6650 Assignment 3

URL: https://github.com/tjuqxb/6650-assignment-3

Database Design

Redis is a Key-Value database and does not support complex join queries. This design uses key names as descriptions of query conditions and values as query results.
It uses Redis Sets to contain non-duplicate records (e.g., the days one user skied in one season) and Redis Strings to record numeric values (e.g., vertical totals for each ski day for a specific user). The following is the example of the query of the days one user skied in one season:

Key: The first two digits are the serviceId, the middle parts are query constraints similar to SQL 'where clause', and the last word describes additional data attributes of the value.
In the first example case, the key format is "serviceId + 'u' + userId + 's' + seasonId + 'days'". For example, the key can be "01u45s2020days".
Value: In the first example case. Its value is a Set of daysId. So, we can add or remove days from the Set. We can also query to get all values of the Set. The values are all distinct days one specific user skied in one specified season.

In the second example case, the key format is similar ("serviceId + 'u' + userId + 's' + seasonId + 'd' + daysId + 'vertical'). But its value is simply a string representation of a number. This enables us to update it use Redis command easily (such as INCR, DECR, INCRBY, etc.)

Skier Microservice:
“For skier N, how many days have they skied this season?”
Key: serviceId + "u" + userId + "s" + seasonId + "days"
Value: a Set of daysId String
Operations: SADD, SREM, SMEMBERS, SCARD
Update: Insert element to the Set
Query: Use SCARD to query the number of elements in the Set.

“For skier N, what are the vertical totals for each ski day?” (calculate vertical as liftID*10)
Key: serviceId + "u" + userId + "s" + seasonId + "d" + daysId + "vertical"
Value: a String representing a number
Operations: SET, GET, INCR, DECR, INCRBY
Update: Add the calculated number to the value.
Query: Use GET to get the value.

“For skier N, show me the lifts they rode on each ski day”
Key: serviceId + "u" + userId + "s" + seasonId + "d" + daysId + "lifts"
Value: a Set of liftId String
Operations:SADD, SREM, SMEMBERS, SCARD

Update: Insert element to the Set
Query: Use SMEMBERS to get all the elements in the Set

Resort Microservice:
“How many unique skiers visited resort X on day N?”
Key: serviceId + "re" + resortId + "s" + seasonId + "d" + daysId + "users"
Value: a Set of userId String
Operations:SADD, SREM, SMEMBERS, SCARD
Update: Insert element to the Set
Query: Use SCARD to query the number of elements in the Set.

“How many rides on lift N happened on day N?”
Key: serviceId + "li" + liftId + "s" + seasonId + "d" + daysId + "freq"
Value: a String representing a number
Operations: SET, GET, INCR, DECR
Update: Add one to the corresponding value for each lift post.
Query: Use GET to get the value.

“On day N, show me how many lift rides took place in each hour of the ski day”
Key: serviceId + "s" + seasonId + "d" + daysId + "h" + hour + "freq"
Value: a String representing a number
Operations: SET, GET, INCR, DECR
Update: Calculate hour from time field of the lift post and add one to the corresponding value for each lift post.
Query: Use GET to get the value.

Deployment Topologies

Servers: I use two t2.micro EC2 instances as servers and add one load balancer.
RabbitMQ: I use one t2.small EC2 instance to run RabbitMQ. When a server receives one POST, the server would publish two identical messages to two queues of RabbitMQ. One queue is "skier_service_queue" and the other queue is "resort_service_queue". The message represents all the skier lift information of a POST.
Consumers: There are two multi-thread consumer services and each of them runs on a t2.micro EC2 instance. They are both connected to RabbitMQ and Redis. One conumser is connected to "skier_service_queue" and is responsible for writing Skier Microservice data to Redis. The other consumer is connected to "resort_service_queue" and is responsible for writing Resort Microservice data to Redis.
Redis: I use two t2.micro EC2 instances to run Redis. They store and persist the data needed for two microservices respectively.

Screenshots and Mitigation Comparisons

Screenshots

The screenshots are after this section. The order is "only skier service", "only resort service", "both services" and "both services after mitigation".

Mitigation Strategy and Comparison

Before mitigation, we can see even single service with 128 client threads would cause a long message queue in RabbitMQ and the queue size would keep increasing during high throughput periods. Basically, before migration, the messages queues are long. Double-services would worsen the situation.

The strategy I adopted is using circuit-breaker with exponential backoffs in the server side. Basically, when the server receives more than 2000 POST in 2 seconds, the current thread would be put to sleep for the time calculated with exponential backoffs (For example, it would pick a random value from 0-1, 0-2, 0-4, 0-8, 0-16...0-16384 milliseconds to sleep.). Each time the picking range would grow exponentially until the range meets the predefined maximum value. The circuit-breaker would be recovered when the throughput is less than 1600 POST in 2 seconds.

The aim of this strategy is a temporary constraining of the publishing message speed to RabbitMQ and controlling throughput.

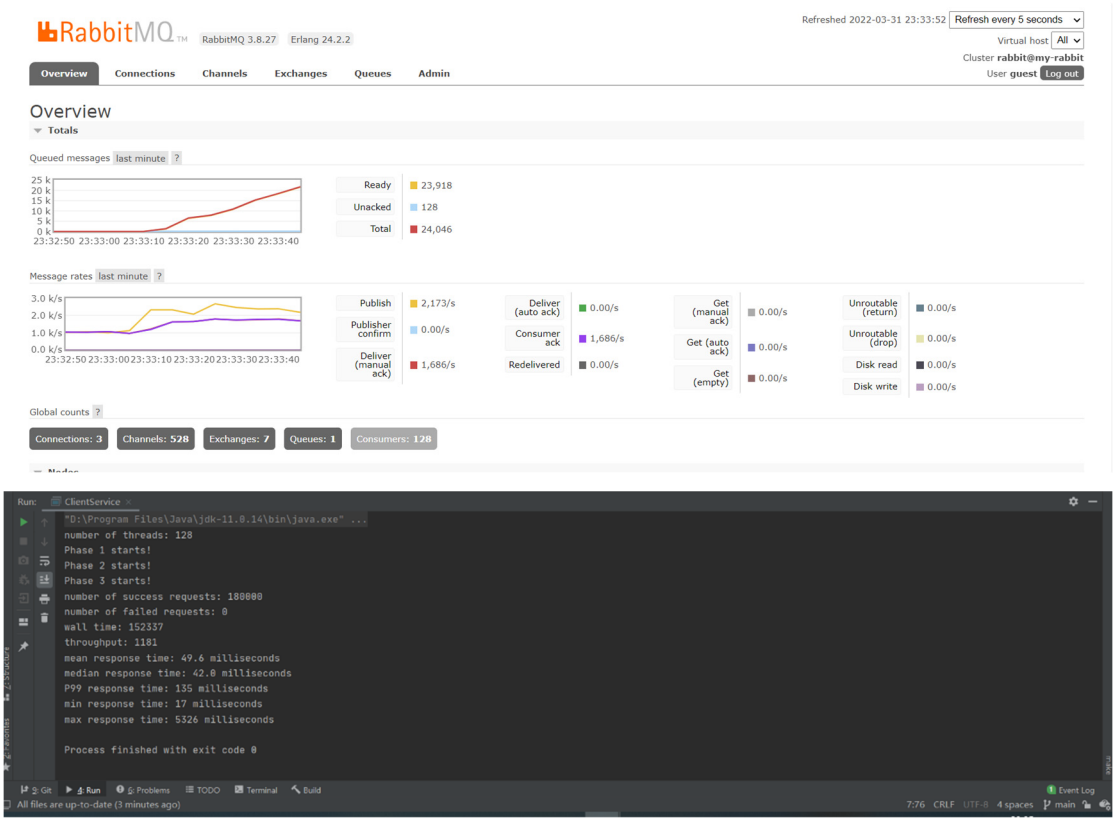
Before Mitigation: For all the conditions (single service or both services), queues are long and would keep increasing size until client threads enter the cool down phase. It can be seen from the RabbitMQ graph that publish rates are larger than deliver rates consistently.

With both services, for 128 threads, the throughput is 933, for 256 threads, the throughput is 1170.

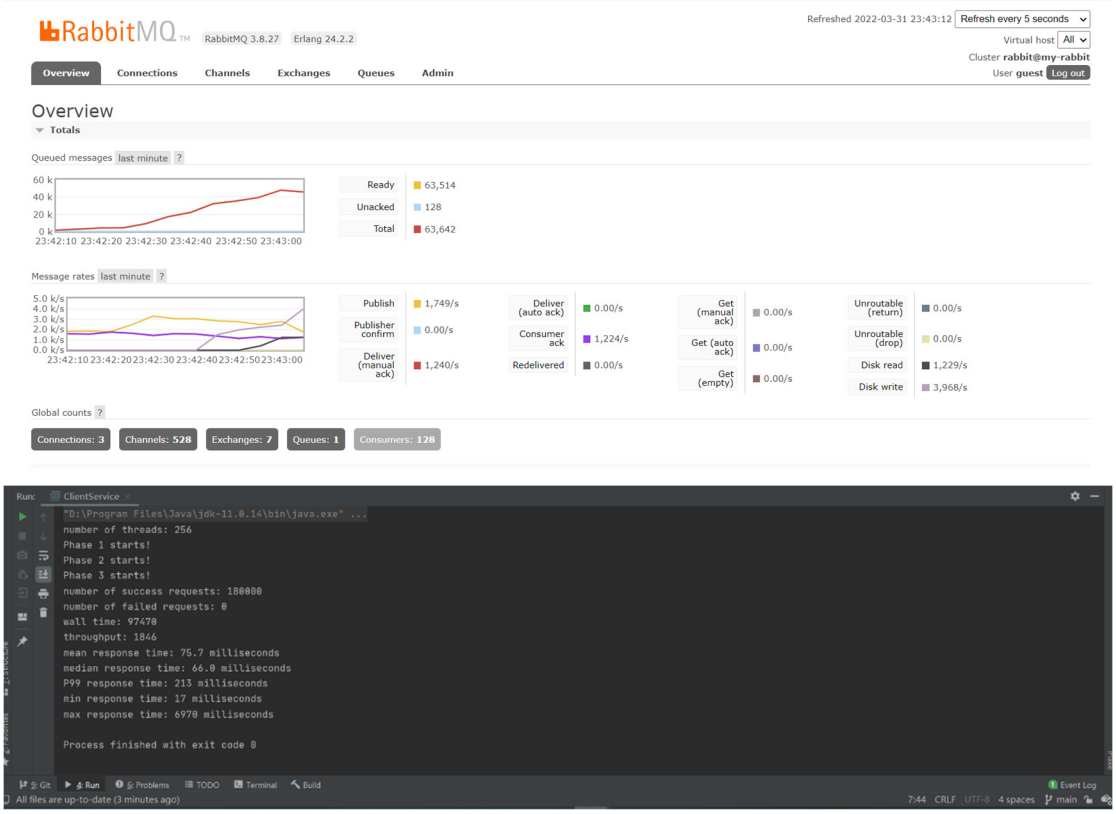
After Mitigation: Queues sizes are controlled. We can see from the graph that when the queue has a tendency of growing, the size would soon become smaller. That is the reason why there are some small spikes on the RabbitMQ graph. With both services, for 128 threads, the throughput is 934, for 256 threads, the throughput is 1155.

Analysis:

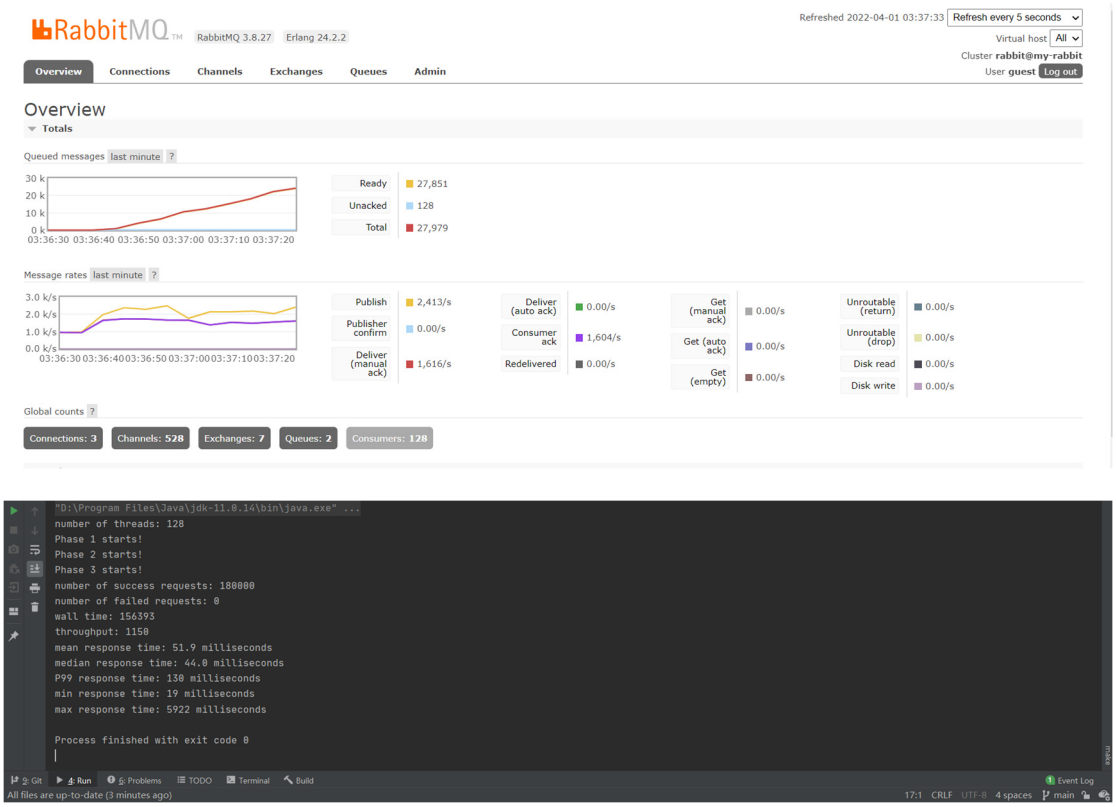
1. The reason of improvement. We cannot control the actual number of threads running in Tomcat. But it can be imagined that during the peak phase, a certain number of threads would be put to sleep and those threads cache the POST data before waking up. Since the total number of POST requests does not change, this mitigation reduces the number of the requests to be handled during the peak phase and this action is immediate. This is the reason why the queue size can be controlled when the throughput of clients POST is high.
2. this strategy does not improve total throughput which is reasonable as the round-trip time and the system resources are not improved. However, if we don't have suitable parameters (e.g., circuit-breaker threshold, maximum time of sleep, etc.), the results are likely to be negative.
3. Actually, for 256 threads, P99 percentile response delay becomes large after mitigation. It can be explained as a certain number of POST requests are delayed during peak phase and they are cached by the sleep threads in Tomcat. They are handled later and their response time grow. The number of delayed requests is more than 1% in 256 threads scenario.



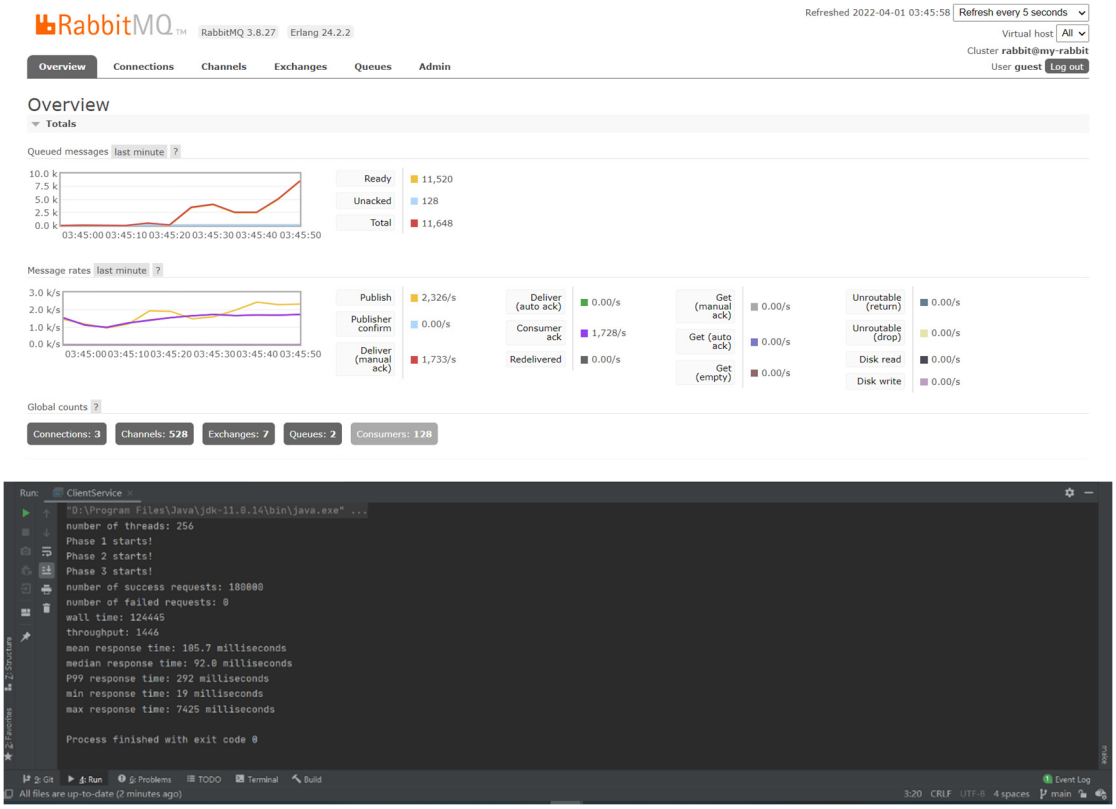
Skier Service 128 threads



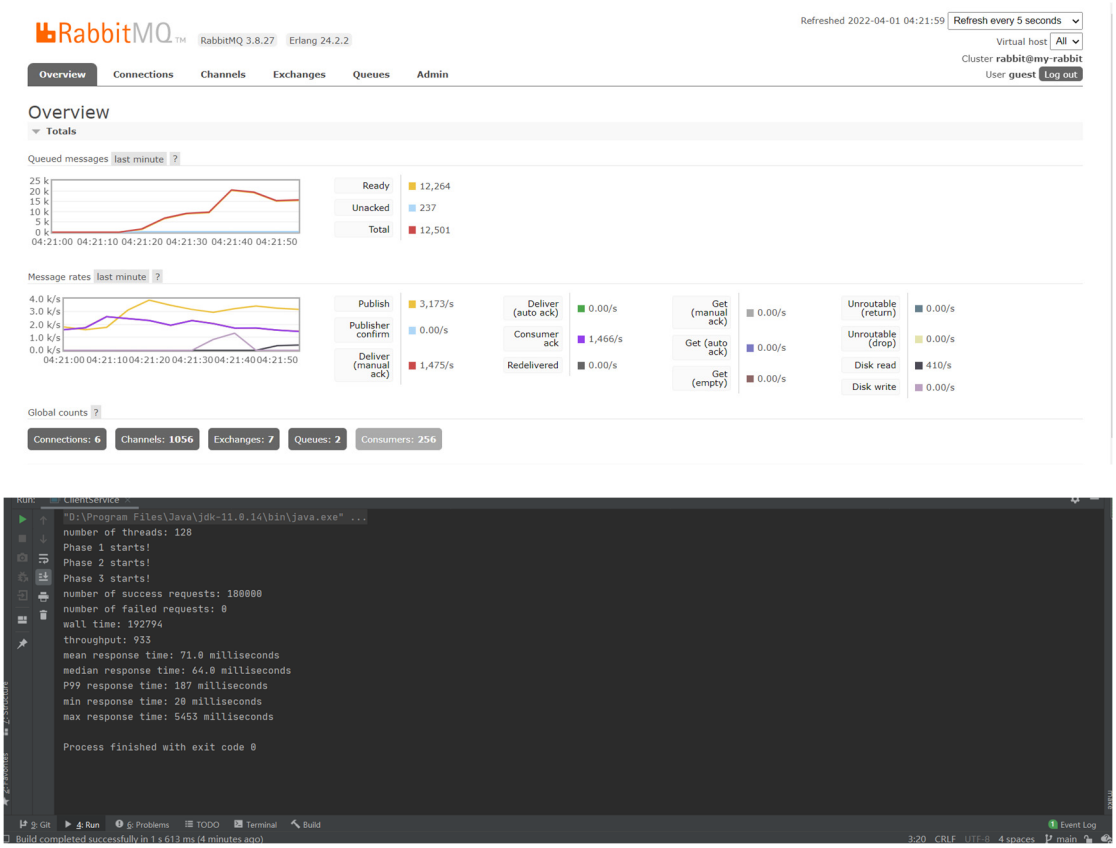
Skier Service 256 threads



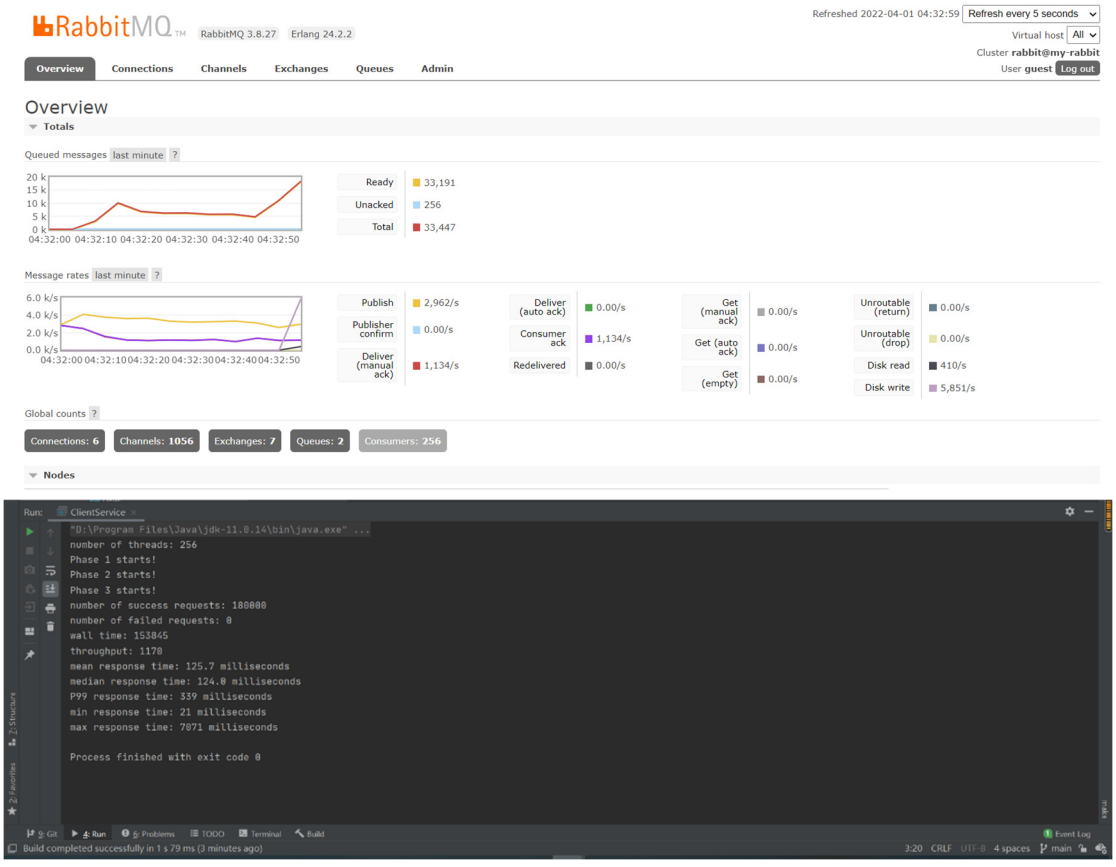
Resort Service 128 threads



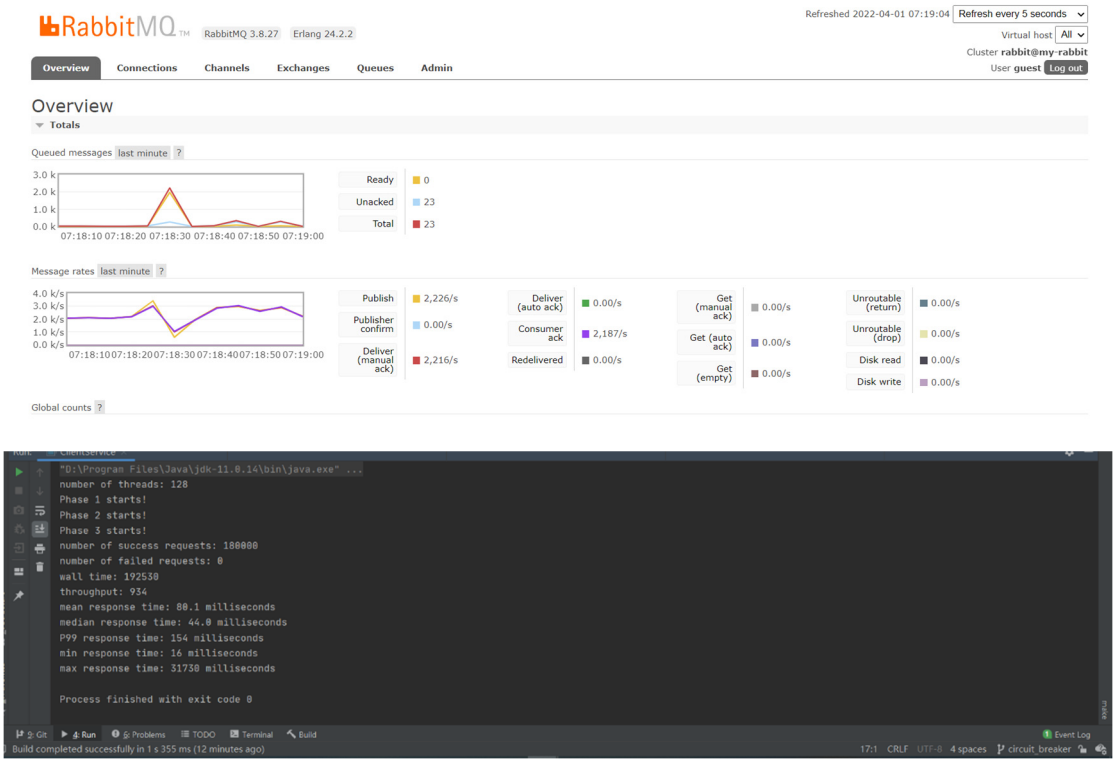
Resort Service 256 threads



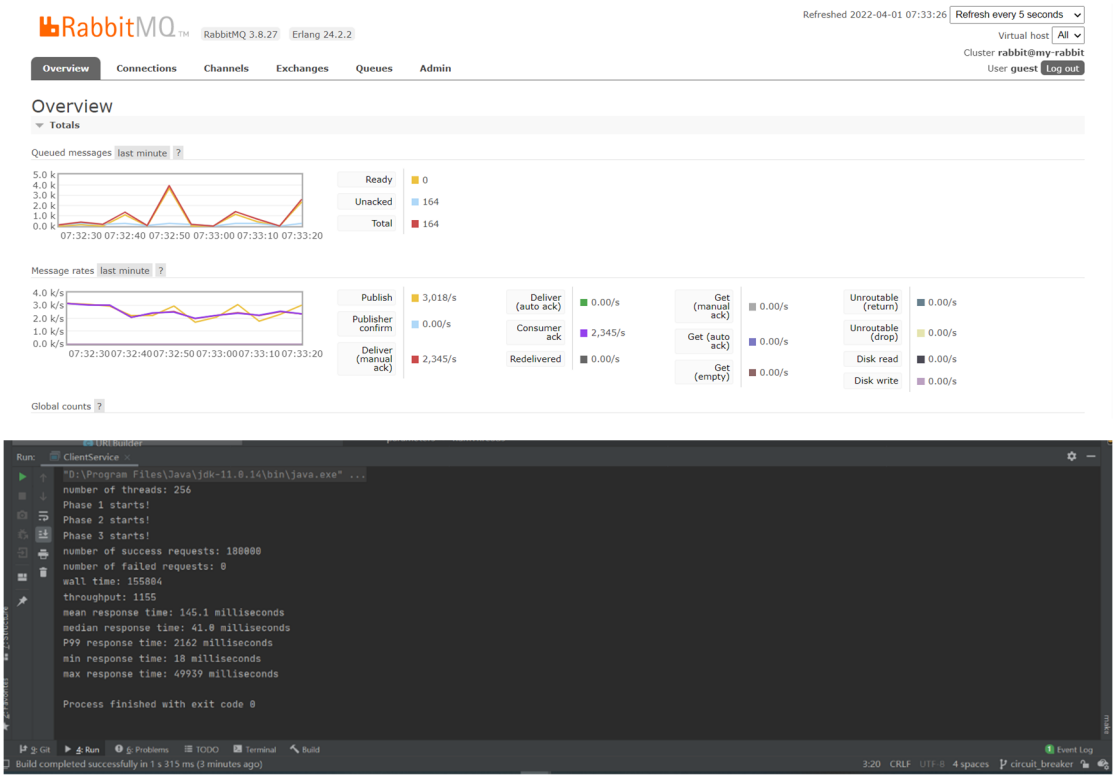
Both Services 128 threads



Both Services 256 threads



Both Services 128 threads After mitigation



Both Services 256 threads After mitigation