Digitalizing the Ski Sport Industry with a Cloud-Based Distributed System

Distributed Software Systems (COEN 6731) Project Report

Limin Xiong
Student ID: 40240127
Department of Electrical and Computer Engineering
Concordia University
Montreal, Canada
xiongx659@gmail.com

Sahibmeet Singh
Student ID: 40211319
Department of Electrical and Computer Engineering
Concordia University
Montreal, Canada
singh.concordia@gmail.com

Vishruth Khatri
Student ID: 40241455
Department of Electrical and Computer Engineering
Concordia University
Montreal, Canada
itsvishruthkhatri@hotmail.com

Pranav Jha
Student ID: 40081750
Department of Electrical and Computer Engineering
Concordia University
Montreal, Canada
jha_k.pranav@live.com

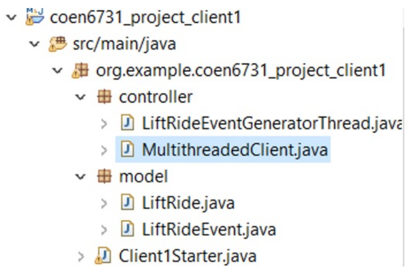
***Abstract*—This project involves developing a distributed system to digitize the sports industry by collecting lift-ride data from geographically distributed resorts. The data collected is used for data analysis, such as determining the most heavily used lifts and the skiers who ride the most lifts. The project requires implementing a server-side API and building a multithreaded Java client to send POST requests to the server. The client generates 10K POST requests and handles errors by retrying requests up to 5 times before counting them as failed requests. The project also involves profiling performance by calculating the mean and median response time, throughput, and p99 response time.**

***Keywords*—Client-server architecture, Multi-threaded Java client, HTTP protocol, Error handling, Concurrency.**

I. Client Design

The URL for our git repository is https://github.com/BBeerBear/PROJECT.git

A. Classes, Packages, Relationships

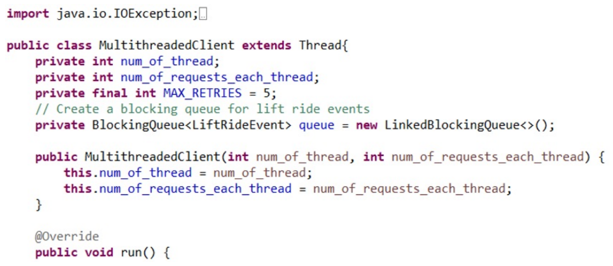


```
coen6731_project_client1
  src/main/java
    org.example.coen6731_project_client1
      controller
        LiftRideEventGeneratorThread.java
        MultithreadedClient.java
      model
        LiftRide.java
        LiftRideEvent.java
      Client1Starter.java
```

Figure 1. Structure of Client 1

1) The controller package, as shown in Fig. 1, includes a thread called LiftRideEventGeneratorThread.java, which can be executed to generate lift-ride event data. Additionally, the MultithreadedClient.java thread is designed to run the HttpClient in multiple threads to send multiple post requests to the server API.
2) In the model package, the LiftRide.java and LiftRideEvent.java classes are Plain Old Java Objects (POJOs).
3) The Client1Starter.java file is used to initiate the execution of the MultithreadedClient.java we created.

B. Multi-Threaded Client



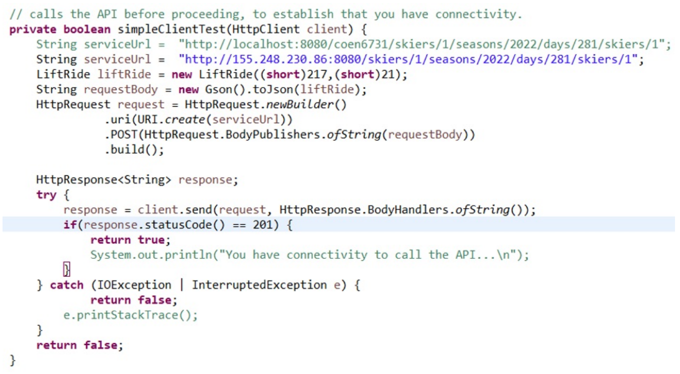
```
import java.io.IOException;

public class MultithreadedClient extends Thread{
    private int num_of_thread;
    private int num_of_requests_each_thread;
    private final int MAX_RETRIES = 5;
    // Create a blocking queue for lift ride events
    private BlockingQueue<LiftRideEvent> queue = new LinkedBlockingQueue<>();

    public MultithreadedClient(int num_of_thread, int num_of_requests_each_thread) {
        this.num_of_thread = num_of_thread;
        this.num_of_requests_each_thread = num_of_requests_each_thread;
    }

    @Override
    public void run() {
```

Figure 2. Creating a Multi-Threaded Client

To simulate the multithreaded client, a new thread has been created with the capability to configure the number of threads and requests per thread as shown in Fig. 2.

C. Testing the Connection between Client and Server

1) Method to Test the Client-Server Connection: Before executing the client's multiple threads, a method has been written to test if the client is connected to the server as shown in Fig. 3.

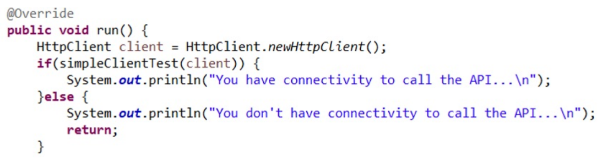


```
// calls the API before proceeding, to establish that you have connectivity.
private boolean simpleClientTest(HttpClient client) {
    String serviceUrl =  "http://localhost:8080/coen6731/skiers/1/seasons/2022/days/281/skiers/1";
    String serviceUrl =  "http://155.248.230.86:8080/skiers/1/seasons/2022/days/281/skiers/1";
    LiftRide liftRide = new LiftRide((short)217,(short)21);
    String requestBody = new Gson().toJson(liftRide);
    HttpRequest request = HttpRequest.newBuilder()
              .uri(URI.create(serviceUrl))
              .POST(HttpRequest.BodyPublishers.ofString(requestBody))
              .build();

    HttpResponse<String> response;
    try {
        response = client.send(request, HttpResponse.BodyHandlers.ofString());
        if(response.statusCode() == 201) {
            return true;
            System.out.println("You have connectivity to call the API...\n");
        }
    } catch (IOException | InterruptedException e) {
            return false;
        e.printStackTrace();
    }
    return false;
}
```

Figure 3. Method for Testing the Client-Server Connection

Cloud URL: http://155.248.230.86:8080/skiers/1/seasons/2022/days/366/skiers/1

2) Stopping the Client Thread: Fig. 4 illustrates the method where the client thread stops when there is no connectivity.

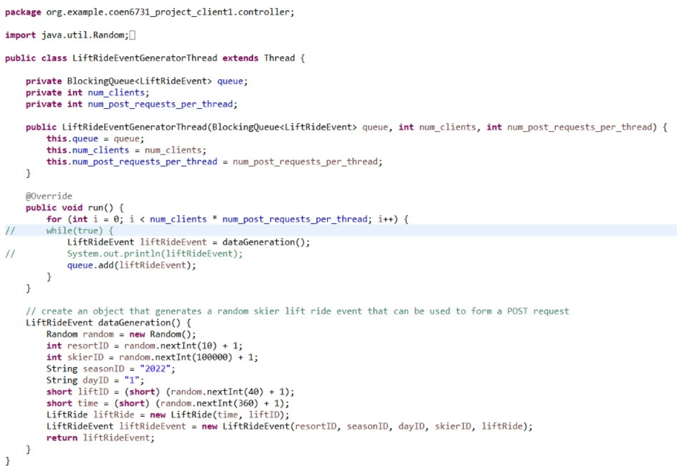


```
@Override
public void run() {
    HttpClient client = HttpClient.newHttpClient();
    if(simpleClientTest(client)) {
        System.out.println("You have connectivity to call the API...\n");
    }else {
        System.out.println("You don't have connectivity to call the API...\n");
        return;
    }
```

Figure 4. Method to Stop the Client Thread When No Connectivity Is Found

D. Creating a Data Generation Thread to Store Lift Ride Event Data in Thread-Safe Memory

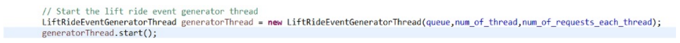
To maintain thread-safe memory, a data generation thread has been created to generate lift ride event data and store it in a BlockingQueue as shown in Fig. 5.



```
package org.example.coen6731_project_client1.controller;

import java.util.Random;

public class LiftRideEventGeneratorThread extends Thread {

    private BlockingQueue<LiftRideEvent> queue;
    private int num_clients;
    private int num_post_requests_per_thread;

    public LiftRideEventGeneratorThread(BlockingQueue<LiftRideEvent> queue, int num_clients, int num_post_requests_per_thread) {
        this.queue = queue;
        this.num_clients = num_clients;
        this.num_post_requests_per_thread = num_post_requests_per_thread;
    }

    @Override
    public void run() {
        for (int i = 0; i < num_clients * num_post_requests_per_thread; i++) {
//      while(true) {
            LiftRideEvent liftRideEvent = dataGeneration();
//          System.out.println(liftRideEvent);
            queue.add(liftRideEvent);
        }
    }

    // create an object that generates a random skier lift ride event that can be used to form a POST request
    LiftRideEvent dataGeneration() {
        Random random = new Random();
        int resortID = random.nextInt(10) + 1;
        int skierID = random.nextInt(100000) + 1;
        String seasonID = "2022";
        String dayID = "1";
        short liftID = (short) (random.nextInt(40) + 1);
        short time = (short) (random.nextInt(360) + 1);
        LiftRide liftRide = new LiftRide(time, liftID);
        LiftRideEvent liftRideEvent = new LiftRideEvent(resortID, seasonID, dayID, skierID, liftRide);
        return liftRideEvent;
    }
}
```

Figure 5. Data Generation Thread for Storing the Lift Ride Event Data in Thread-Safe Memory

To start the LiftRideEventGeneratorThread as shown in Fig. 6, the run() method of the multithreaded client is utilized.

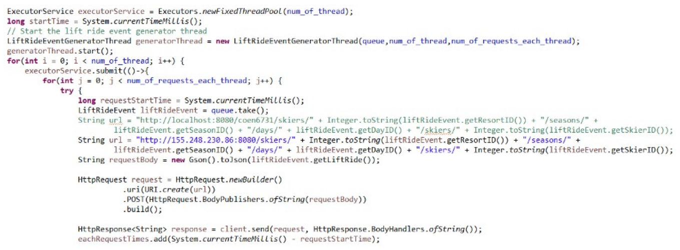


```
// Start the lift ride event generator thread
LiftRideEventGeneratorThread generatorThread = new LiftRideEventGeneratorThread(queue,num_of_thread,num_of_requests_each_thread);
generatorThread.start();
```

Figure 6.

E. Multithreading with ExecutorService and Request Sending Mechanism

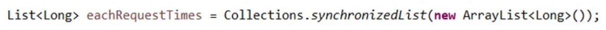
Fig. 7 illustrates the utilization of ExecutorService's submit() method for running multiple threads, where each thread sends multiple requests. Each request takes a lift ride event object from a blocking queue that stores the lift ride events generated by the LiftRideEventGeneratorThread and sends it to the server API.



```
ExecutorService executorService = Executors.newFixedThreadPool(num_of_thread);
long startTime = System.currentTimeMillis();
// Start the lift ride event generator thread
LiftRideEventGeneratorThread generatorThread = new LiftRideEventGeneratorThread(queue,num_of_thread,num_of_requests_each_thread);
generatorThread.start();
for(int i = 0; i < num_of_thread; i++) {
    executorService.submit(()->{
        for(int j = 0; j < num_of_requests_each_thread; j++) {
            try {
                long requestStartTime = System.currentTimeMillis();
                LiftRideEvent liftRideEvent = queue.take();
                String url = "http://localhost:8080/coen6731/skiers/" + Integer.toString(liftRideEvent.getResortID()) + "/seasons/" +
                        liftRideEvent.getSeasonID() + "/days/" + liftRideEvent.getDayID() + "/skiers/" + Integer.toString(liftRideEvent.getSkierID());
                String url = "http://155.248.230.86:8080/skiers/" + Integer.toString(liftRideEvent.getResortID()) + "/seasons/" +
                        liftRideEvent.getSeasonID() + "/days/" + liftRideEvent.getDayID() + "/skiers/" + Integer.toString(liftRideEvent.getSkierID());
                String requestBody = new Gson().toJson(liftRideEvent.getLiftRide());

                HttpRequest request = HttpRequest.newBuilder()
                            .uri(URI.create(url))
                            .POST(HttpRequest.BodyPublishers.ofString(requestBody))
                            .build();

                HttpResponse<String> response = client.send(request, HttpResponse.BodyHandlers.ofString());
                eachRequestTimes.add(System.currentTimeMillis() - requestStartTime);
```

Figure 7. Utilizing ExecutorService and BlockingQueue to run multiple threads and send lift ride event data to server API

Before executing multiple threads, record the start time and save each response time in a thread-safe list as shown in Fig. 8.

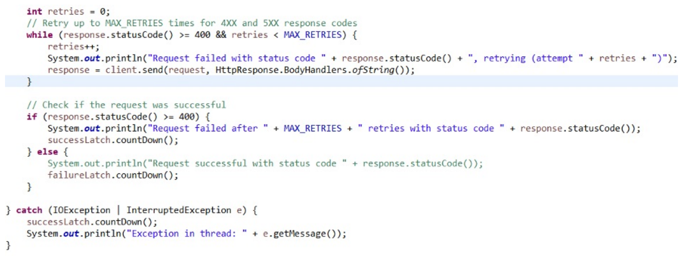


```
List<Long> eachRequestTimes = Collections.synchronizedList(new ArrayList<Long>());
```

Figure 8. Recording Start Time and Response Times in Thread-Safe List during Multi-Threaded Execution

F. Handling Errors

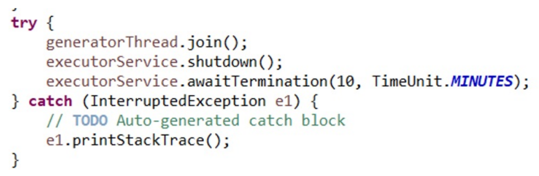
When the client receives a 5XX response code (Web server error) or a 4XX response code (from the servlet), it retries the request up to 5 times before considering it a failed request as shown below in Fig. 9.



```
    int retries = 0;
    // Retry up to MAX_RETRIES times for 4XX and 5XX response codes
    while (response.statusCode() >= 400 && retries < MAX_RETRIES) {
        retries++;
        System.out.println("Request failed with status code " + response.statusCode() + ", retrying (attempt " + retries + ")");
        response = client.send(request, HttpResponse.BodyHandlers.ofString());
    }

    // Check if the request was successful
    if (response.statusCode() >= 400) {
        System.out.println("Request failed after " + MAX_RETRIES + " retries with status code " + response.statusCode());
        successLatch.countDown();
    } else {
        System.out.println("Request successful with status code " + response.statusCode());
        failureLatch.countDown();
    }

} catch (IOException | InterruptedException e) {
    successLatch.countDown();
    System.out.println("Exception in thread: " + e.getMessage());
}
```

Figure 9. Retrying Failed Requests

All threads terminate cleanly when all 10k requests have been successfully sent as shown below.

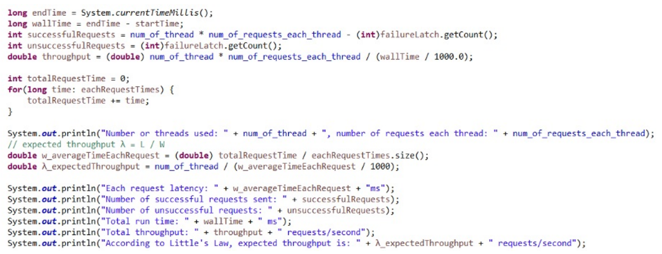


```
try {
    generatorThread.join();
    executorService.shutdown();
    executorService.awaitTermination(10, TimeUnit.MINUTES);
} catch (InterruptedException e1) {
    // TODO Auto-generated catch block
    e1.printStackTrace();
}
```

Figure 10. Clean termination of all threads upon successful completion of 10k requests

The program outputs the following upon completion as shown in Fig. 11:

(a) The number of successful requests sent
(b) The number of unsuccessful requests, which should be 0
(c) The total wall time for all phases to complete, calculated by taking a timestamp before starting any threads and another after all threads are complete
(d) The total throughput in requests per second, which is calculated as the total number of requests divided by the wall time
(e) The predicted throughput according to Little's Law

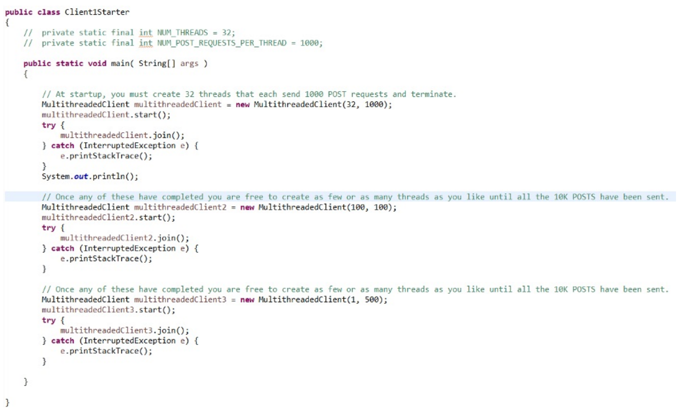


```
long endTime = System.currentTimeMillis();
long wallTime = endTime - startTime;
int successfulRequests = num_of_thread * num_of_requests_each_thread - (int)failureLatch.getCount();
int unsuccessfulRequests = (int)failureLatch.getCount();
double throughput = (double) num_of_thread * num_of_requests_each_thread / (wallTime / 1000.0);

int totalRequestTime = 0;
for(long time: eachRequestTimes) {
    totalRequestTime += time;
}

System.out.println("Number or threads used: " + num_of_thread + ", number of requests each thread: " + num_of_requests_each_thread);
// expected throughput λ = L / W
double w_averageTimeEachRequest = (double) totalRequestTime / eachRequestTimes.size();
double λ_expectedThroughput = num_of_thread / (w_averageTimeEachRequest / 1000);

System.out.println("Each request latency: " + w_averageTimeEachRequest + "ms");
System.out.println("Number of successful requests sent: " + successfulRequests);
System.out.println("Number of unsuccessful requests: " + unsuccessfulRequests);
System.out.println("Total run time: " + wallTime + " ms");
System.out.println("Total throughput: " + throughput + " requests/second");
System.out.println("According to Little's Law, expected throughput is: " + λ_expectedThroughput + " requests/second");
```

Figure 11. Program output upon completion

G. Running the MultithreadedClient Thread with Different Configurations

In the main entry file Client1Starter.java, we create and run the MultithreadedClient thread with various configurations. These include different numbers of threads and the number of requests per thread shown below.

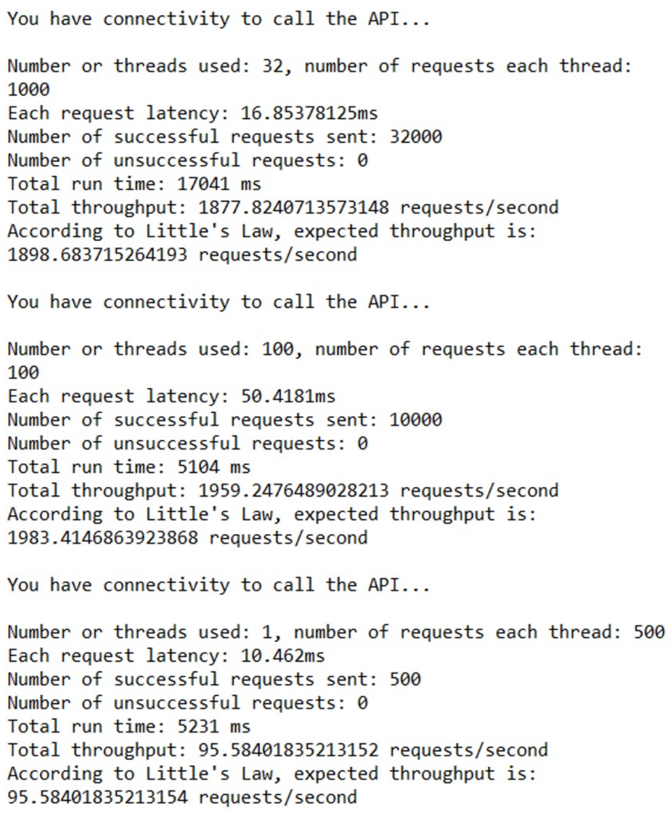


```
public class Client1Starter
{
    //  private static final int NUM_THREADS = 32;
    //  private static final int NUM_POST_REQUESTS_PER_THREAD = 1000;

    public static void main( String[] args )
    {

        // At startup, you must create 32 threads that each send 1000 POST requests and terminate.
        MultithreadedClient multithreadedClient = new MultithreadedClient(32, 1000);
        multithreadedClient.start();
        try {
            multithreadedClient.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println();

        // Once any of these have completed you are free to create as few or as many threads as you like until all the 10K POSTS have been sent.
        MultithreadedClient multithreadedClient2 = new MultithreadedClient(100, 100);
        multithreadedClient2.start();
        try {
            multithreadedClient2.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Once any of these have completed you are free to create as few or as many threads as you like until all the 10K POSTS have been sent.
        MultithreadedClient multithreadedClient3 = new MultithreadedClient(1, 500);
        multithreadedClient3.start();
        try {
            multithreadedClient3.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

    }

}
```

Figure 12. Testing the MultithreadedClient with Different Configurations of Threads and Requests per Thread

II. Building Client

The output window for the client design is provided below in Fig. 13.



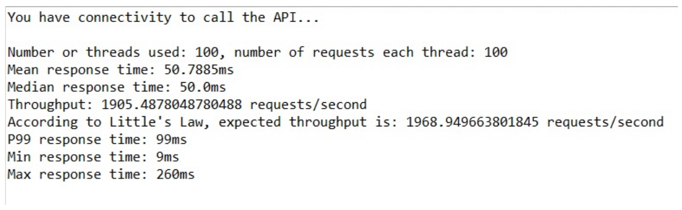
```
You have connectivity to call the API...

Number or threads used: 32, number of requests each thread:
1000
Each request latency: 16.85378125ms
Number of successful requests sent: 32000
Number of unsuccessful requests: 0
Total run time: 17041 ms
Total throughput: 1877.8240713573148 requests/second
According to Little's Law, expected throughput is:
1898.683715264193 requests/second

You have connectivity to call the API...

Number or threads used: 100, number of requests each thread:
100
Each request latency: 50.4181ms
Number of successful requests sent: 10000
Number of unsuccessful requests: 0
Total run time: 5104 ms
Total throughput: 1959.2476489028213 requests/second
According to Little's Law, expected throughput is:
1983.4146863923868 requests/second

You have connectivity to call the API...

Number or threads used: 1, number of requests each thread: 500
Each request latency: 10.462ms
Number of successful requests sent: 500
Number of unsuccessful requests: 0
Total run time: 5231 ms
Total throughput: 95.58401835213152 requests/second
According to Little's Law, expected throughput is:
95.58401835213154 requests/second
```

Figure 13. The console output

Based on the output displayed above, the actual throughput is observed to be close to the predicted throughput, which indicates that the client has executed successfully.

III. Profiling Performance

The performance profiling results are shown below in Fig. 14.



```
You have connectivity to call the API...

Number or threads used: 100, number of requests each thread: 100
Mean response time: 50.7885ms
Median response time: 50.0ms
Throughput: 1905.4878048780488 requests/second
According to Little's Law, expected throughput is: 1968.949663801845 requests/second
P99 response time: 99ms
Min response time: 9ms
Max response time: 260ms
```

Figure 14. Profiling performance

As can be seen from the figure above, the profiling performance indicates that the throughput is within 5% of the total throughput of client part 1 as shown in Fig. 13.