Git Repo URL

https://github.com/Syl-Ying/Skier/tree/master/A2

Server Design

1-2 page description of your server design. Include major classes, packages, relationships, how messages get sent/received

Overview

This server design is intended for processing lift ride data efficiently using **RabbitMQ** as the messaging broker. The main purpose of the server is to consume messages, represent data for skier lift rides, and store it for further processing or analytics. The design aims for scalability and reliability by utilizing RabbitMQ's message broker capabilities and Java concurrency for message consumption.

Service Deployment on EC2

The calling chain: Client → Servlet → RabbitMQ → Consumer

- **Client**: Local machine.
- **Servlet server:** Deployed on **2** EC2 `t2.micro` instances.
- **RabbitMQ**: Deployed on 1 EC2 `t2.micro` instance.
- **Consumer:** Deployed on 1 EC2 `t3.large` instance using `Docker`. Because t2.micro only has 1 GB RAM, an out-of-memory (OOM) error occurred when running the Docker image on t2.micro, so a t3.large instance with 8 GB RAM was chosen.

Class and Package Relationships

1. **Client package**

a. Local client with 200 threads in total. 32 thread at first , then 168 threads. each sending 1k request.

b. **A Single Producer**

- `LiftRideEvent` class: a lift ride event.
- `LiftRideProducer` class: generate 200k random `LiftRideEvent` instances and store them in a thread-safe queue for the worker threads to consume.

- **Multithreading Consumers**:
  - `HttpPostThread` class: a Runnable class to send given numbers of requests. Each POST request will retry up to 5 times upon receiving 4XX or 5XX errors.
  - `SkierClient` class: the main class
    - First use a single thread to generates 200k events.
    - Then 32 initial threads, each sending 1000 POST requests, will start.
    - Once those threads finish, 168 threads will be created until all 200K requests are sent.
- **Performance Metrics**: Track the number of successful and unsuccessful requests, the total runtime, and calculate throughput (requests per second).
  - `Record` class: a request's status code, starting time…
  - `ReportProcessor` class
    - First calculate related metrics and print to terminal.
    - Then output a CSV file with all the records

2. **Server package**

- **SkierServlet** class
  - initializes a **RabbitMQ connection** and a **pool of 30 channels** during startup. Each channel publishes messages concurrently. This setup helps in handling multiple client requests simultaneously without overwhelming a single channel.
  - Servlet workflow:
    1. validating incoming requests (both URL parameters and JSON body)
    2. formatting messages
    3. sending formatted messages to RabbitMQ using the available channels from the pool.

4. After processing the request, it sends an appropriate response back to the client.

3. **Consumer package:** Responsible for individual message consumption, storing messages in a thread safe map.
    a. **LiftRideConsumer** class
        - Starting point of the consumer application. It is responsible for establishing the connection to RabbitMQ, creating `25` worker tasks, and delegating the actual processing to worker threads.
        - One connection, multiple channel: It launches **multiple Worker** instances, each acting as a message consumer.
    b. **Worker** class
        - The Worker class represents a single consumer parsing and storing the lift ride data.. Each worker runs in its own thread and continuously listens for new messages from RabbitMQ.
        - It creates a **Channel** from the RabbitMQ connection to consume messages from the queue.
        - It uses acknowledgments to manage the queue size, accepting only 10 unacknowledged messages at a time to avoid overloading the consumer.
    c. **Constant** class
        - The Constant class defines static configurations like RabbitMQ's IP address, port, username, password, and queue name. This allows for better modularity and easy configuration.

Communication Flow - Message Send/Receive

1. **Message Publishing:** Local client generates 200k **lift rides and sends to servlet.** Servlet receives the http request and publishes them as Messages to a queue in RabbitMQ. Each message typically contains the details of a lift ride, including skier ID, resort ID, time, lift ID, and day.
2. **Connection and Worker Setup:** Consumer establishes the **RabbitMQ connection** and starts 25 **worker threads**, allowing multiple messages to be processed concurrently.
3. **Message Receiving and Processing**
    - Each worker, running in a separate thread, creates a **Channel** to communicate with RabbitMQ. The worker sets up a consumer, which listens to the specified queue for

incoming messages.

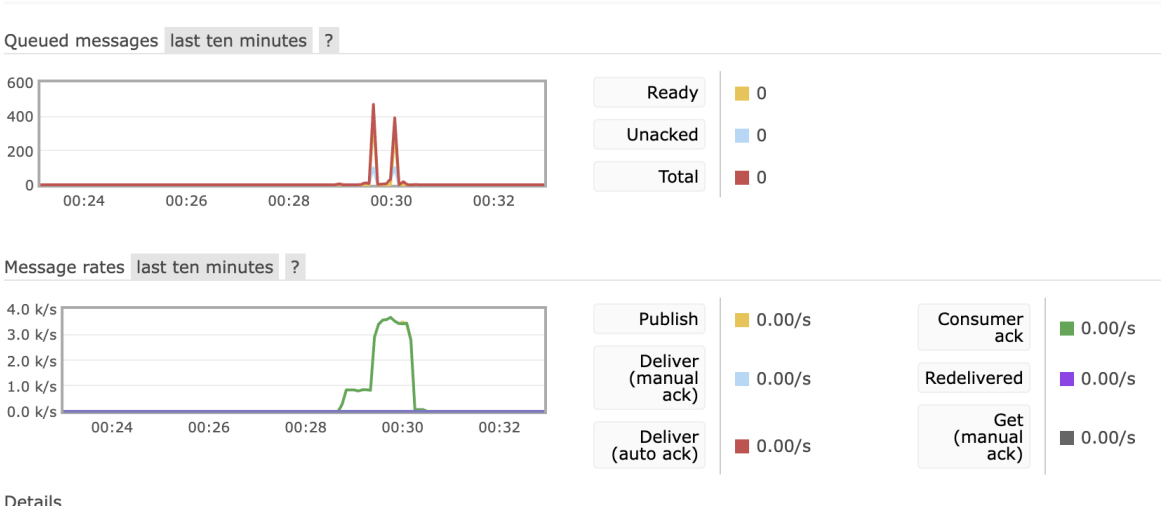
- A worker only processes 10 **unacknowledged message at a time**, improving reliability by avoiding message overload.

4. **Message Handling**
   - Upon receiving a message, the `deliverCallback` is triggered. The callback function deserializes the JSON message and processes the message by extracting relevant information (such as skier ID and lift ride details) and storing it in an in-memory **ConcurrentHashMap**, which keeps the data of lift rides associated with each skier.
   - The **ConcurrentHashMap** is used to ensure **thread safety**, allowing multiple worker threads to store and update lift ride data concurrently without data corruption.
5. **Acknowledgment**
   - After successfully processing the message, the worker sends an acknowledgment. This tells RabbitMQ that the message has been processed and can be removed from the queue.

Test run results

(command lines showing metrics, RMQ management windows showing queue size, send/receive rates) for a single servlet showing your best throughput.

```
---------------------- Report Client1 ----------------------
Number of successful requests: 200,000
Number of failed requests: 0
Wall time for all 200k requests using 200 threads: 102,182 ms
Actual throughput (RPS): 1,957.29
----------------------  Report End  ------------------------

---------------------- Report Client2 ----------------------
Mean response time (millisecs): 46
Median response time (millisecs): 44
p99 (99th percentile) response time: 44
Min response time (millisecs): 18
Max response time (millisecs): 288
Throughput (RPS): 1,959.29
---------------------- Report End --------------------------
```



Test run results

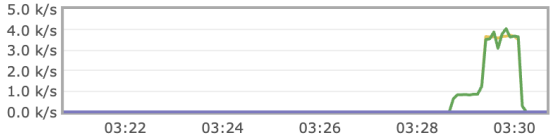
(command lines showing metrics, RMQ management windows showing queue size, send/receive rates) for a load balanced servlet showing your best throughput.

```
Start generating lift ride events!

Created 32 threads.
One of the initial 32 threads has completed. Starting 168 threads...
Created 168 threads.
---------------------- Report Client1 ----------------------
Number of successful requests: 200,000
Number of failed requests: 0
Wall time for all 200k requests using 200 threads: 85,855 ms
Actual throughput (RPS): 2,329.51
----------------------  Report End  ------------------------

---------------------- Report Client2 ----------------------
Mean response time (millisecs): 45
Median response time (millisecs): 43
p99 (99th percentile) response time: 46
Min response time (millisecs): 18
Max response time (millisecs): 669
Throughput (RPS): 2,332.14
---------------------- Report End --------------------------
```

Queued messages last ten minutes ?

Ready	0
Unacked	0
Total	0

Message rates last ten minutes ?



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

Details

2 EC2 instances