CSCI-201: Principles of Software Development

Fall 2021

PA3: SalEats v2.0

Concepts Covered:

- Networking
- Multi-Threading
- Concurrency Issues
- API Querying

Introduction:

After a stressful and sweaty pitch, the Council of SAL is satisfied with your order-scheduling prototype demo application and has unanimously decided to give you the green light to fully implement your application. This involves pulling real data from Yelp, issuing orders from SalEats HQ to drivers in real time over the network, and chaining orders for drivers, much like some other food delivery application whose name shall not be mentioned.

Restaurant Data:

In this assignment, you will first read in a JSON file (PA3.txt) containing various information regarding the restaurants and their menus. The JSON format is generally the same as the one we used in the previous assignment, so feel free to reuse the parser you've built. Here's a sample entry

```
{ "name": "Momota Ramen House",
  "address": "3019 S Figueroa St",
  "latitude": 34.024840,
  "longitude": -118.278060,
  "drivers": 2,
  "menu": [
     "black garlic ramen",
     "spicy miso ramen"
   ]
}
```

Yelp API:

In addition to reading in data from a hard-coded text file, you will use the Yelp API to retrieve restaurant data. In other words, the restaurant data (i.e. address, coordinates, etc.) should now be pulled from this API, and you will only have to read from the text file for the schedule of orders. You can learn more about the Yelp API here: https://www.yelp.com/developers/documentation/v3/get_started. The data will be returned in a JSON format. Please keep in mind that there is a limit of 5,000 API calls per day, so *do not wait to start your assignment until the last day*. You will also need to generate an API Key, so start the assignment early to allow enough time for testing.

What To Do:

In this assignment, you will create two different programs - a server and a client. You will implement a networked delivery system where clients (which represent drivers) receive orders from the server (which represents the headquarters), and the drivers will deliver the orders to the appropriate restaurants. If it seems counterintuitive, you can think of it as a backwards food delivery system, where food is delivered to restaurants. Alternatively, your drivers are delivering ingredients for the orders to the restaurants.

There will be some concurrency issues since drivers will have to wait on other drivers before starting the delivery. There will be many ways to design out the program, so it would be wise to spend some time designing the program before you begin coding.

Aside from the restaurant data, you will need to read in a scheduling information. Below is a layout of that text file, which contains information for the orders:

```
0, Momota Ramen House, black garlic ramen
0, Momota Ramen House, spicy miso ramen
0, Slurpin' Ramen Bar, veggie ramen
3, Daikokuya Little Tokyo, daikoku ramen
3, Daikokuya Little Tokyo, daikoku ramen
9, Daikokuya Little Tokyo, daikoku ramen
9, Tengoku Ramen Bar, chicken ramen
10, Tengoku Ramen Bar, tonkotsu ramen
```

The first parameter indicates when the order is ready. The second parameter indicates the location that the order is coming from. The third parameter indicates which food is being delivered.

Server Functionality:

When your server first runs, prompt the user for the name of the schedule file and for their coordinates. This will be the "home" location that drivers will leave from and return to. Then, the server should ask the user how many drivers will be dispatched. For example, if the user enters "3," that means the server will not send out any orders until 3 clients (aka drivers) have connected to the server. Afterwards, the server should begin listening on port 3456 for client connections. Every time a client connects to the server, verify that the connection was made by printing an output from the server, similar to the sample output below.

Client Functionality:

The client will begin by welcoming the user to the program and prompting the user for the server hostname and port. If a valid connection is made, the program should let the user know how many more drivers are needed before the orders are delivered. For example, if there is currently only 1 connection, the client should print a message saying that 2 more drivers are needed before the orders can be delivered. This number should be inclusive, such that the total number of connected drivers includes the current driver as well. Once the orders are dispatched, the client will be responsible for delivering the order to the appropriate restaurants. The client should also be handling the returned data from the API calls to determine the distances of each restaurant from the current position.

Drivers can deliver more than one order at a time. Additionally, drivers can deliver orders to multiple restaurants at a time. Drivers will deliver orders with the shortest distance first. Ou nay assume that it takes each driver exactly one second to travel one mile, which is the equivalent of a *USAF X-51A Waverider* jet traveling at Mach 6. You need to calculate how many seconds it will take for a driver to deliver food to the next restaurant. So, once an order is delivered, the driver will need to recalibrate the distance from their current location to the next location and pick the shortest distance again. This is continued until there are no more orders, in which case the driver will return to the "home" location that was provided when the server first executed.

A driver should handle as many orders as possible at the moment of dispatch. This means that if there are 5 orders that all have the same timestamp, one driver should be responsible for all 5 orders. Similarly, if only 1 order is ready at a timestamp, the driver should only deliver 1 order. As long as there are available drivers, orders should be delivered promptly. If there are no available drivers, the order will remain in the queue until a driver returns from their delivery. Once a driver returns, the driver should pick up all of the queued orders (with respect to the current time). Your program should print upon completing the delivery to a restaurant, and sleep according to the

calculated distance between points. As such, please follow the sample execution below for the proper output.

Sample Output:

Server	Client 1	Client 2
What is the name of the schedule file? **missing.txt** That file does not exist. What is the name of the schedule file? **badformat.txt** That file is not properly formatted. What is the name of the schedule file? **schedule.txt** What is your latitude? **34.02116** What is your longitude? **-118.287132** How many drivers will be in service today? **2** Listening on port 3456. Waiting for drivers…	Welcome to SalEats v2.0!	

	Enter the server hostname: **localhost** Enter the server port: **3456**	
Connection from 127.0.0.1 Waiting for 1 more driver(s)...	1 more driver is needed before the service can begin. Waiting...	
		Welcome to SalEats v2.0! Enter the server hostname: **localhost** Enter the server port: **3456**
Connection from 127.0.0.1 Starting service.	All drivers have arrived! Starting service. **[18:06:000]** Starting delivery of black garlic ramen to Momota Ramen House. **[18:06:000]** Starting delivery of spicy miso ramen to Momota Ramen House. **[18:06:000]** Starting delivery of veggie ramen to Slurpin' Ramen Bar. **[18:06.600]** Finished delivery of black garlic ramen to Momota Ramen House. **[18:06.600]** Finished delivery of spicy miso ramen to Momota Ramen House. **[18:06.650]** Continuing delivery to Slurpin' Ramen	All drivers have arrived! Starting service.

	Bar. **[18:09:440]** Finished delivery of veggie ramen to Slurpin' Ramen Bar. **[18:09:450]** Finished all deliveries, returning back to HQ. **[18:12:200]** Returned to HQ. **[18:12:200]** Starting delivery of tonkotsu ramen to Daikokuya **[18:15:000]** Starting delivery of daikoku ramen to Daikokuya Little Tokyo. **[18:15:000]** Starting delivery of chicken ramen to Tengoku Ramen Bar.	**[18:09:000]** Starting delivery of daikoku ramen to Daikokuya Little Tokyo. **[18:09:000]** Starting delivery of daikoku ramen to Daikokuya Little Tokyo. **[18:12:350]** Finished delivery of daikoku ramen to Daikokuya Little Tokyo. **[18:12:350]** Finished delivery of daikoku ramen to Daikokuya Little Tokyo. **[18:12:360]** Finished all deliveries, returning back to HQ. **[18:15:710]** Returned to HQ. **[18:16:000]** Starting delivery of tonkotsu

		ramen to Tengoku Ramen Bar.
	[18:18:250] Finished delivery of chicken ramen to Tengoku Ramen Bar. **[18:18:300]** Continuing delivery to Daikokuya Little Tokyo.	
		[18:19:350] Finished delivery of tonkotsu ramen to Tengoku Ramen Bar. **[18:19:360]** Finished all deliveries, returning back to HQ.
	[18:22:380] Finished delivery of daikoku ramen to Daikokuya Little Tokyo. **[18:22:390]** Finished all deliveries, returning back to HQ.	
		[18:22:610] Returned to HQ.
	[18:25:740] Returned to HQ.	
All orders completed!	**[18:25:750]** All orders completed!	**[18:25:750]** All orders completed!

Starter Code:

Since networking is a challenging concept for some and requires quite a bit of time to digest, we provide some free code to ease the pressure:

- Code for API requests is provided. However, you will still need to read Yelp's documentation and come up with the correct query parameters and of course, your own API key.
- Code for timestamps is included.
- Note that the code provided is not an export and therefore should not be imported. Copy and paste the files into your directories of choice, modify the import and package lines as you see fit, then manually include the jar libraries.

Grading Criteria:

The way you go about implementing the solution is not specifically graded, but the output must match exactly what you see in the execution above.

Networking (20 points)
4 pts. The first client can connect to the server
6 pts. Only the number of clients specified can connect to the server.
10 pts. Server output is correct

Data I/O (20 points)
6 pts. The schedule file is read appropriately
4 pts. Data is parsed from the Yelp API
10 pts. Client output is correct

Program Execution (60 points)
20 pts. The order of deliveries is correct
20 pts. The timing of deliveries is correct
20 pts. Orders are delivered as expected with no exceptions, crashing, deadlock, starvation, or freezing.