Assignment 1 Report

Github Repo:

https://github.com/CloudyZ524/6650-Distributed-System/tree/main/Assignment/Assignment1

Client Design

1. MultiThreadLiftRideClient .java (Entry Point):

- Initializes the event generation and starts the multi-threaded posting process.
- Sets up the ThreadPoolExecutor for managing multiple threads.
- Waits for all tasks to complete and prints out the results

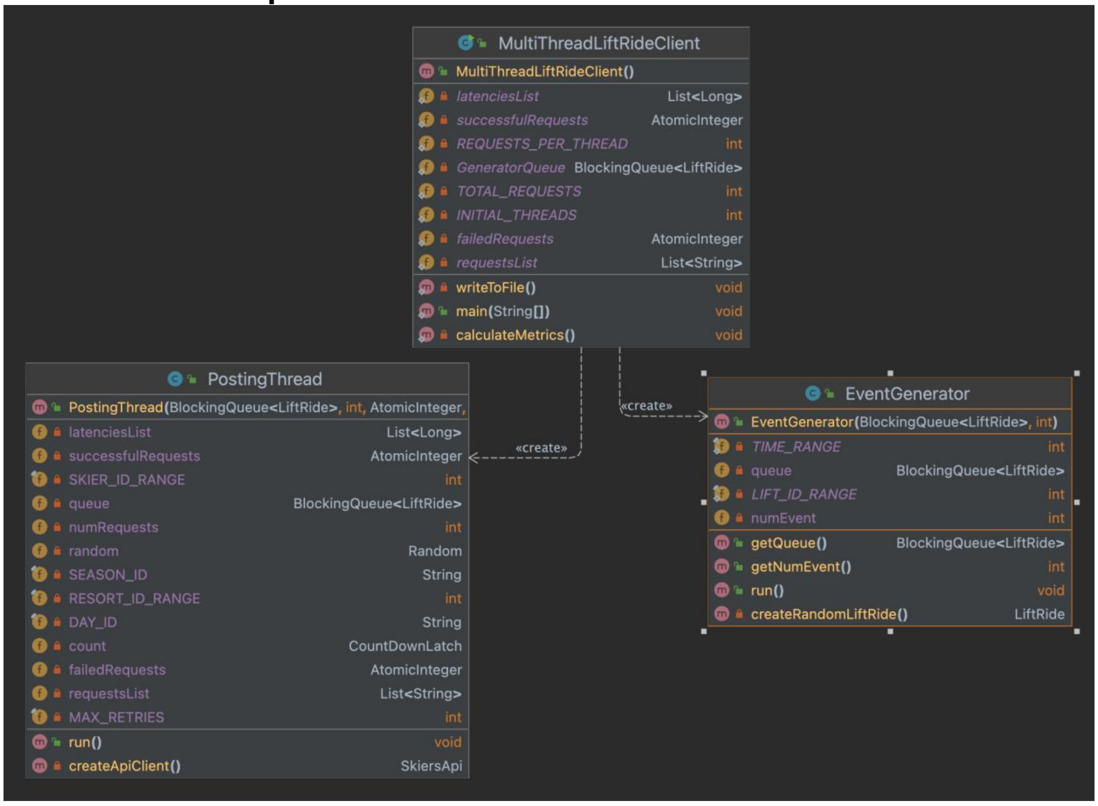
2. EventGenerator.java:

- Implements Runnable for concurrent execution.
- Generates a specified number of LiftRide events and adds them to a BlockingQueue.
- Uses random values to populate the LiftRide objects, simulating different lift rides.

3. PostingThread.java:

- Implements Runnable for concurrent execution.
- Takes LiftRide events from the BlockingQueue and posts them to a server using the SkiersApi.
- Handles retries for failed requests and collects latency metrics.

Class Relationships



MultiThreadLiftRideClient
MultiThreadLiftRideClient()
latenciesList List<Long>
successfulRequests AtomicInteger
REQUESTS_PER_THREAD int
GeneratorQueue BlockingQueue<LiftRide>
TOTAL_REQUESTS int
INITIAL_THREADS int
failedRequests AtomicInteger
requestsList List<String>
writeToFile() void
main(String[]) void
calculateMetrics() void
«create»
«create»
PostingThread
PostingThread(BlockingQueue<LiftRide>, int, AtomicInteger,
latenciesList List<Long>
successfulRequests AtomicInteger
SKIER_ID_RANGE int
queue BlockingQueue<LiftRide>
numRequests int
random Random
SEASON_ID String
RESORT_ID_RANGE int
DAY_ID String
count CountDownLatch
failedRequests AtomicInteger
requestsList List<String>
MAX_RETRIES int
run() void
createApiClient() SkiersApi
EventGenerator
EventGenerator(BlockingQueue<LiftRide>, int)
TIME_RANGE int
queue BlockingQueue<LiftRide>
LIFT_ID_RANGE int
numEvent int
getQueue() BlockingQueue<LiftRide>
getNumEvent() int
run() void
createRandomLiftRide() LiftRide

Threading and Synchronization

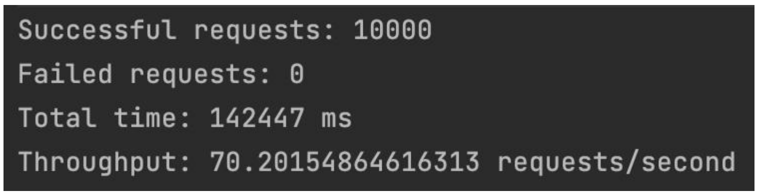
Multi-threaded Approach:
The client utilizes Java's threading capabilities for concurrent HTTP requests through a ThreadPoolExecutor. This executor manages threads that execute tasks within the PostingThread class, acting as individual SingleClients.

CountDownLatch:
A CountDownLatch is employed to synchronize the start and completion of thread tasks, ensuring that the main flow waits for all threads to finish before proceeding. This mechanism ensures accurate timing and performance measurement by preventing premature termination or aggregation of results.

AtomicInteger
The use of AtomicInteger for tracking successful and failed requests across threads provides a thread-safe mechanism to update shared counters without race conditions, essential for accurate metrics collection in a concurrent environment.

BlockingQueue
The application introduces a ConcurrentLinkedQueue to store LiftRide objects, which encapsulate detailed information about each HTTP request. This queue allows for thread-safe additions, enabling detailed performance analysis post-execution.

Little's Law throughput predictions



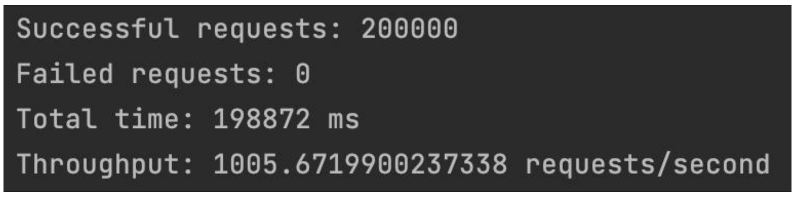
```
Successful requests: 10000
Failed requests: 0
Total time: 142447 ms
Throughput: 70.20154864616313 requests/second
```

Baseline response time: 142.447 sec / 10000 = 0.0142447 sec/request

According to Little’s Law, N = Throughput * Response Time.
If size of thread pool is 15, then N = 15.
Estimated throughput = N / Response Time
= 15 / 0.0142447
= 1053.02323 request/sec

The estimation is similar to the actual result shown below.

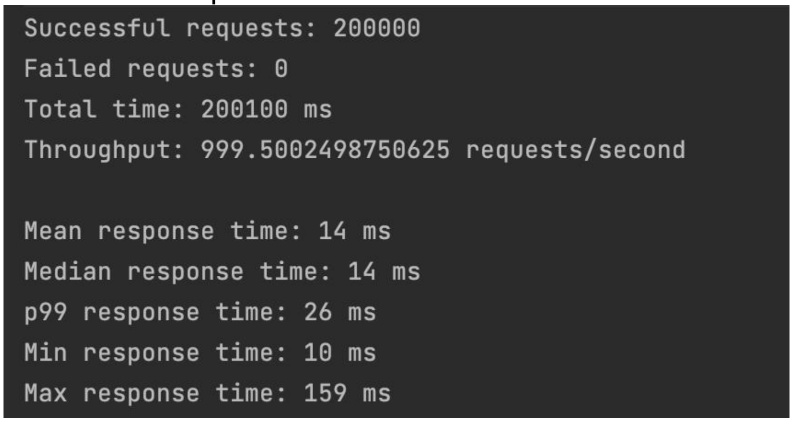
Client - Part 1
Size of thread pool : 15



```
Successful requests: 200000
Failed requests: 0
Total time: 198872 ms
Throughput: 1005.6719900237338 requests/second
```

Client - Part 2

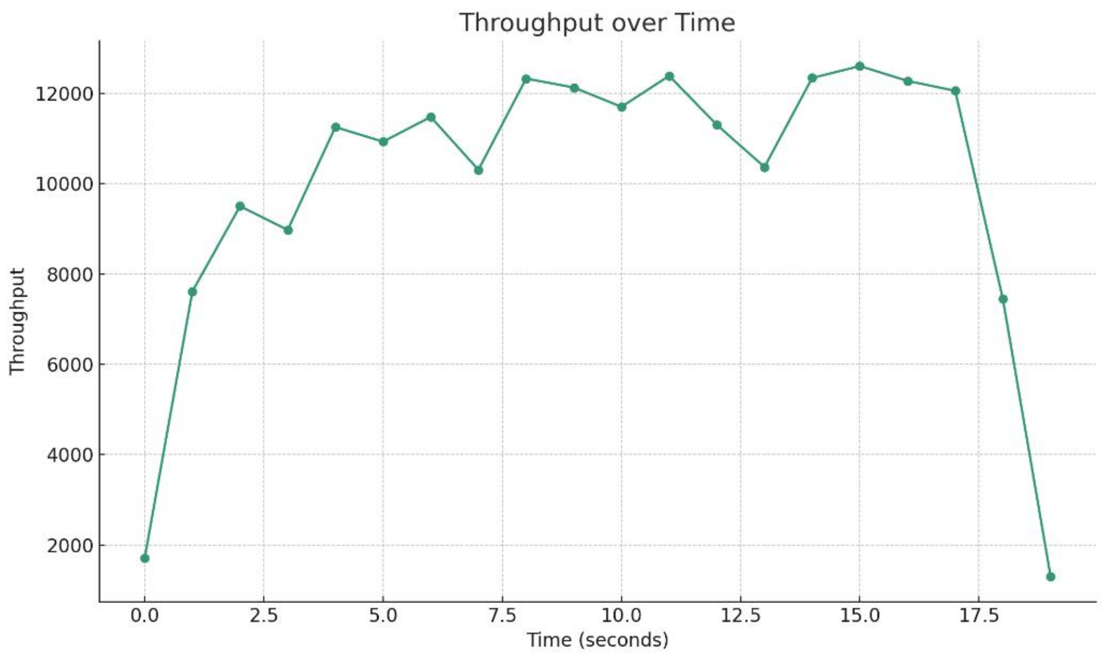
Size of thread pool : 15



```
Successful requests: 200000
Failed requests: 0
Total time: 200100 ms
Throughput: 999.5002498750625 requests/second

Mean response time: 14 ms
Median response time: 14 ms
p99 response time: 26 ms
Min response time: 10 ms
Max response time: 159 ms
```

Plot of Throughput


Throughput over Time
Throughput
Time (seconds)