The Cabify Ridesharing Challenge: A Quick Overview

Carlos Perez-Guerra

2/1/2020

The goal of this short text is to explain some of my design ideas.

Code structure and organization

The design achieves a clean architecture with separation of concerns through a classic layered approach (illustrated in Fig. 1). The layers are as follows:

- The `main` layer is the entry point of the application and wraps the other layers.
- The `api` layer contains the handlers for the api endpoints. Because both the api and business logic in this case are very simple (just a few lines per handler), I have chosen to keep them coupled under the same package. A larger API with more endpoints or more sophisticated business logic would justify splitting this layer into two (api & business layers).
- The `models` layer contains data models and their methods.
- The `data` layer provides an interface which is consumed by the `api` layer. It is best practice to segregate data access operations and business logic for several well-known reasons (data operations are bug-prone and benefit from extensive unit testing, business logic changes more often than data access procedures, etc.)
- The `backing` layer simulates a queue-based backing system such as Redis or RabbitMQ. An important part of designing cloud-native applications is that processes should be stateless (e.g. Factor 6 of the 12-Factor App). The Readme file is clear on the preference for this project to be self-contained, so I am just rolling my own queue-based backing service as its own layer.

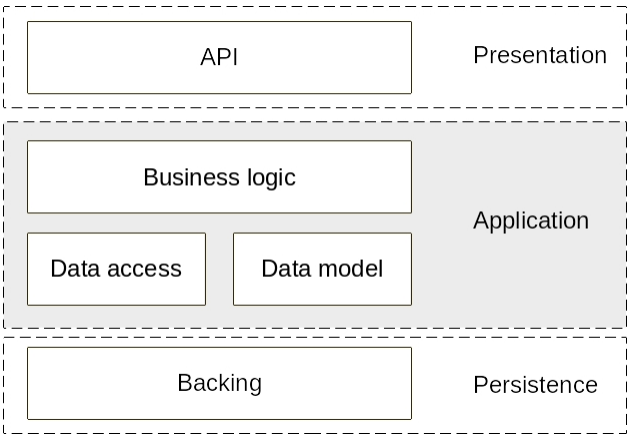

Figure 1: A diagram of the architecture of the ridesharing API

Time complexity of the match procedure

The backing layer is a very efficient queue that can easily handle Cabify-scale levels of users. It contains a single data structure, which I have named `HashQueue`, owing to the fact that it is both a queue and a

hashmap, which is designed to index objects with a size and an id. It wraps both an array of queues (`BySize`) and a hashmap (`ById`) (see Fig. 2 for an illustration). The following operations are supported in constant time:

- Create / Read / Update / Delete an object.
- Change the size of an object.
- Find the oldest or newest element of a certain size.
- Find the oldest or newest object with size within a certain interval.

This structure is then specialized into a `CarQueue` and a `JourneyQueue` in the `data` layer. The `Match` procedure can then match journeys to cars in $O(m)$ time, where m is the number of feasible journeys (according to the instructions in the Readme, a journey is feasible is it can both fit in an available car and there isn't an older journey which could fit in a car).

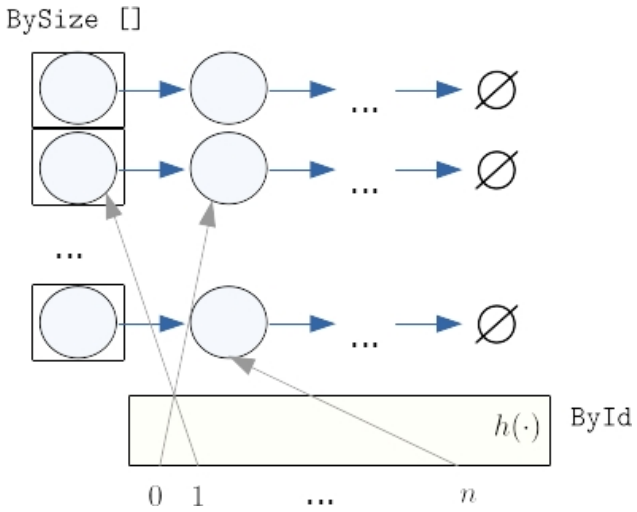

Figure 2: An illustration of the HashQueue data structure. The BySize array contains the head of several queues, allowing efficient retrieval by size. A hashmap then maps ids to pointers allowing efficient CRUD operations by ID.

Remark: There are better algorithms for this problem which would increase occupancy rates (online knapsack, or iterated local search would probably give better results) while maintaining the constraint that older journeys should be assigned first.

Scalability

The match procedure should run concurrently, e.g. in its own thread. This would increase latency of the api during peak periods. I have not implemented concurrent operation for this exercise, but it is quite trivial to do in Go.