Assignment 2

Yinlu Gong

1.
   Personal uRL: https://github.com/zoegongyinlu/Distributed-System-6650/tree/main/A1
   Khoury github url:
   https://github.khoury.northeastern.edu/yinlugong16/CS6650Spring2025

Overview

Updated from previous submission: I have added codes to start phase 2 early without waiting for phase 1 fully complete.

The server design implements a distributed system for processing ski resort lift ride events. It leverages RabbitMQ for message queuing to handle high throughput and provide resilience. The application follows a servlet-based architecture that receives HTTP requests, validates them, publishes messages to RabbitMQ queues, and has separate consumer components that process these messages asynchronously. Elastic load balancer is added in front of 3 servlet instances.

~~~
#### Configuration

The BASE_PATH in PostRequestSingleThread should be updated based on the desired target:

```java
private static final String BASE_PATH = "http://44.233.246.8:8080/A1_war/"; // EC2 instance
private static final String BASE_PATH = "http://servlet-ELB-422677375.us-west-2.elb.amazonaws.com/A1_war"; //ELB
private static final String BASE_PATH = "http://localhost:8080/A1_war_exploded"; // Localhost
```
~~~

Architecture Components

Packages and Major Classes

1. **`cs6650Spring2025.assignment2server`**
   - **`SkierServlet`**: Main entry point for HTTP requests, handles request validation and publishing messages to RabbitMQ
   - **`RabbitMQChannelPool`**: Manages a pool of RabbitMQ channels for efficient message publishing
   - **`ChannelFactory`**: Creates and validates RabbitMQ channels for the pool

   - **`LiftRideMessageConsumer`**: Consumes lift ride events from RabbitMQ and processes them
   - **`ApplicationShutDownListener`**: Ensures proper cleanup of resources during application shutdown
2. **`cs6650Spring2025.clientPart1`**
   - **`SkierClientMain`**: The `SkierClientMain` class manages the execution of multiple threads that send HTTP POST requests. It calculates and prints performance metrics, and saves request performance statistics using RecordWriter.

`Note:`

The class **`ChannelFactory, RabbitMQChannelPool`** were adapted from:

https://github.com/gortonator/foundations-of-scalable-systems/blob/main/Ch7/rmqpool/RMQChannelFactory.java

https://github.com/gortonator/foundations-of-scalable-systems/blob/main/Ch7/rmqpool/RMQChannelPool.java

Key Relationships

1. **Web Request Flow**:
   - `SkierServlet` receives HTTP POST requests from clients
   - Validates request parameters and JSON body
   - Uses `RabbitMQChannelPool` to obtain a channel for message publishing
   - Publishes validated events to a specific RabbitMQ queue based on skier ID
2. **Channel Pooling Mechanism**:
   - `RabbitMQChannelPool` maintains a connection to RabbitMQ and a pool of channels: Channels are borrowed, used, and returned to the pool for efficient resource utilization
   - `ChannelFactory` creates and manages the lifecycle of individual channels
3. **Message Consumption**:
   - `LiftRideMessageConsumer` runs independently from the servlet
   - Maintains multiple consumer threads that process messages from designated queues
   - Stores processed data in thread-safe data structures (`ConcurrentHashMap` with skierID as key and `CopyOnWriteArrayList as the value`)

Message Flow

1. **Request Reception**:

- SkierClientMain sends a POST request to `/skiers/{resortID}/seasons/{seasonID}/days/{dayID}/skiers/{skierID}`
- `SkierServlet.doPost()` validates the URL format and request body
- Extracts parameters to create a `LiftRideEvent` object

2. **Message Publishing**:
   - The event is serialized to JSON using Gson
   - A RabbitMQ channel is borrowed from the `RabbitMQChannelPool`
   - The message is published to a specific queue based on skier ID for load distribution
   - `channel.basicPublish("", queueName, null, messageBody.getBytes())`
   - The channel is returned to the pool after use (or invalidated if an error occurs)
3. **Message Consumption**:
   - `LiftRideMessageConsumer` maintains multiple consumer threads (configurable, default is 2 per queue, default 7 message queues concurrently consume)
   - Each consumer subscribes to a specific queue with a prefetch count to control throughput
   - When a message is received, it's deserialized into a `LiftRideEvent` object
   - The event data is stored in a thread-safe map for later retrieval: `skierIdToRecords.computeIfAbsent(liftRideEvent.getSkierID(), k -> new CopyOnWriteArrayList<>())`
   - Message is acknowledged once successfully processed
4. **Resource Management**
   - ApplicationShutDownListener monitors for application shutdown
   - On shutdown, it ensures proper cleanup of:
     1. RabbitMQ channel pool
     2. Consumer threads
     3. Other application resources

Scalability Features

1. **Multiple Queues**:
   - System uses multiple queues (default 7) to distribute load
   - Messages are routed to queues based on skier ID (mod operation)
   - Each queue has multiple consumer threads for parallel processing
2. **Connection and Channel Management**:
   - Connections are shared across the application
   - Channels are pooled for efficient resource utilization
   - Pool size is configurable (default max 50 channels)
   - Automatic recovery is enabled for network resilience
3. **Asynchronous Processing**:
   - Web requests are decoupled from data processing via message queuing
   - HTTP responses are returned immediately after message publishing

    - Message processing occurs asynchronously
4. **Thread Safety**:
    - Thread-safe data structures (`ConcurrentHashMap`, `CopyOnWriteArrayList`, `AtomicInteger`)
    - Thread pools with controlled sizes to prevent resource exhaustion
    - Thread synchronization using countdown latches for coordinated startup/shutdown

Error Handling

1. **Request Validation doPost() url pattern**:
    - URL format validation before processing
    - JSON payload validation
    - Appropriate HTTP error codes returned to clients
2. **RabbitMQ Connection Resilience**:
    - Automatic connection recovery enabled
    - Configurable recovery delay and heartbeat
    - Channel validation before reuse
3. **Message Processing**:
    - Error tracking with `AtomicInteger`
    - Exception handling with message requeuing when appropriate
    - Manual acknowledgment to ensure processing guarantees

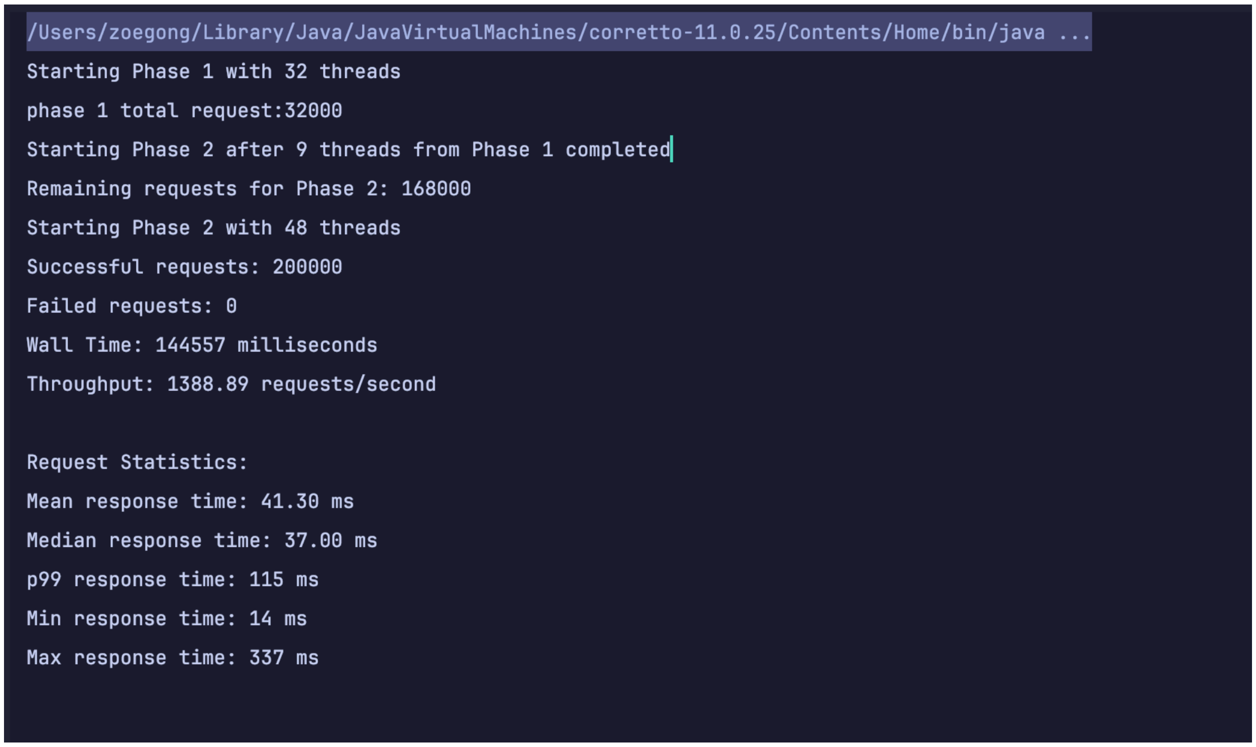
Resource Management

1. **Graceful Shutdown**:
    - `ApplicationShutDownListener` ensures proper cleanup
    - Channels and connections closed during application shutdown
    - Thread pools properly terminated
2. **Channel Management**:
    - Channels validated on borrow and return
    - Invalid channels removed from the pool
    - Pool configuration optimized for the application's workload
    -

Result:

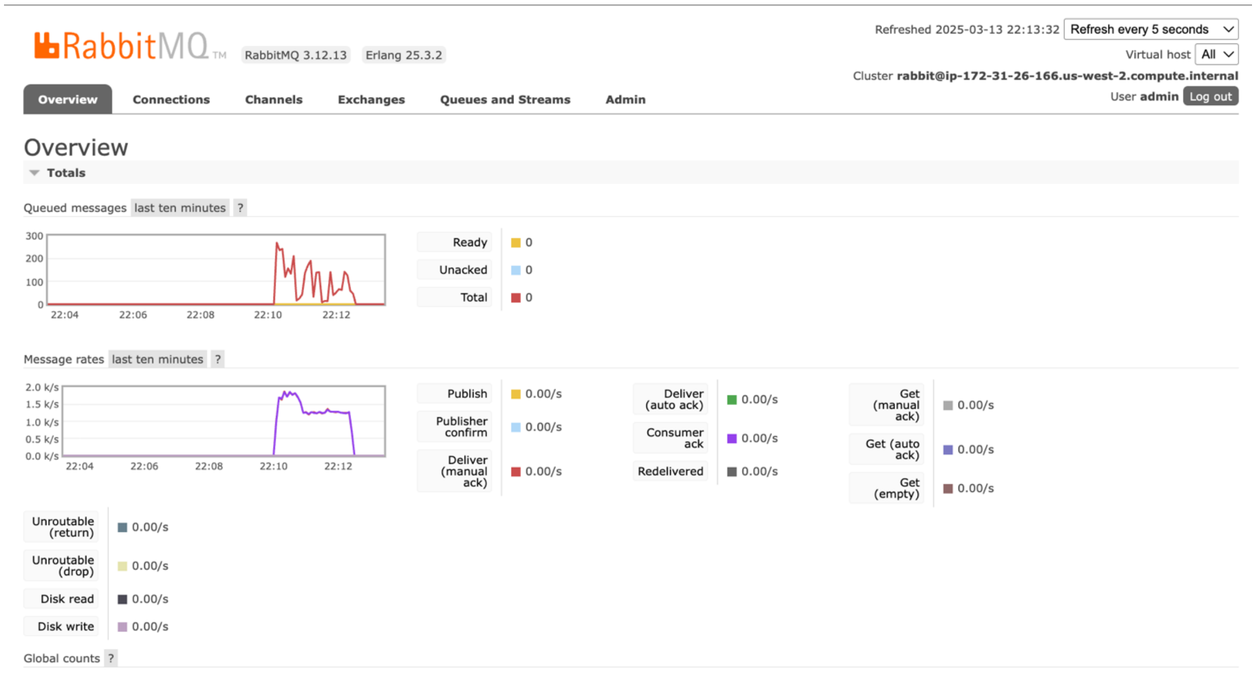
1. Single Servlet:

Metrics:



```
/Users/zoegong/Library/Java/JavaVirtualMachines/corretto-11.0.25/Contents/Home/bin/java ...
Starting Phase 1 with 32 threads
phase 1 total request:32000
Starting Phase 2 after 9 threads from Phase 1 completed
Remaining requests for Phase 2: 168000
Starting Phase 2 with 48 threads
Successful requests: 200000
Failed requests: 0
Wall Time: 144557 milliseconds
Throughput: 1388.89 requests/second

Request Statistics:
Mean response time: 41.30 ms
Median response time: 37.00 ms
p99 response time: 115 ms
Min response time: 14 ms
Max response time: 337 ms
```



RabbitMQ 3.12.13 Erlang 25.3.2

Refreshed 2025-03-13 22:13:32 Refresh every 5 seconds

Virtual host All

Cluster **rabbit@ip-172-31-26-166.us-west-2.compute.internal**

User **admin**

Overview

Queued messages last ten minutes ?

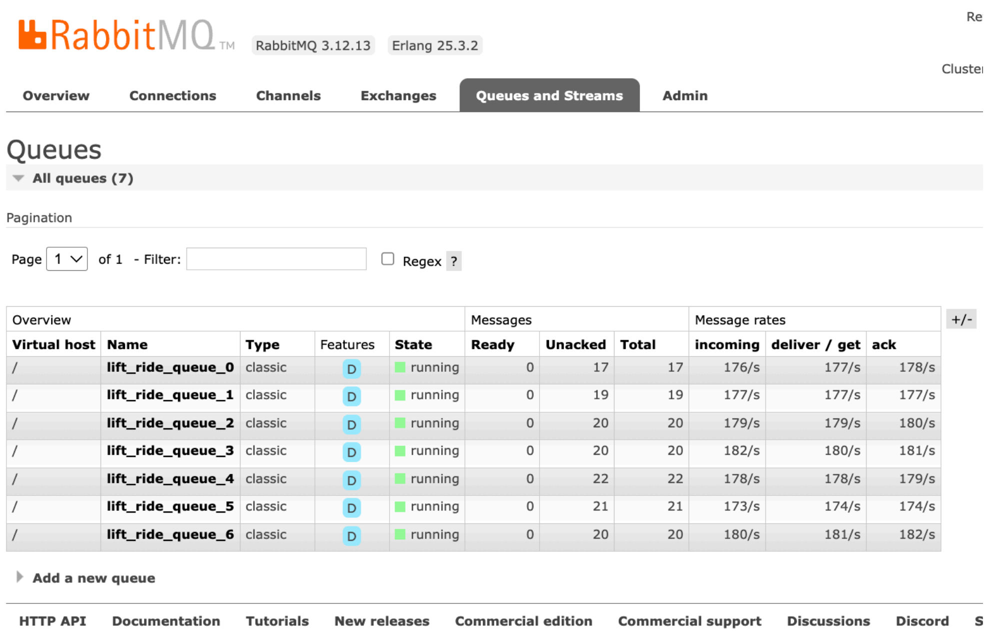
Ready	0
Unacked	0
Total	0

Message rates last ten minutes ?

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

Unroutable (return)	0.00/s
Unroutable (drop)	0.00/s
Disk read	0.00/s
Disk write	0.00/s

Global counts ?



RabbitMQ 3.12.13 Erlang 25.3.2

Queues

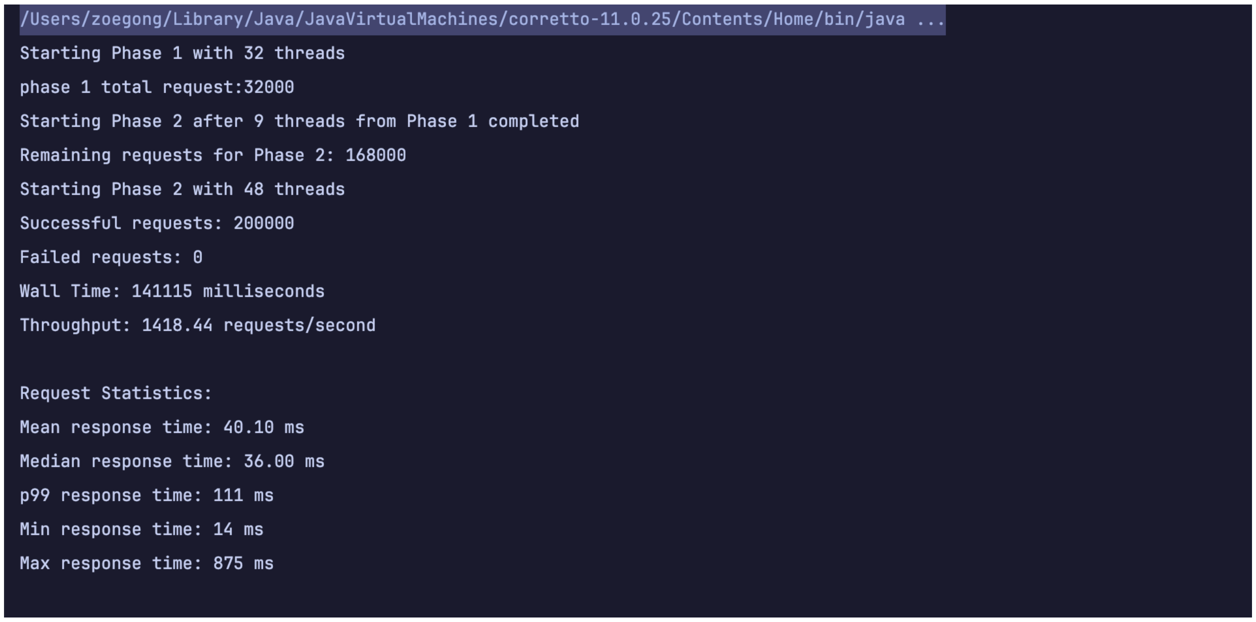
All queues (7)

Pagination

Page 1 of 1 - Filter: Regex ?

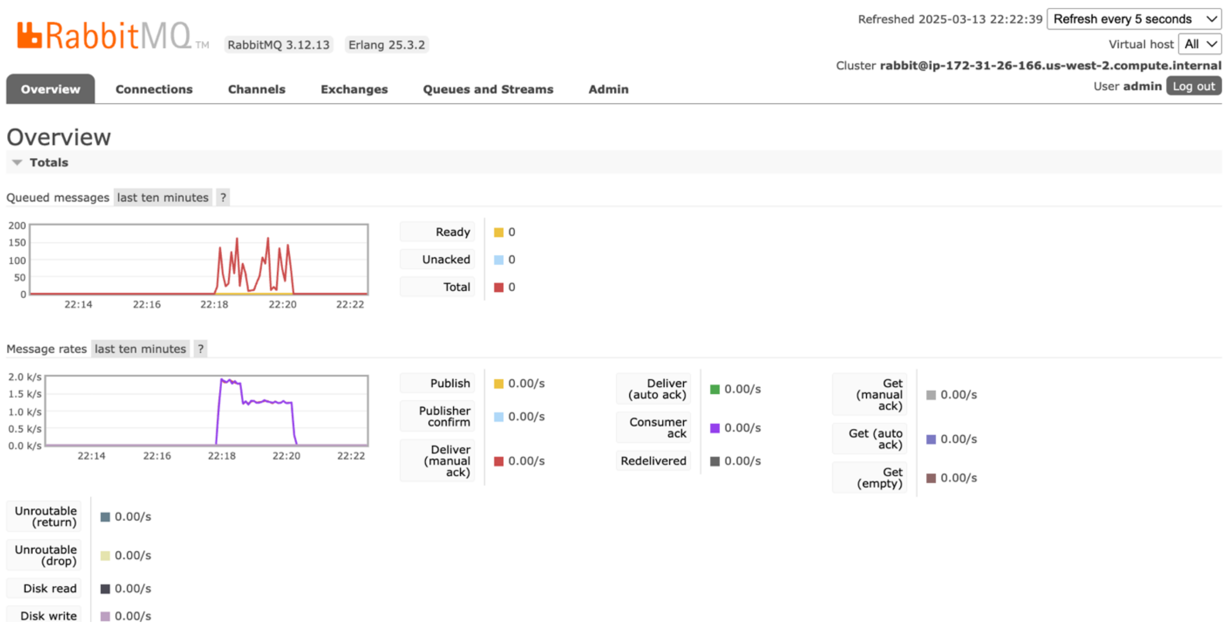
Overview					Messages			Message rates		
Virtual host	Name	Type	Features	State	Ready	Unacked	Total	incoming	deliver / get	ack
/	lift_ride_queue_0	classic	D	running	0	17	17	176/s	177/s	178/s
/	lift_ride_queue_1	classic	D	running	0	19	19	177/s	177/s	177/s
/	lift_ride_queue_2	classic	D	running	0	20	20	179/s	179/s	180/s
/	lift_ride_queue_3	classic	D	running	0	20	20	182/s	180/s	181/s
/	lift_ride_queue_4	classic	D	running	0	22	22	178/s	178/s	179/s
/	lift_ride_queue_5	classic	D	running	0	21	21	173/s	174/s	174/s
/	lift_ride_queue_6	classic	D	running	0	20	20	180/s	181/s	182/s

2. With elastic load balancer:



```
/Users/zoegong/Library/Java/JavaVirtualMachines/corretto-11.0.25/Contents/Home/bin/java ...
Starting Phase 1 with 32 threads
phase 1 total request:32000
Starting Phase 2 after 9 threads from Phase 1 completed
Remaining requests for Phase 2: 168000
Starting Phase 2 with 48 threads
Successful requests: 200000
Failed requests: 0
Wall Time: 141115 milliseconds
Throughput: 1418.44 requests/second

Request Statistics:
Mean response time: 40.10 ms
Median response time: 36.00 ms
p99 response time: 111 ms
Min response time: 14 ms
Max response time: 875 ms
```



Refreshed 2025-03-13 22:22:39 Refresh every 5 seconds

Virtual host All

Cluster **rabbit@ip-172-31-26-166.us-west-2.compute.internal**

User **admin**

Overview

Totals

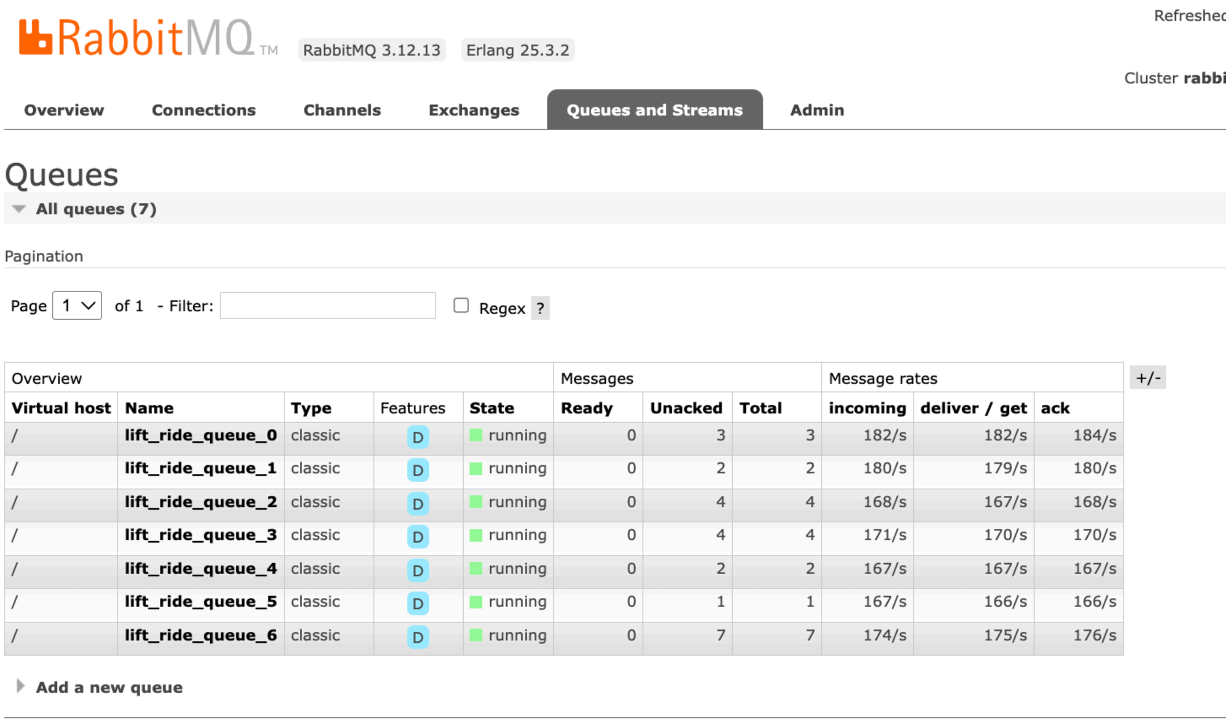
Queued messages last ten minutes ?

Ready	0
Unacked	0
Total	0

Message rates last ten minutes ?

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

Unroutable (return)	0.00/s
Unroutable (drop)	0.00/s
Disk read	0.00/s
Disk write	0.00/s



RabbitMQ™ RabbitMQ 3.12.13 Erlang 25.3.2

Refreshed

Cluster **rabbi**

Queues

All queues (7)

Pagination

Page 1 of 1 - Filter: ☐ Regex ?

Overview					Messages			Message rates		
Virtual host	**Name**	**Type**	Features	**State**	**Ready**	**Unacked**	**Total**	**incoming**	**deliver / get**	**ack**
/	**lift_ride_queue_0**	classic	D	🟩 running	0	3	3	182/s	182/s	184/s
/	**lift_ride_queue_1**	classic	D	🟩 running	0	2	2	180/s	179/s	180/s
/	**lift_ride_queue_2**	classic	D	🟩 running	0	4	4	168/s	167/s	168/s
/	**lift_ride_queue_3**	classic	D	🟩 running	0	4	4	171/s	170/s	170/s
/	**lift_ride_queue_4**	classic	D	🟩 running	0	2	2	167/s	167/s	167/s
/	**lift_ride_queue_5**	classic	D	🟩 running	0	1	1	167/s	166/s	166/s
/	**lift_ride_queue_6**	classic	D	🟩 running	0	7	7	174/s	175/s	176/s

+/-

Add a new queue