**Assignment 2 – CS6650**

Yuchen Tian

1. URL for Github Repo: <https://github.com/tyuchn/cs6650-hw2-ytian>
2. Server Design
   1. Class Diagram

Graphical user interface, text, application, chat or text message, Teams

Description automatically generated

Graphical user interface, text, application

Description automatically generated

* 1. Mechanisms

The server consists of a tomcat servlet (including a producer), a RabbitMQ message queue and a consumer. For the producer, since we mainly focus on the skier servlet. There are three main classes in the skier servlet: *SkierServlet*, *ChannelFactory* and *Producer*. The SkierServlet is refactored from the previous assignment, it has an instance of GenericObjectPool (in the init() method), which is initiated by a ChannelFactory class. This pool will serve as a channel pool that shares pre-created channels with the producers. The ChannelFactory class is a factory for channel creation, it is implemented on top of the apache pool2 package. Producer class is the producer for sending message to RabbitMQ, it is instantiated by a channel pool, and in the produce() method, it will borrow a channel object from the channel pool, then publish the message to the message queue, at last, it will return the borrowed channel to the channel pool for future use. Recv is the consumer class, it was a 256-thread program, each thread creates a channel, get the message and writes to a concurrent hashmap. It is always implemented on the rabbitmq client package.

* 1. The life cycle of a message

When a post request was sent to the server, the skier servlet will first validate it, if the validation was passed, it will instantiate a producer object, and assign a channel from the channel pool to the producer, then the producer will publish the message to the message queue. On the consumer side, there is a 256-thread program, each thread will have a channel connecting with the message queue and consumer the message. At the same time, I use a concurrent map to store the record for each liftID.

1. Test Results
   1. 64 Threads
      1. Commands lines:

*-numThreads 64 -numSkiers 100000 -address 54.82.5.13:8080*

* + 1. RMQ management console:

Graphical user interface, application

Description automatically generated

* + 1. Result:

Text

Description automatically generated

* 1. 128 Threads
     1. Commands lines:

*-numThreads 128 -numSkiers 100000 -address 54.82.5.13:8080*

* + 1. RMQ management console:

Graphical user interface, text, application

Description automatically generated

* + 1. Result:

Text

Description automatically generated

* 1. 256 Threads
     1. Commands lines:

*-numThreads 256 -numSkiers 100000 -address 54.82.5.13:8080*

* + 1. RMQ management console:

Graphical user interface, text, application, email

Description automatically generated

* + 1. Result:

Text

Description automatically generated

* 1. 512 Threads
     1. Commands lines:

*-numThreads 512 -numSkiers 100000 -address 54.82.5.13:8080*

* + 1. RMQ management console:

Graphical user interface, text, application

Description automatically generated

* + 1. Result:

Text

Description automatically generated

1. Result Analysis:

Based on the result, in most of the time, the message queue size is close to zero. Which means our 256-thread consumer is capable to consumer all the message sent by 64, 128, 256 and 512 threads producer, there’s barely any message remains in the message queue. When a message was sent by the client, the producer will immediately send it to the queue, and there must be a consumer thread taking the message from its own channel at almost the same time. The testing result also indicates our design is efficient enough.