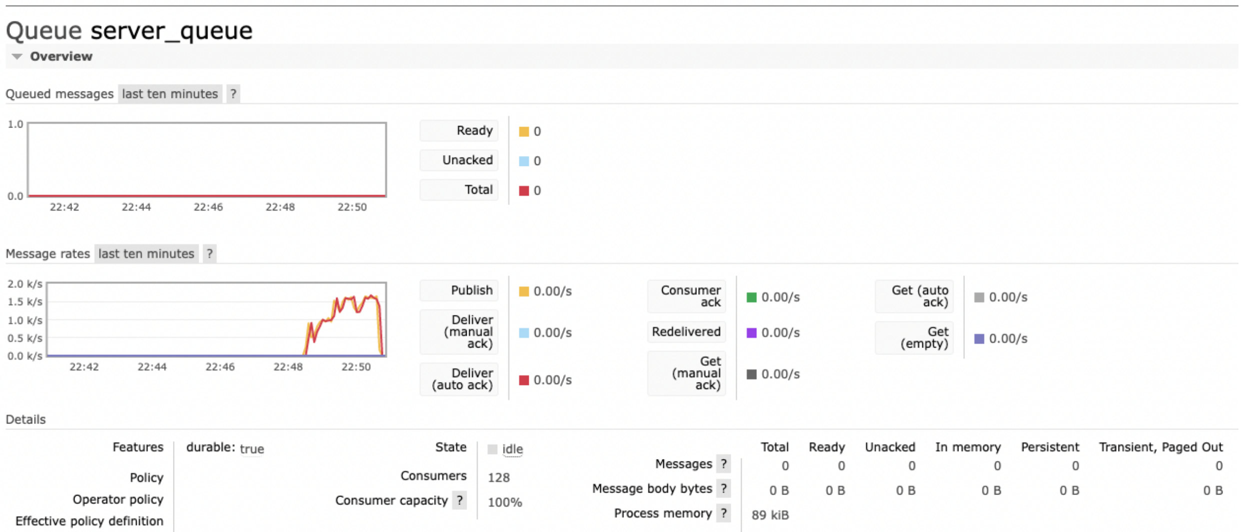
- 128 client threads

```
------  PART 1  ------
number of successful requests sent: 159977
number of unsuccessful requests: 0
the total run time for all phases to complete: 129550
the total throughput in requests per second: 1000
```

```
------  PART 2  ------
mean response time (millisecs): 73
median response time (millisecs): 57
throughput: 1000
p99 (99th percentile) response time: 319
min response time (millisecs): 12
max response time (millisecs): 1058
```

- The queue size range is 0 - 1, message rate is send/receive = 1617 / 1579 = 1.02



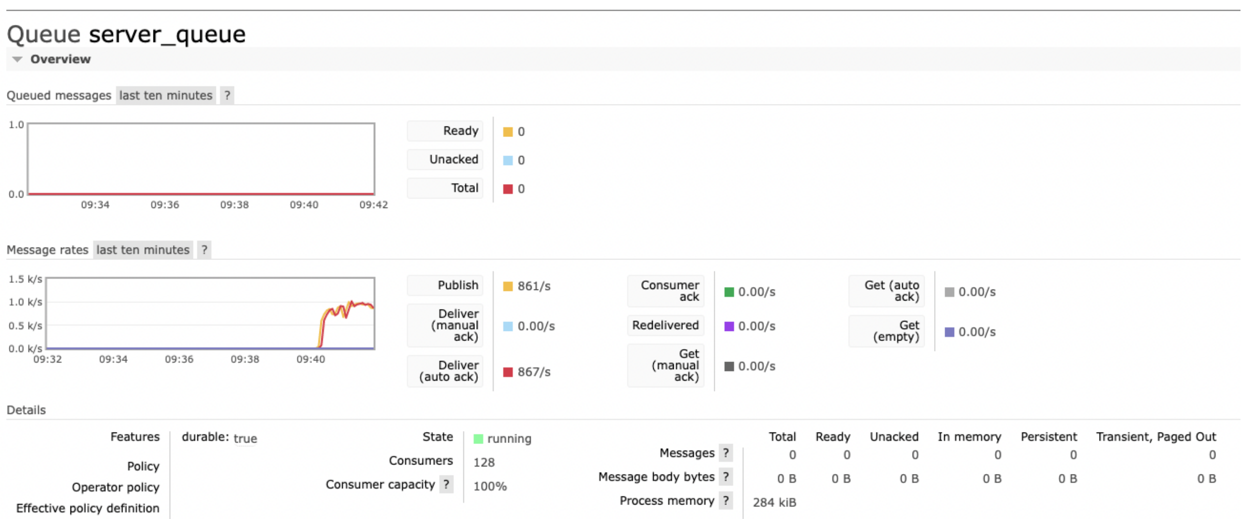


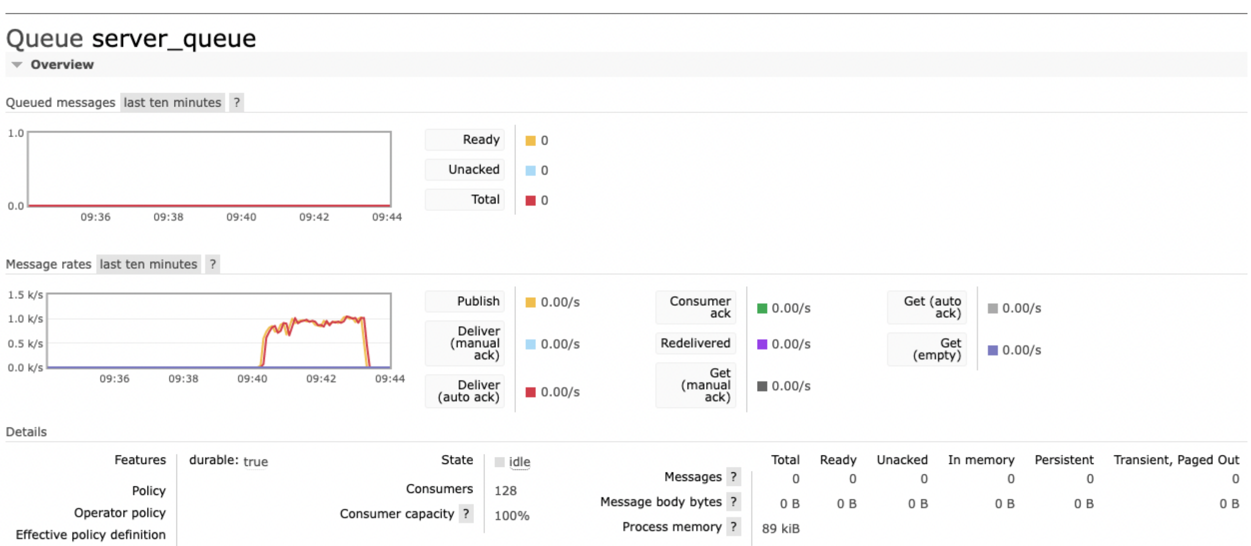
- 256 client threads

```
------  PART 1  ------
number of successful requests sent: 160420
```

```
number of unsuccessful requests: 0
the total run time for all phases to complete: 181003
the total throughput in requests per second: 0

------  PART 2  ------
mean response time (millisecs): 218
median response time (millisecs): 184
throughput: 0
p99 (99th percentile) response time: 884
min response time (millisecs): 1
max response time (millisecs): 1761
```

- The queue size range is 0 - 1, message rate is send/receive = 861 / 867 = 0.99





Step 3 Both

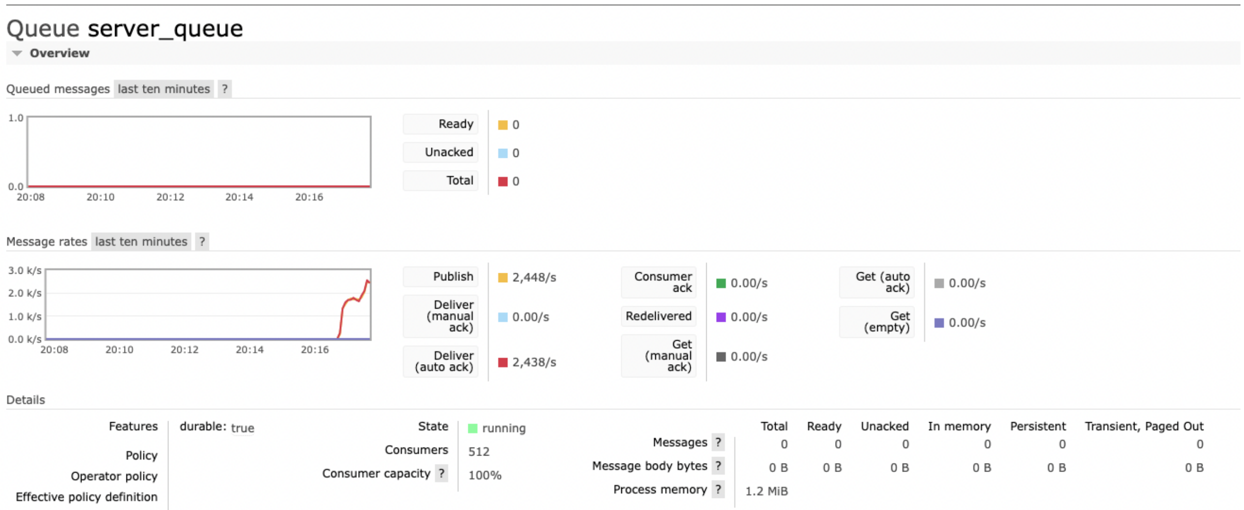
I run experiments for different consumer max threads 128/256, but the outputs show the thread doesn't affect too much, since the runtimes are similar.

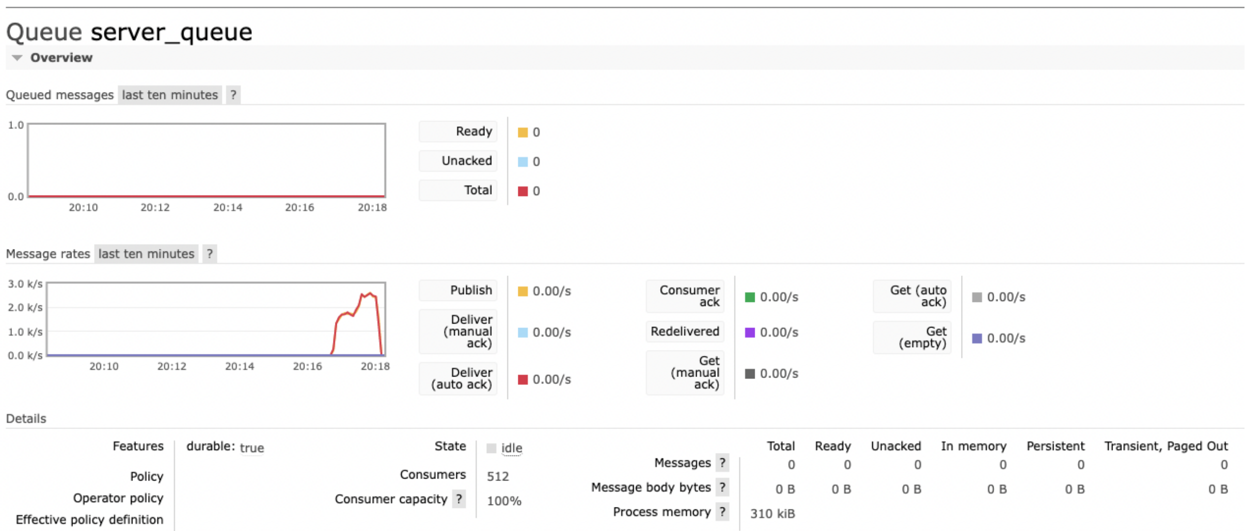
In two EC2 instances, one for skier jar, one for resort jar, start together


```
consumer max thread = 256
client thread = 256
------  PART 1  ------
number of successful requests sent: 160420
number of unsuccessful requests: 0
the total run time for all phases to complete: 103891
the total throughput in requests per second: 1000

------  PART 2  ------
mean response time (millisecs): 118
median response time (millisecs): 78
throughput: 1000
p99 (99th percentile) response time: 590
min response time (millisecs): 12
max response time (millisecs): 1575
```

The following are RMQ screenshots:

- The queue size range is 0 - 1, message rate is send/receive = 2448 / 2438 = 1.00

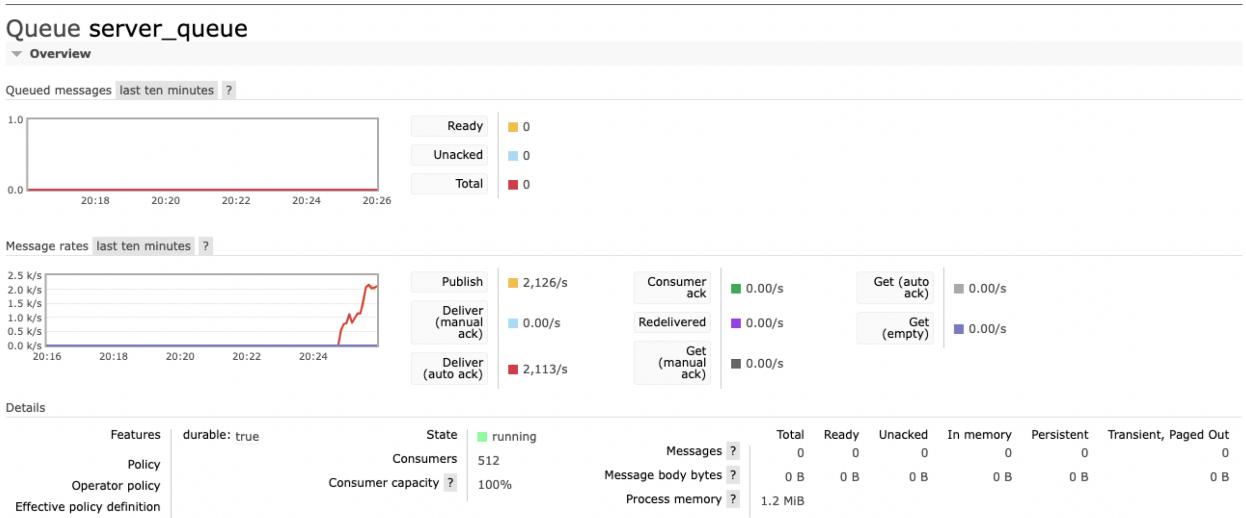


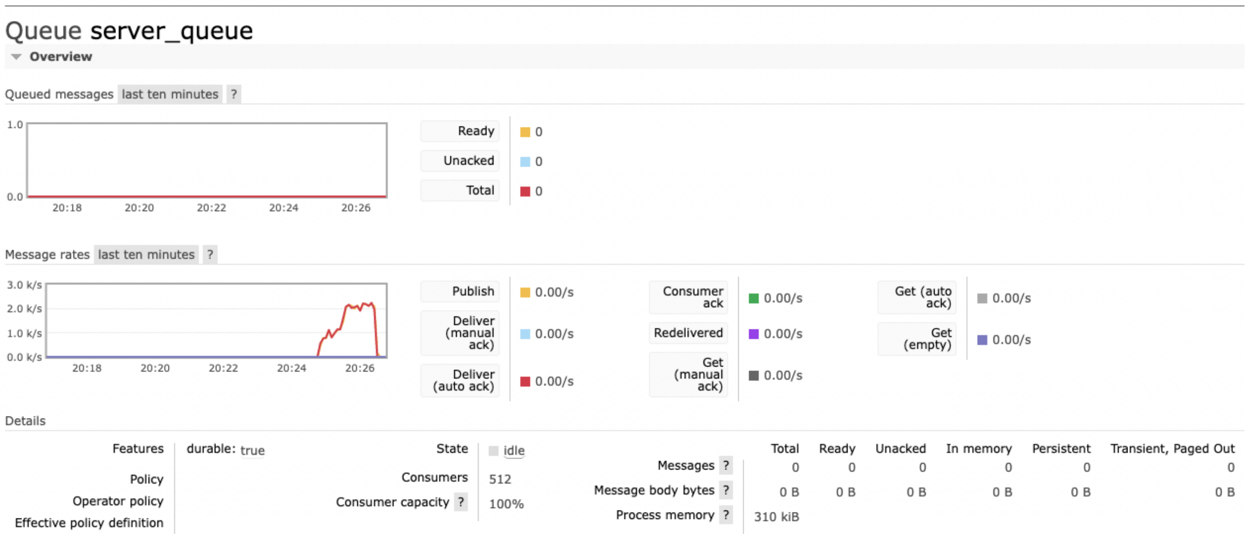



```
consumer max thread = 256
client thread = 128
------  PART 1  ------
number of successful requests sent: 160420
number of unsuccessful requests: 0
the total run time for all phases to complete: 102436
the total throughput in requests per second: 1000

------  PART 2  ------
mean response time (millisecs): 116
median response time (millisecs): 79
throughput: 1000
p99 (99th percentile) response time: 534
min response time (millisecs): 11
max response time (millisecs): 1555
```

The following are RMQ screenshots:

- The queue size range is 0 - 1, message rate is send/receive = 2126 / 2113 = 1.01



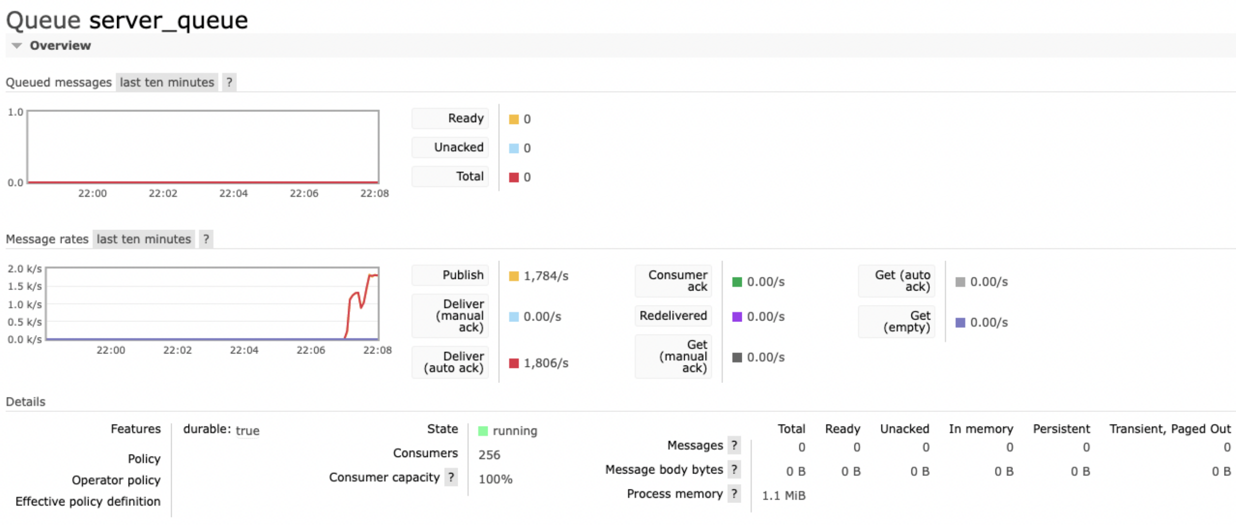


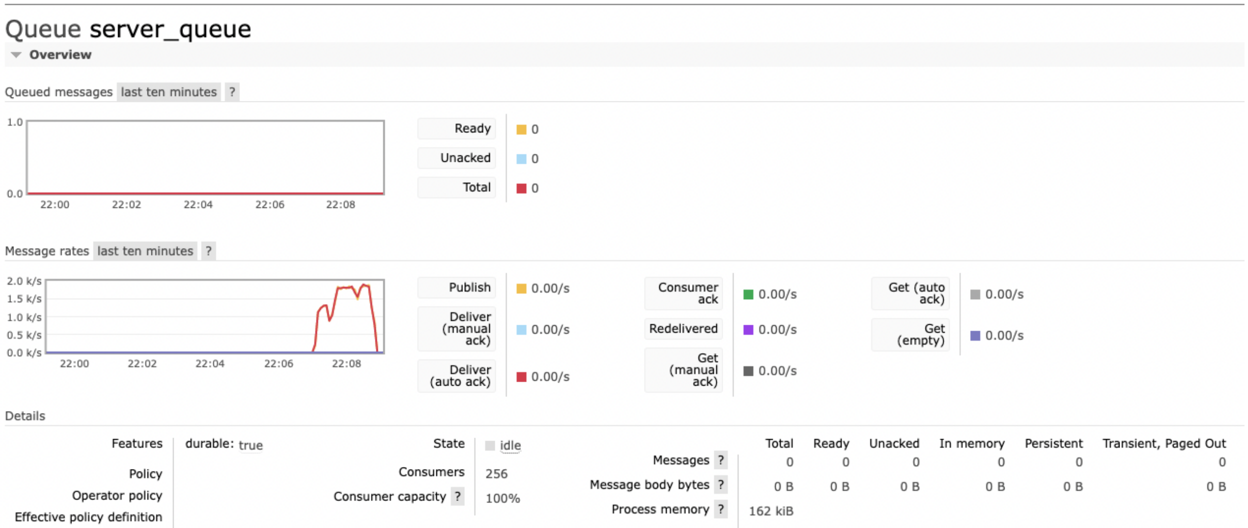

```
consumer max thread = 128
client thread = 256
------  PART 1  ------
number of successful requests sent: 159977
number of unsuccessful requests: 0
the total run time for all phases to complete: 100840
the total throughput in requests per second: 1000

------  PART 2  ------
mean response time (millisecs): 53
median response time (millisecs): 42
throughput: 1000
```

```
p99 (99th percentile) response time: 227
min response time (millisecs): 11
max response time (millisecs): 801
```

The following are RMQ screenshots:

- The queue size range is 0 - 1, message rate is send/receive = 1784 / 1806 = 0.99






```
consumer max thread = 128
client thread = 128
------  PART 1  ------
number of successful requests sent: 159977
number of unsuccessful requests: 0
the total run time for all phases to complete: 123651
the total throughput in requests per second: 1000
```

```
------  PART 2  ------
mean response time (millisecs): 68
median response time (millisecs): 55
throughput: 1000
p99 (99th percentile) response time: 261
min response time (millisecs): 12
max response time (millisecs): 1066
```

The following are RMQ screenshots:

- The queue size range is 0 - 1, message rate is send/receive = 1528 / 1503 = 1.02