6650 Assignment 2

URL: https://github.com/tjuqxb/6650-assignment2

Main Classes In Server

ResortServlet:This class handles the requests to /resorts/* route. It is responsible for validating the paths and parameters. It also checks the format of JSON POST.
SkierServlet:This class handles the requests to /skiers/* route. It is responsible for validating the paths and parameters. It also checks the format of JSON POST. In init(), it also creates a connection to RabbitMQ and initializes a channel pool.
StatisticsServlet: This class handles the requests to /statistics/* route. It is responsible for validating the paths and parameters.
Utils: This class contains static common utility functions.
SkierPost: This class can be mapped to a standard POST JSON object specified by the skier API.
ResortPost: This class can be mapped to a standard POST JSON object specified by the resort API.

Packages

domain: This package is used to store classes which can be mapped to standard POST JSON objects specified by the API. It currently contains two class: SkierPost and ResortPost.

Messages Send and Receive

Send: In SkierServlet: in init() method, a connection is set up at the initial stage. A BlockingQueue is utilized as the thread safe container of the Channel pool. 200 Channels are created and added to the BlockingQueue. In doPost() method, if the JSON string is valid, the program would encode the skier ID and the JSON string together as String message. Then it would be blocked and apply for a Channel object from the BlockingQueue until it gets one available Channel. After setting up some parameters, the encoded message would be sent to the Channel. Then the Channel would be returned to the BlockingQueue.
Receive: The user would input the number of consumer threads to the main entry point in ConsumerService. Then in main() method the program would put the same number of threads to the thread pool. Every single Consumer thread would own its own Channel and keep consuming messages from the Channel. Every message Object would be in a Vector and every Vector is associated with a skier ID in a ConcurrentMap. Those two data structures are thread safe in multi-thread programming.

Results and analysis

Analysis on throughput:
I run the loading tests from local laptop. The loading tests run 64, 128, 256, 512 client

threads to exert load on 1 instance, 2 instances and 4 instances (t2 micro instance). There are some analyses:

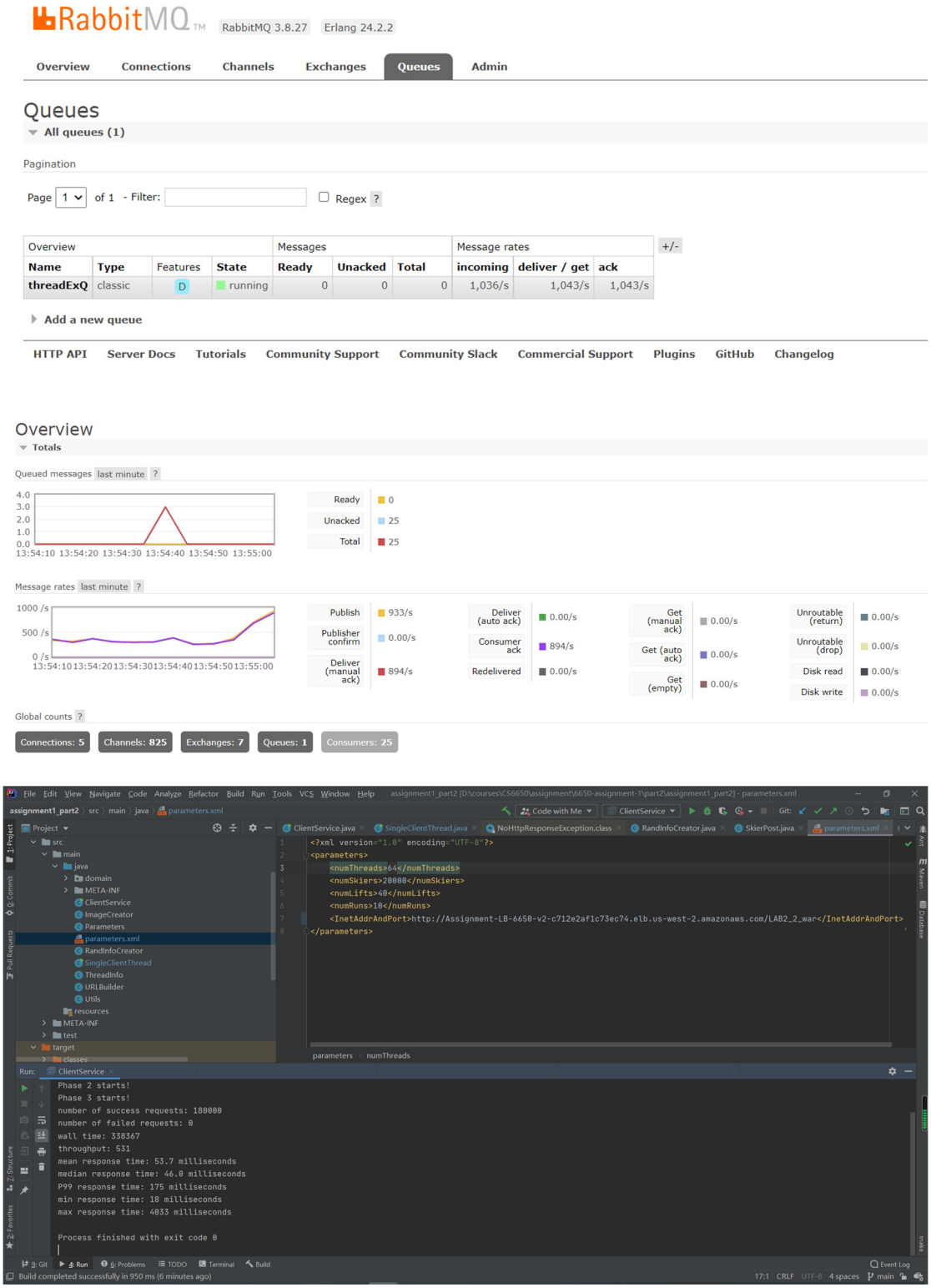
1. It can be seen that it is necessary to scale out when we have large number of client threads (>= 256). We don't need to scale out when the number of client threads is small. Although it might also increase throughput a bit with small number of clients, the cost is too high in comparison to the benefits.
2. For one instance throughput, if the number of base client threads is large, then it is more likely to reach the throughput limit in each phase. This explains the small difference between 256 threads and 512 threads on one instance throughput.
3. It can be seen that for 64 threads and 128 threads, one EC2 t2 micro instance is enough. For 256 threads, we might need two t2 micro instances for better performance. For 512 threads, we might need four t2 micro instances.
4. RabbitMQ limit is its memory. A free tier's memory might not be enough for large throughput. I use t2.medium for larger memory.
5. I also run loading test application on an EC2 instance from another availability zone in the same region. 64 threads throughput would increase rapidly with a greater number of instances. We can deduce that the latency is quite low when testing from an EC2 instance in the same region.

Throughputs (screenshots for 4 instances are attached):

For 64 threads: 1 instance: 481; 2 instances: 481; 4 instances: 531
For 128 threads: 1 instance: 751; 2 instances: 792; 4 instances: 821
For 256 threads: 1 instance: 881; 2 instances: 1436; 4 instances: 1503
For 512 threads: 1 instance: 987; 2 instances: 1533; 4 instances: 2013

Analysis on number of consumer threads:

One consumer thread can handle about 1.5k/s incoming rate. After testing, if we need to handle 4k to 6k incoming rates and keep the queue short, we might need more than 50 consumer threads.



64 threads 4 instances

RabbitMQ™ RabbitMQ 3.8.27 Erlang 24.2.2

Overview | Connections | Channels | Exchanges | Queues | Admin

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	
threadExQ	classic	D	running	0	0	0	1,419/s	1,412/s	1,413/s	

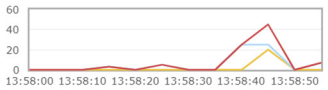
Add a new queue

HTTP API | Server Docs | Tutorials | Community Support | Community Slack | Commercial Support | Plugins | GitHub | Changelog

Overview

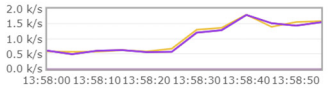
Totals

Queued messages last minute ?



Ready 0
Unacked 15
Total 15

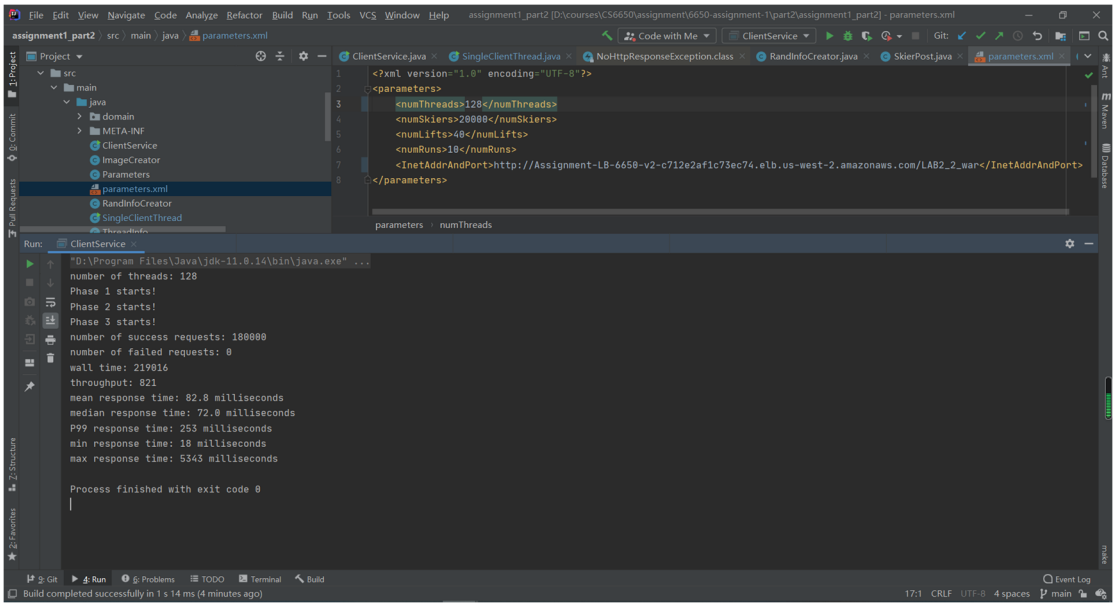
Message rates last minute ?



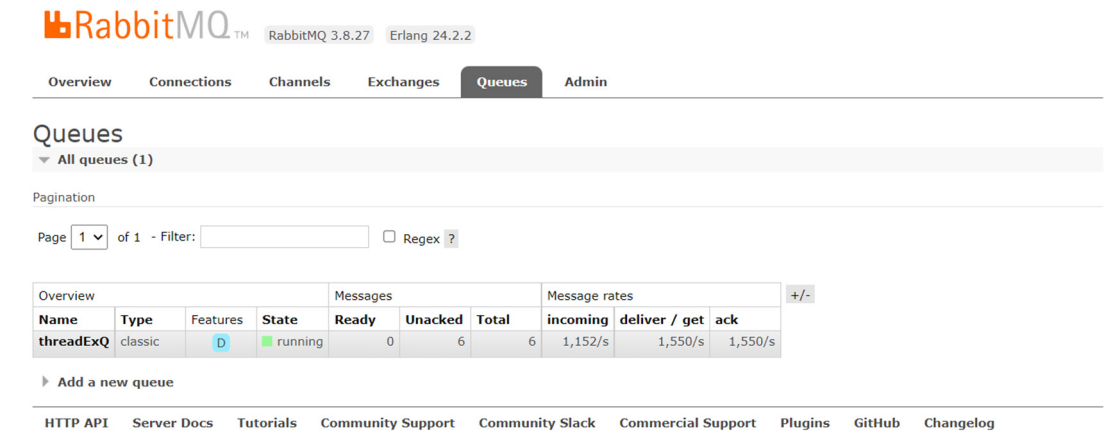
Publish	1,585/s	Deliver (auto ack)	0.00/s	Get (manual ack)	0.00/s	Unroutable (return)	0.00/s
Publisher confirm	0.00/s	Consumer ack	1,552/s	Get (auto ack)	0.00/s	Unroutable (drop)	0.00/s
Deliver (manual ack)	1,551/s	Redelivered	0.00/s	Get (empty)	0.00/s	Disk read	0.00/s
						Disk write	0.00/s

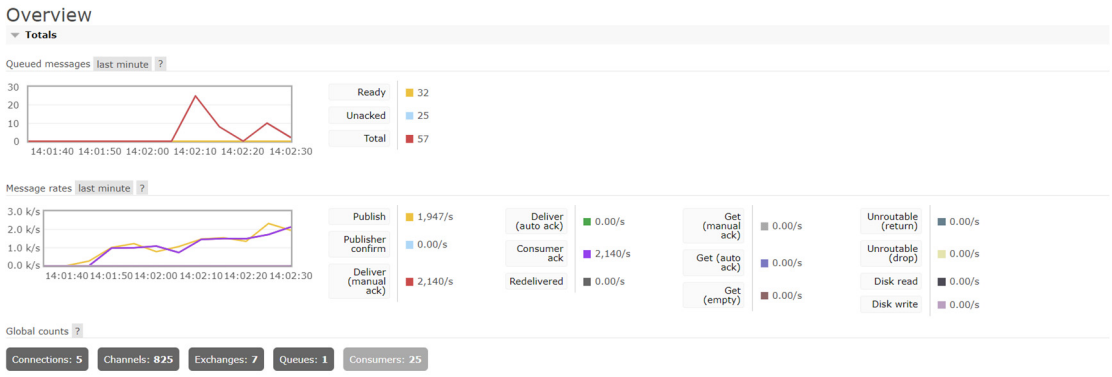
Global counts ?

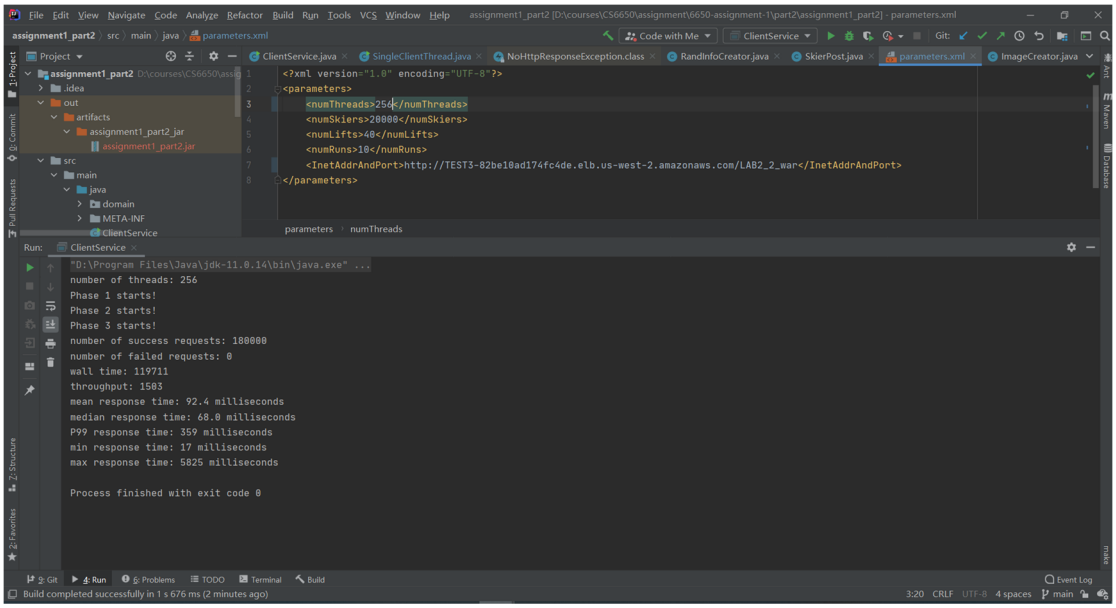
Connections: 5 | Channels: 825 | Exchanges: 7 | Queues: 1 | Consumers: 25



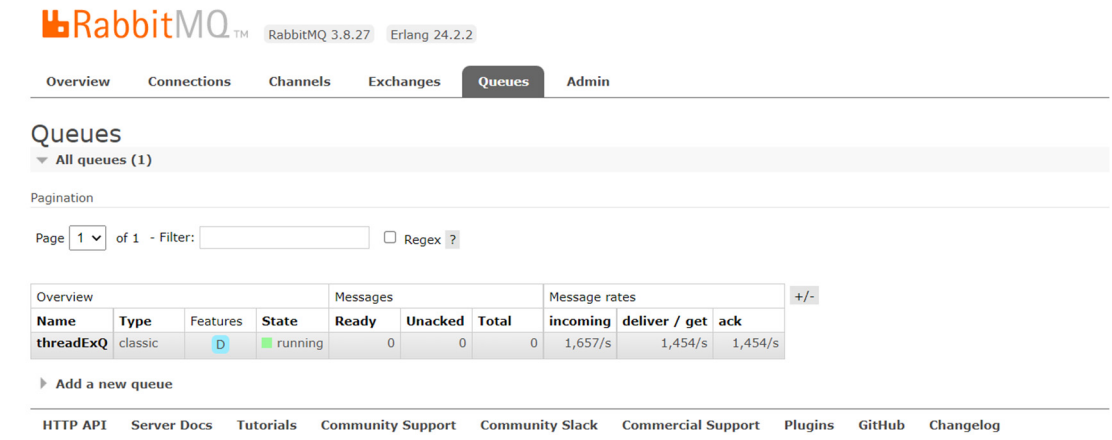
128 threads 4 instances

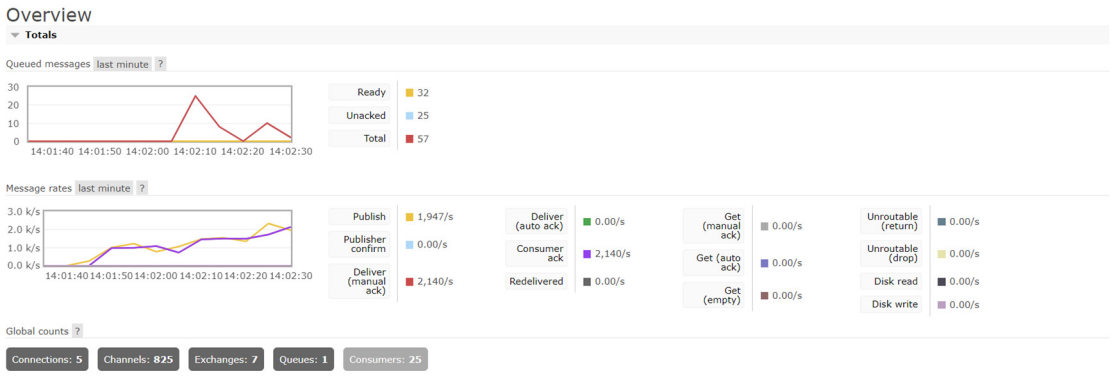


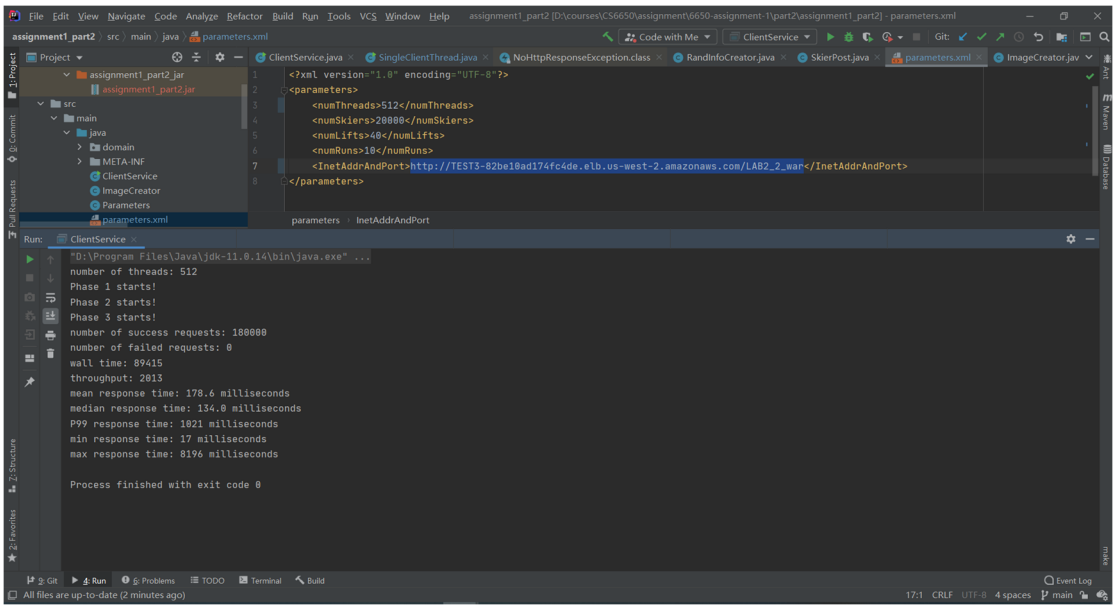




256 threads 4 instances







512 threads 4 instances