Design Uber

Practice On: 21 FEB 2025

Time Taken: 120 Mins

Open Qs for the interviewer

1. What all operation can be performed by the user?
2. What all operation can be performed by the cab driver?
3. What all operation/option we show on the application?
4. Will their be choice for diff types of vehicles for booking?
5. Feature like adv booking, credit etc needs to be in scoop?

Requirements

Functional Requirements

1. User is able to request for a can from a source A to destination B
2. User can see the quotation price and can decide to accept and request the ride or cancel
3. User will be shown the live location of the cab, as it moves from current location to source A to destination B
4. Drive will be able to get request for rides, driver can choose to accept or reject

Non Functional Requirement

1. Consistency - single cab driver is mapped / booked for single cab requester
2. High availability for cab requests
3. Scalable for users and geographically
4. Near time live location for the driver to user

Entity

1. User
2. Driver
3. Ride
4. Location

User
1. UserID (PK) 2. Name 3. Contact No 4. Email ID

Driver
1. DriverID (PK) 2. Name 3. Contact No 4. Email ID 5. isOnline? 6. CabNo

Ride
1. BookingID (PK) 2. UserID (FK) 3. DriverID (PK) 4. Source 5. Destination 6. Status (ENUM) 7. Rate 8. Time

Location
1. DriverID 2. Long 3. Lati 4. Geohash

APIs

1. User requesting for the ride from location A to location B

Request:
```
PUT /ride/request
body {
	source: {long:"", lati:""},
	destination: {long:"", lati:""},
	time: # optional - for adv booking
}
```

Response:
```
{
	{ride_type: "premium car", wait_time: "", estimate_price: ""},
	{ride_type: "economic car", wait_time: "", estimate_price: ""},
	{ride_type: "auto", wait_time: "", estimate_price: ""},
	estimate_time: "" # point A -> B
	ride_request_id: ""
}
```

2. User choose and finalise an option for his ride

Request:
```
PATCH /ride/request/book
body {
	ride_request_id: ""
}
```

Response:
```
{
	message: "please wait, finding your ride"
}
```

3. Send & show ride request to nearby drivers

Request:
```
PUT /ride/offer
body {
	source: {long:"", lati:""},
	destination: {long:"", lati:""},
	price: "",
	ride_request_id: ""
}
```

At this point driver will see option an his app to accept / reject - lets say he accepts the ride

```
PATCH ride/offer
body {
	"accepted",
	ride_request_id: ""
}
```

In response application will send the source location A to the driver
NOTE FOR SELF: need to see what API call will happen her PUT or PATCH

```
PATCH ride/offer
body {
	ride_request_id: "",
	source: {long:"", lati:""},
}
```

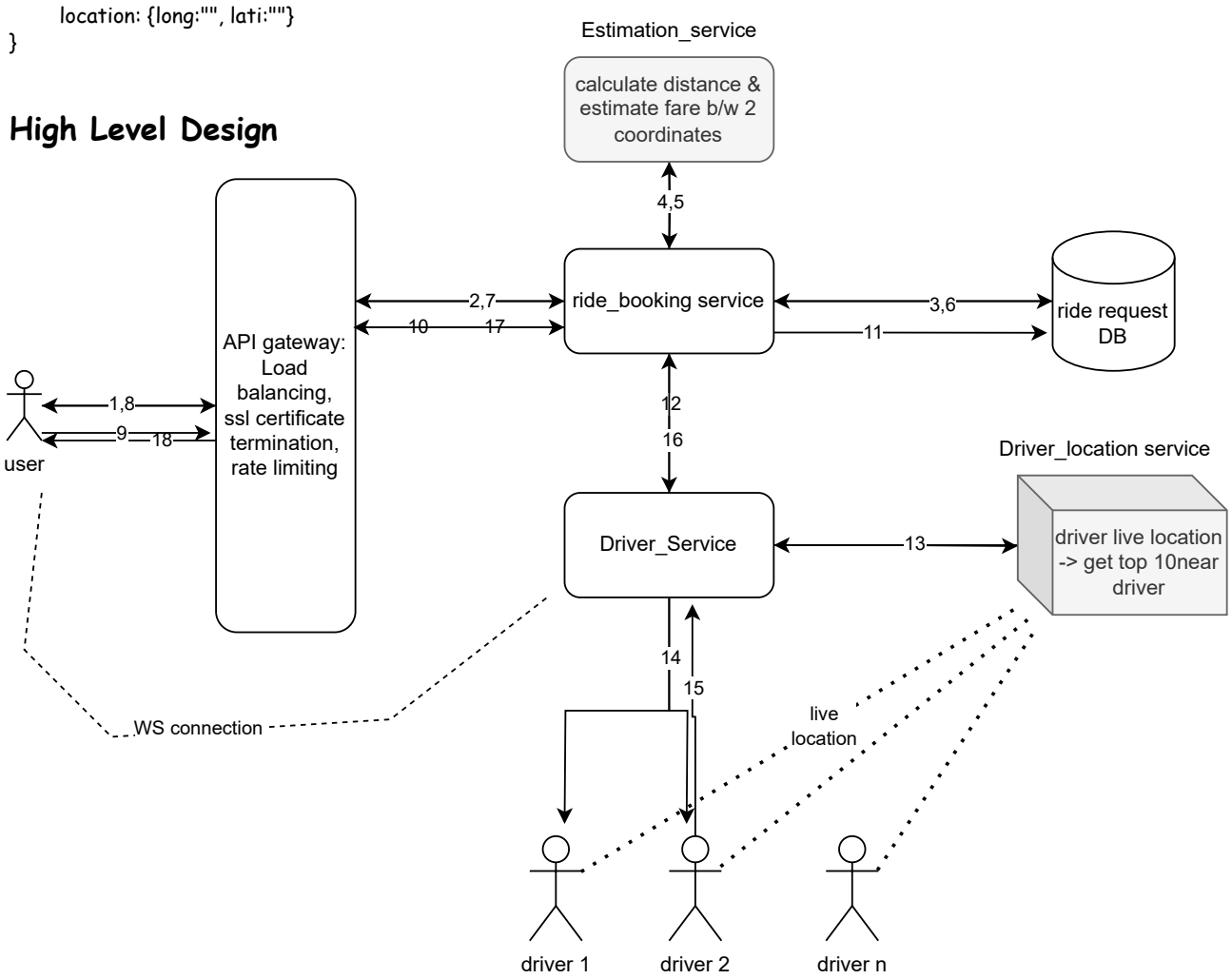
Once driver reach source location A he will start the ride
```
PATCH ride/offer
body {
	ride_request_id: "",
	status: "STARTED"
}
```

In response driver will get destination B
PATCH ride/offer
body {
ride_request_id: "",
destination: {long:"", lati:""},
}

Once driver reach source destination B he will mark the ride as completed
PATCH ride/offer
body {
ride_request_id: "",
status: "COMPLETED"
}

4. Location: drivers will keep on sending their location every 3s/5s when online to the Location service
PATCH /driver/liveLocation
body {
driverID: "",
location: {long:"", lati:""}
}

High Level Design



Flow:

1. User open App, source A is his/her live location & user enters his destination B
2. Booking request send to ride_booking service
3. Ride_booking service creates bookingID, persist request data in the ride request DB
4. Ride_booking service request for estimate price to estimation_service
    4.1 for now have added estimation_service as black box
    4.2. estimation_service get coordinate of A and B, depending on road distance, can demand and traffice, estimation_service returns estimate price for different cab type
    4.3 estimation_service can be in build or 3rd party application service like google maps
5. ride_booking service get the estimate price for diff type of cabs
6. persist data in DB for late analytics - even if users opt not to take the cab
7. ride_booking service returns the ride estimate and options to user
8. User get options and estimate
9. User booked a cab from the provided option
10. reservation request goes to ride_booking service
11. ride_booking service persists the data in DB
12. ride_booking service request driver_service to assign a driver
13. driver_service checks with driver location service to get top 10 drivers that are nearly source location A
14. driver_service send request to 10 drivers one by one, with wait time of 5 sec
15. A driver (2) accepts the request > get source location A coordinates from the driver_service
16. driver_service > ride_booking_service > user to inform about driver OR better option is that a web socket connection is open between driver service and user that tell user about the love location of the driver

Design Deep Dive

Estimation_service

1. estimation_service get source A and source B coordinates
2. With coordinates one can find geoHash
3. Using geoHash can find the road distance between A & B
4. taking courts for live traffic status(from Google Maps) and demand (from own service) estimate calculations can be done for A & B

Driver_location_service

1. driver_location_service keeps live location for all the drivers(cab)
2. Every driver when comes online is in touch with this service and via Web Socket keep on sending its live location
3. One main task of driver_location_service is to return top 10 nearby drivers for a given source location A
4. This can be calculated via
    4.1. brute force but as location is continuously changing its not scalable
    4.2. geoHash - but again as location is continuously changing and geohash would need to calculate road distance
    4.3. quad tree - as continuously changing location we can keep updating the quad tree as well. For less load on quad tree we can keep just non assigned cab in quad tree

We can either use quad tree or even google s2 database for keeping the live location of drivers and returning the top 10(k) drivers near source location A

Web Socket Connection(Near time live location for the driver to user)

1. between driver and driver_location_service we can keep a web socket(WS) so that we always have the live location info for the cab
2. once cab is finalised driver_service can have a WS connection with the user application, to share the live location update for the driver & cab

Consistency - single cab driver is mapped / booked for single cab requester

concurrent / parallel request won't be sent to drivers, at a time, only one driver will be requested with a ride booking

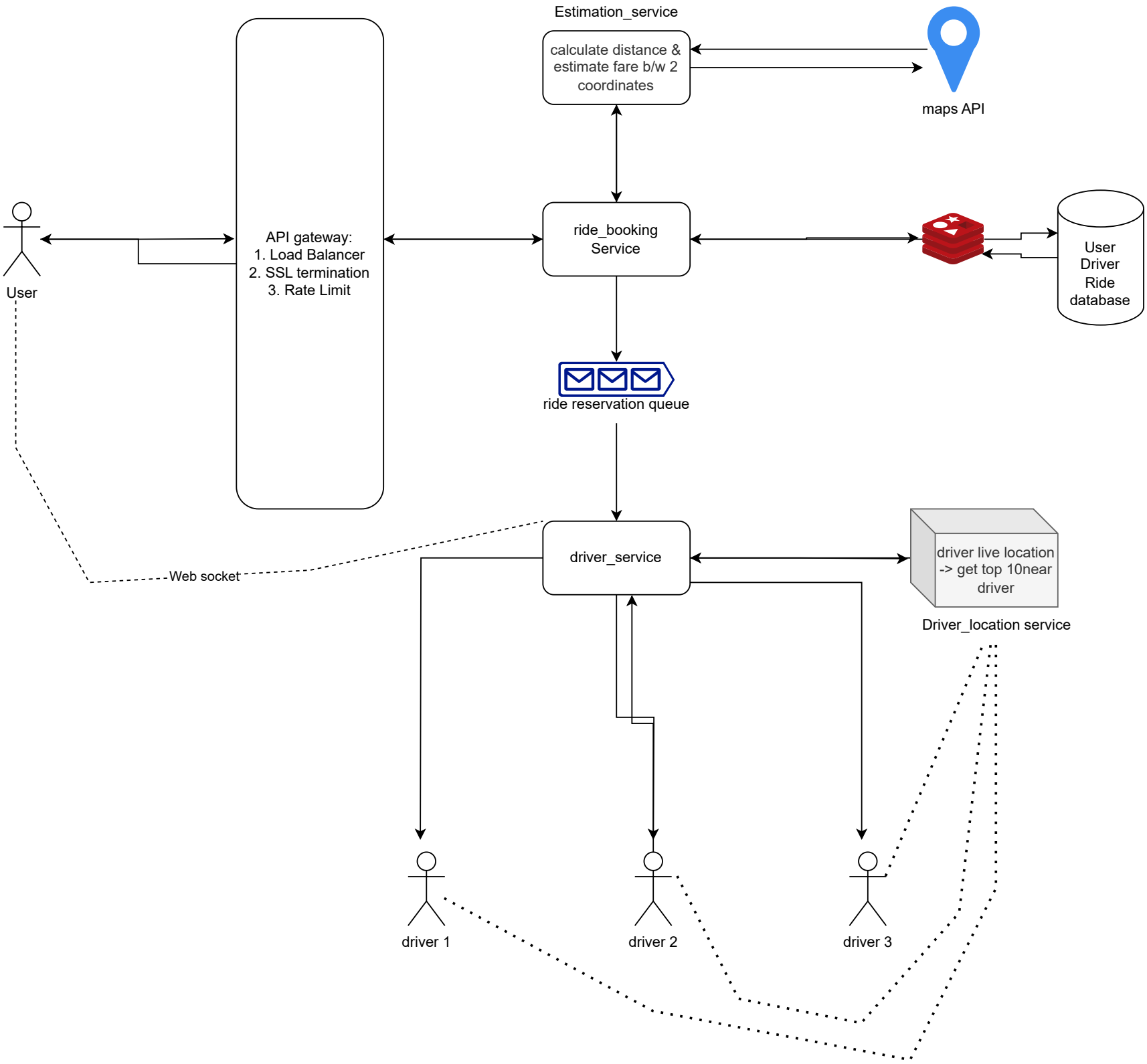
High availability for cab requests

1. We can have redis case for user and driver DB

Scalable for users and geographically (Peak hours)

1. For peak hours or rush hours we can introduce queue in different parts of the system like
    1.1 can request for booking can be in queue
2. multiple instance of every service

Final Design Diagram



Design review by ChatGPT

Your high-level design (HLD) for a ride-booking service like Uber, Ola, or Lyft is quite comprehensive. Here's my structured review with feedback:

What's Well Done?

1. **Clear Functional & Non-Functional Requirements**
    - Well-defined functional and non-functional requirements make it easy to understand the system's expectations.
    - You considered aspects like scalability, high availability, and real-time location tracking.
2. **Well-Structured API Endpoints**
    - The APIs cover essential operations, such as ride requests, booking confirmations, and live location updates.
    - The use of `PUT` and `PATCH` for different update scenarios makes sense.
3. **Solid Database Schema**
    - The schema includes key entities (User, Driver, Ride, Location) with appropriate attributes.
    - The use of foreign keys ensures relational integrity.
4. **Good Use of Geospatial Indexing**
    - You mentioned using **Geohash, Quadtree, or Google S2** for finding nearby drivers, which is a strong design choice.
    - The discussion on trade-offs (brute-force vs. Quadtree vs. S2) shows an understanding of real-world system performance.
5. **WebSocket for Real-Time Updates**
    - WebSocket-based **driver-to-location-service** and **driver-to-user** communication ensures near real-time location tracking.

6. **Scalability Considerations**
    - Redis caching for user and driver data enhances performance.
    - Load balancing, rate limiting, and API gateway are properly accounted for.
    - Queues for handling peak-hour booking requests are a great addition.

What Can Be Improved?

1. **Enhance Consistency & Concurrency Handling**
    - While you mentioned "single cab driver mapped to a single requester," it's important to ensure no two users book the same driver due to race conditions.
    - **Solution**: Use **distributed locking mechanisms** (e.g., Redis Redlock) when assigning a driver.

2. **Ride Matching Algorithm Could Be Detailed More**
    - You discussed finding the "top 10 nearest drivers" but didn't mention:
        - **Sorting mechanism**: Should it prioritize ETA or distance?
        - **Load balancing among drivers**: How do you prevent overloading a driver who gets many requests?
    - **Solution**: You can use a weighted selection algorithm considering **driver ETA, demand, surge pricing, and acceptance rate**.

3. **Optimizing Live Location Updates**
    - Drivers sending updates every **3-5 seconds** can cause high system load.
    - **Solution**:
        - Use an **adaptive frequency update** mechanism where updates are sent less frequently if the driver is idle.
        - Consider **event-driven updates** (e.g., only update location when moving X meters).

4. **Better Handling of No Driver Availability**
    - What happens if no driver is available?
    - **Solution**: Implement a **retry mechanism** (e.g., re-broadcast the request after 5 seconds) or show **alternative transport options**.

5. **Surge Pricing & Demand Handling**
    - You mentioned demand-based fare estimation but didn't elaborate on surge pricing mechanisms.
    - **Solution**:
        - Use machine learning or historical data to **predict demand** and adjust pricing dynamically.
        - Store past ride requests in a **time-series database (e.g., InfluxDB, TimescaleDB)** to analyze traffic trends.

6. **Driver Selection Flow Clarifications**
    - You mentioned a **5-second wait per driver**, but this could increase latency.
    - **Solution**: Instead of waiting sequentially, consider a **parallel driver request model**:
        - Send requests to **multiple drivers simultaneously** (e.g., top 3 drivers at once).
        - The first driver to accept gets assigned.

7. **Ride Cancellation & No-Show Scenarios**
    - What if a user cancels after booking?
    - What if the driver doesn't show up?
    - **Solution**:
        - Implement a **grace period** (e.g., no fee if canceled within 1 minute).
        - Have an **auto-reassign mechanism** to find a new driver.

Overall Rating & Next Steps

- **Completeness:** ★★★★☆ **(4.5/5)**
- **Scalability:** ★★★★★ **(5/5)**
- **Real-time Handling:** ★★★★☆ **(4.5/5)**
- **Edge Cases Consideration:** ★★★☆ **(3.5/5)**

Your design is **strong and well-structured**, but adding **concurrency control, better driver selection logic, and surge pricing optimizations** will make it even more robust.