Assignment 2 Report

Github Repo:

https://github.com/CloudyZ524/6650-Distributed-System/tree/main/Assignment/Assignment2

Server Design

Packages and Major Classes:

1. **Servlet Package**
   HttpServlet: The SkierServlet class extends HttpServlet, which provides methods to handle HTTP requests and responses. This is a part of the Java Servlet API.

2. **RabbitMQ Client Package**
   - ConnectionFactory: Used to create a new connection to the RabbitMQ server.
   - Connection: Represents a connection to the RabbitMQ server.
   - Channel: Represents a channel within the connection, used for sending and receiving messages.
   - Queue: Represents a queue in RabbitMQ where messages are stored.

Relationships and Flow:

1. Channel Pool Initialization

The static constructor (static { initializeChannelPool(); }) is used to initialize the channel pool. It creates a fixed number of channels and adds them to the channelPool queue.

2. HTTP Request Handling

The servlet overrides doGet and doPost methods to handle GET and POST requests, respectively.

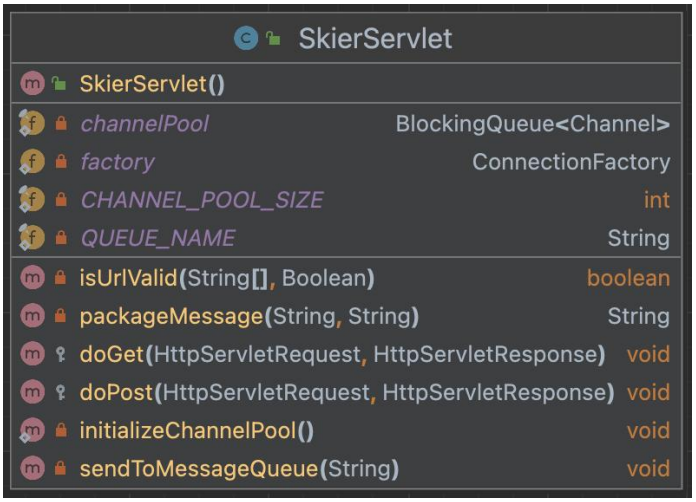
In both methods, the URL path is parsed and validated. If the URL is not valid, a 404 (Not Found) status is returned.

3. Message Packaging and Sending

In the doPost method, if the URL is valid, the request body is read and packaged into a JSON message along with the skier ID.

The sendToMessageQueue method is then called to send this message to the RabbitMQ queue.

The message is sent by polling a channel from the channelPool, publishing the message to the queue, and then offering the channel back to the pool.



Consumer Design

Major Classes:
1. LiftRideConsumer:
This class is responsible for setting up the consumer environment, including the connection to the RabbitMQ server and the channel pool. It also contains utility methods to extract data from the received messages.

- skierLiftRides: A concurrent map to store lift ride data associated with each skier ID.
- factory: A ConnectionFactory used to create a connection to the RabbitMQ server.
- channelPool: A BlockingQueue that holds a pool of channels for consuming messages.
- initializeChannelPool(): A static method to initialize the channel pool.
- getSkierIdFromMessage(String message): Extracts the skier ID from the received JSON message.
- getLiftRideFromMessage(String message): Extracts the LiftRide object from the received JSON message.

2. ConsumerThread:
This class implements the Runnable interface and is responsible for consuming messages from the RabbitMQ queue. Each instance of this class runs in a separate thread.

- channelPool: The shared pool of channels for consuming messages.
- skierLiftRides: The shared map to store lift ride data.
- run(): The method that is executed when the thread starts. It consumes messages from the queue and updates the skierLiftRides map.

Relationships and Flow:
1. Initialization:

The LiftRideConsumer class initializes the channel pool and starts a number of ConsumerThread instances equal to NUM_OF_THREAD. Each thread consumes messages from the RabbitMQ queue.

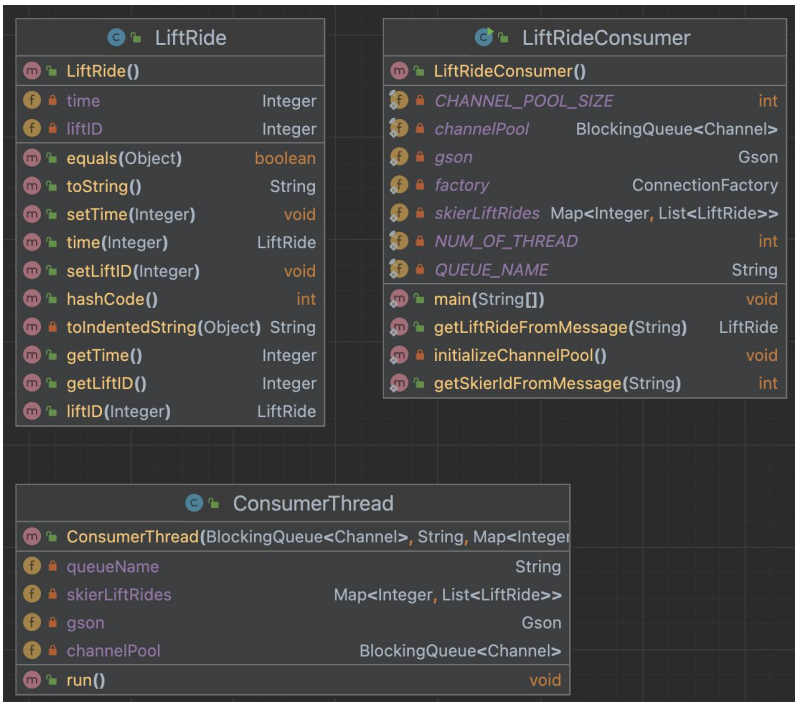
2. Message Consumption:
Each ConsumerThread polls a channel from the channelPool and starts consuming messages from the RabbitMQ queue.

When a message is received, the ConsumerThread parses the message to extract the skier ID and the LiftRide object.

The skierLiftRides map is then updated with the new lift ride data for the corresponding skier ID.

3. Concurrency:
The use of a ConcurrentHashMap for skierLiftRides ensures thread-safe updates to the map.

The channelPool is a BlockingQueue, which provides thread-safe operations for polling and offering channels.



Message Flow:

Sending Messages:

A client sends an HTTP POST request to the servlet with the skier data in the request body. The servlet packages the data into a JSON message and sends it to the RabbitMQ queue using one of the pooled channels.

Receiving Messages:

the LiftRideConsumer class consumes messages from the SkierQueue using multiple ConsumerThread instances. Each thread extracts the skier ID and lift ride information from the JSON message and updates the skierLiftRides map with this data. The concurrent map and multithreaded approach ensure efficient and safe handling of messages.

Design for Optimization

Channel Pooling: The use of a channel pool allows for efficient use of resources when sending and receiving messages to the RabbitMQ queue. By reusing channels, the overhead of creating and closing channels for each message is avoided.

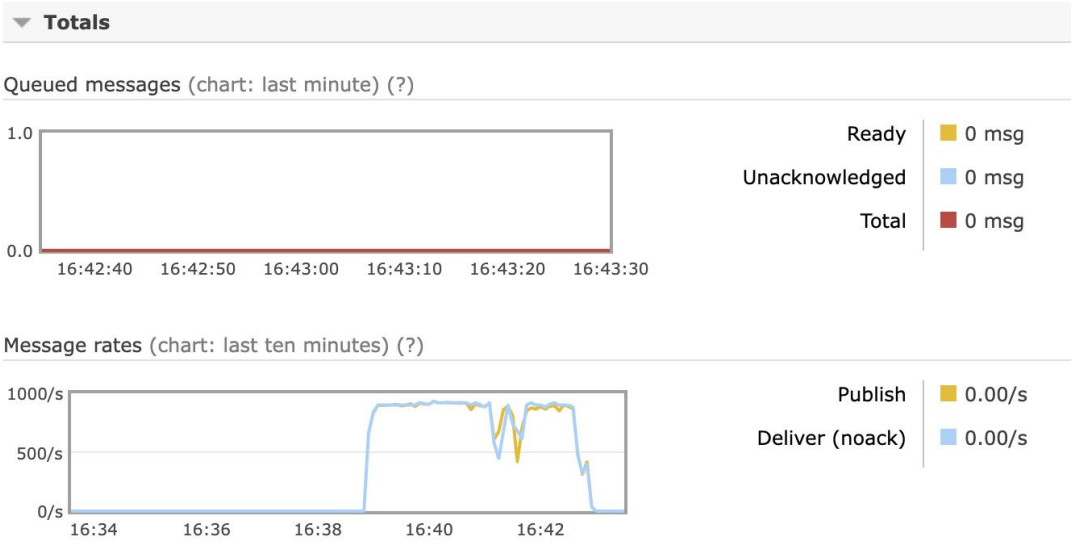
Thread Pooling: The design allows for multiple threads to consume messages concurrently, increasing throughput and improving scalability.

ConcurrentHashMap: The use of thread-safe concurrent data structures ensures that the consumer can handle concurrent updates and channel operations safely.

BlockingQueue: The LinkedBlockingQueue used for the channel pool implementation is thread-safe, ensuring that concurrent access to the pool is handled properly.

Result for single EC2 instance:

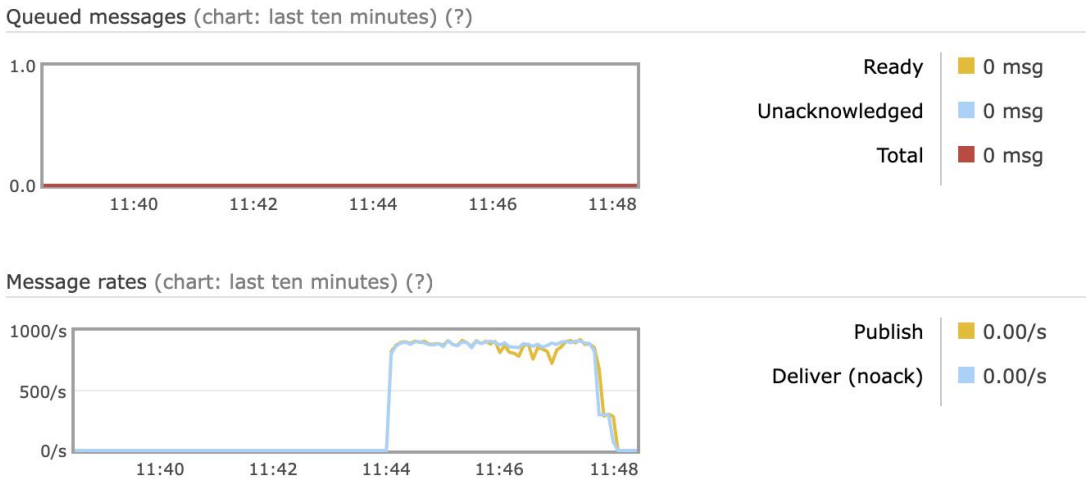
Queue length: 0
Throughput: 826 requests/second
Number of producer channels: 120
Number of consumer channels: 120



```
Successful requests: 200000
Failed requests: 0
Total time: 242026 ms
Throughput: 826.3574987811227 requests/second
```

Result for load balanced instances:

Queue length: 0
Throughput: 889 requests/second
Number of producer channels: 120
Number of consumer channels: 120
Number of EC2 instance used for load balancer: 3




```
Successful requests: 200000
Failed requests: 0
Total time: 224947 ms
Throughput: 889.0983209378209 requests/second
```