Back To Course Home

Grokking the System Design Interview

0% completed

Search Course

System Design Problems

- System Design Interviews: A step by step guide
- Designing a URL Shortening service like TinyURL
- Designing Pastebin
- Designing Instagram
- Designing Dropbox
- Designing Facebook Messenger
- Designing Twitter
- Designing Youtube or Netflix
- Designing Typeahead Suggestion
- Designing an API Rate Limiter
- Designing Twitter Search
- Designing a Web Crawler
- Designing Facebook's Newsfeed
- Designing Yelp or Nearby Friends
- Designing Uber backend
- Design Ticketmaster
- Additional Resources

Glossary of System Design Basics

- System Design Basics
- Key Characteristics of Distributed Systems
- Load Balancing
- Caching
- Data Partitioning
- Indexes
- Proxies
- Redundancy and Replication
- SQL vs. NoSQL
- CAP Theorem
- Consistent Hashing
- Long-Polling vs WebSockets vs Server-Sent Events

Appendix

- Contact Us
- Other courses

Course Certificate

Mark Course as Completed

Designing Uber backend

Let's design a ride-sharing service like Uber, which connects passengers who need a ride with drivers who have a car.

Similar Services: Lyft, Didi, Via, Sidecar, etc.
Difficulty level: Hard
Prerequisite: Designing Yelp

We'll cover the following

1. What is Uber?
2. Requirements and Goals of the System
3. Capacity Estimation and Constraints
4. Basic System Design and Algorithm
5. Fault Tolerance and Replication
6. Ranking
7. Advanced Issues

1. What is Uber?

Uber enables its customers to book drivers for taxi rides. Uber drivers use their personal cars to drive customers around. Both customers and drivers communicate with each other through their smartphones using the Uber app.

2. Requirements and Goals of the System

Let's start with building a simpler version of Uber.

There are two types of users in our system: 1) Drivers 2) Customers.

- Drivers need to regularly notify the service about their current location and their availability to pick passengers.
- Passengers get to see all the nearby available drivers.
- Customer can request a ride; nearby drivers are notified that a customer is ready to be picked up.
- Once a driver and a customer accept a ride, they can constantly see each other's current location until the trip finishes.
- Upon reaching the destination, the driver marks the journey complete to become available for the next ride.

3. Capacity Estimation and Constraints

- Let's assume we have 300M customers and 1M drivers with 1M daily active customers and 500K daily active drivers.
- Let's assume 1M daily rides.
- Let's assume that all active drivers notify their current location every three seconds.
- Once a customer puts in a request for a ride, the system should be able to contact drivers in real-time.

4. Basic System Design and Algorithm

We will take the solution discussed in Designing Yelp and modify it to make it work for the above-mentioned "Uber" use cases. The biggest difference we have is that our QuadTree was not built keeping in mind that there would be frequent updates to it. So, we have two issues with our Dynamic Grid solution:

- Since all active drivers are reporting their locations every three seconds, we need to update our data structures to reflect that. If we have to update the QuadTree for every change in the driver's position, it will take a lot of time and resources. To update a driver to its new location, we must find the right grid based on the driver's previous location. If the new position does not belong to the current grid, we have to remove the driver from the current grid and move/reinsert the user to the correct grid. After this move, if the new grid reaches the maximum limit of drivers, we have to repartition it.
- We need to have a quick mechanism to propagate the current location of all the nearby drivers to any active customer in that area. Also, when a ride is in progress, our system needs to notify both the driver and passenger about the current location of the car.

Although our QuadTree helps us find nearby drivers quickly, a fast update in the tree is not guaranteed.

Do we need to modify our QuadTree every time a driver reports their location? If we don't update our QuadTree with every update from the driver, it will have some old data and will not reflect the current location of drivers correctly. If you recall, our purpose of building the QuadTree was to find nearby drivers (or places) efficiently. Since all active drivers report their location every three seconds, therefore there will be a lot more updates happening to our tree than querying for nearby drivers. So, what if we keep the latest position reported by all drivers in a hash table and update our QuadTree a little less frequently? Let's assume we guarantee that a driver's current location will be reflected in the QuadTree within 15 seconds. Meanwhile, we will maintain a hash table that will store the current location reported by drivers; let's call this DriverLocationHT.

How much memory we need for DriverLocationHT? We need to store DriveID, their present and old location, in the hash table. So, we need a total of 35 bytes to store one record:

1. DriverID (3 bytes - 1 million drivers)
2. Old latitude (8 bytes)
3. Old longitude (8 bytes)
4. New latitude (8 bytes)
5. New longitude (8 bytes) Total = 35 bytes

If we have 1 million total drivers, we need the following memory (ignoring hash table overhead):

1 million * 35 bytes => 35 MB

How much bandwidth will our service consume to receive location updates from all drivers? If we get DriverID and their location, it will be (3+16 => 19 bytes). If we receive this information every three seconds from 500K daily active drivers, we will be getting 9.5MB per three seconds.

Do we need to distribute DriverLocationHT onto multiple servers? Although our memory and bandwidth requirements don't require this, since all this information can easily be stored on one server, but, for scalability, performance, and fault tolerance, we should distribute DriverLocationHT onto multiple servers. We can distribute based on the DriverID to make the distribution completely random. Let's call the machines holding DriverLocationHT the Driver Location server. Other than storing the driver's location, each of these servers will do two things:

1. As soon as the server receives an update for a driver's location, they will broadcast that information to all the interested customers.
2. The server needs to notify the respective QuadTree server to refresh the driver's location. As discussed above, this can happen every 10 seconds.

How can we efficiently broadcast the driver's location to customers? We can have a **Push Model** where the server will push the positions to all the relevant users. We can have a dedicated Notification Service that can broadcast the current location of drivers to all the interested customers. We can build our Notification service on a publisher/subscriber model. When a customer opens the Uber app on their cell phone, they query the server to find nearby drivers. On the server side, before returning the list of drivers to the customer, we will subscribe the customer for all the updates from those drivers. We can maintain a list of customers (subscribers) interested in knowing the location of a driver and, whenever we have an update in DriverLocationHT for that driver, we can broadcast the current location of the driver to all subscribed customers. This way, our system makes sure that we always show the driver's current position to the customer.

How much memory will we need to store all these subscriptions? As we have estimated above, we will have 1M daily active customers and 500K daily active drivers. On average let's assume that five customers subscribe to one driver. Let's assume we store all this information in a hash table so that we can update it efficiently. We need to store driver and customer IDs to maintain the subscriptions. Assuming we will need 3 bytes for DriverID and 8 bytes for CustomerID, we will need 21MB of memory.

(500K * 3) + (500K * 5 * 8) ~= 21 MB

How much bandwidth will we need to broadcast the driver's location to customers? For every active driver, we have five subscribers, so the total subscribers we have:

5 * 500K => 2.5M

To all these customers we need to send DriverID (3 bytes) and their location (16 bytes) every second, so, we need the following bandwidth:

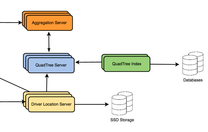
2.5M * 19 bytes => 47.5 MB/s

How can we efficiently implement Notification service? We can either use HTTP long polling or push notifications.

How will the new publishers/drivers get added for a current customer? As we have proposed above, customers will be subscribed to nearby drivers when they open the Uber app for the first time, what will happen when a new driver enters the area the customer is looking at? To add a new customer/driver subscription dynamically, we need to keep track of the area the customer is watching. This will make our solution complicated; how about if instead of pushing this information, clients pull it from the server?

How about if clients pull information about nearby drivers from the server? Clients can send their current location, and the server will find all the nearby drivers from the QuadTree to return them to the client. Upon receiving this information, the client can update their screen to reflect the current positions of the drivers. Clients can query every five seconds to limit the number of round trips to the server. This solution looks simpler compared to the push model described above.

Do we need to repartition a grid as soon as it reaches the maximum limit? We can have a cushion to let each grid grow a little bigger beyond the limit before we decide to partition it. Let's say our grids can grow/shrink an extra 10% before we partition/merge them. This should decrease the load for a grid partition or merge on high traffic grids.



How would "Request Ride" use case work?

1. The customer will put a request for a ride.
2. One of the Aggregator servers will take the request and asks QuadTree servers to return nearby drivers.
3. The Aggregator server collects all the results and sorts them by ratings.
4. The Aggregator server will send a notification to the top (say three) drivers simultaneously, whichever driver accepts the request first will be assigned the ride. The other drivers will receive a cancellation request. If none of the three drivers respond, the Aggregator will request a ride from the next three drivers from the list.
5. Once a driver accepts a request, the customer is notified.

5. Fault Tolerance and Replication

What if a Driver Location server or Notification server dies? We would need replicas of these servers, so that if the primary dies the secondary can take control. Also, we can store this data in some persistent storage like SSDs that can provide fast IOs; this will ensure that if both primary and secondary servers die we can recover the data from the persistent storage.

6. Ranking

How about if we want to rank the search results not just by proximity but also by popularity or relevance?

How can we return top rated drivers within a given radius? Let's assume we keep track of the overall ratings of each driver in our database and QuadTree. An aggregated number can represent this popularity in our system, e.g., how many stars does a driver get out of ten? While searching for the top 10 drivers within a given radius, we can ask each partition of the QuadTree to return the top 10 drivers with a maximum rating. The aggregator server can then determine the top 10 drivers among all the drivers returned by different partitions.

7. Advanced Issues

1. How will we handle clients on slow and disconnecting networks?
2. What if a client gets disconnected when they are a part of a ride? How will we handle billing in such a scenario?
3. How about if clients pull all the information, compared to servers always pushing it?