ARCHITECTURE OF UBER

Overall Primary Functionality of UBER:

Whenever a customer requests for a cab, the nearest drivers are sent notifications. One of them accepts the requests and that particular driver is assigned with a particular customer. Below are the sequence of things that are to be tracked:

1. Customer requests for the Cab
2. Tracking the location of the drivers and finding the nearby drivers
3. Mapping the drivers with the customers

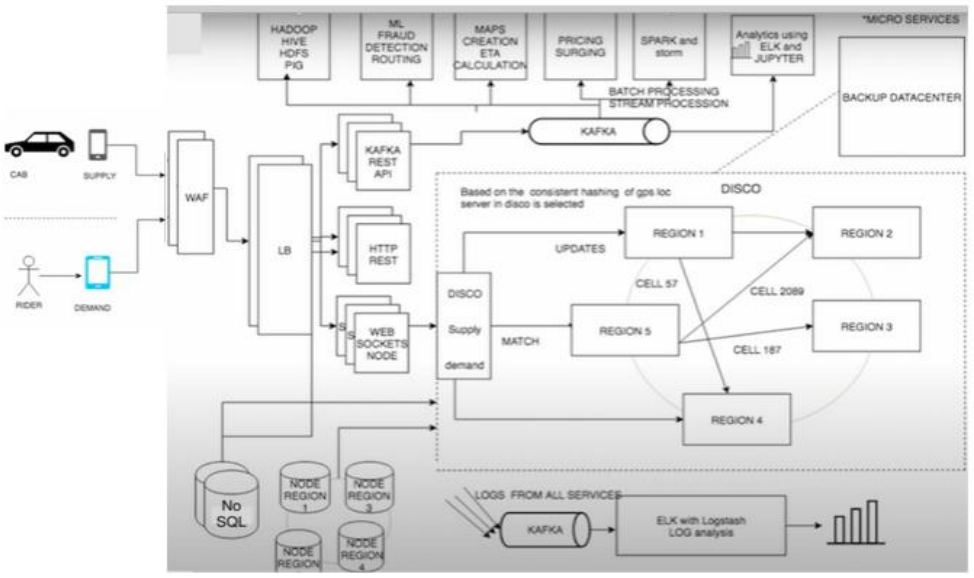

Figure 1 Uber Architecture (3)

Note that, here, Cabs are considered as supply service and customers are considered as demand service. So, we need to map demand to the supply

What happens internally, when a customer requests for a cab from his/her mobile?

When a user requests for a cab, the request is sent to Dispatch Optimizer(which maps the driver to the customer) via Web Sockets. Web Sockets are mainly used for sending asynchronous messages between client and server(Dispatch Optimiser). Web Sockets are **Load Balanced**, as there will be thousands and thousands of customers, sending the requests for the cabs. We have a Web Application Firewall, placed in front of Load Balancer. This security firewall blocks the requests from blocked IP's, bots or block the requests from the regions where Uber is not yet launched

How are we going to track the location of the drivers?

For every 4 or 5 seconds, the driver shares his location to the server. For this, Uber uses **KAFKA REST API'S**. As thousands of drivers keep on sending their locations, these API's are designed to be Load Balanced. From the load balancer, this data is archived in the messaging queue called **KAFKA**, through KAFKA REST API's. It is the responsibility of KAFKA to route this data to Databases. Also, Uber uses **HADOOP** for performing analytics on the data.

How are we going to map drivers with the customers?

As we know, the customer needs to be mapped with the nearest driver. So, this is achieved with the help of location data of both customers and the driver. For this, uber takes the help of **Google-S2 Library.** Google-S2 Library captures the spherical map and divides it into small tiny cells for about 1km or 2km cells. Each cell is given with the unique ID. So, with this we can find the nearest drivers, who are about 1 km or 2 km radius. All the nearby drivers are sent the notifications and once the driver accepts the request, he is mapped with the customer.

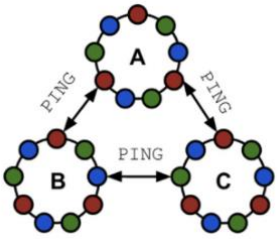

Figure 2 Ringpop(6)

Uber's Matching System(Dispatch Optimization):

Uber's matching system is built of **Ringpop**, which is a Node.js library. This Ringpop enables the coordination and cooperation of different servers by distributing the work among multiple servers using **Consistent Hashing**. So, here each server is responsible for handling some group of Cell Ids, that were assigned using Google S2 Library (For example, Cells 1-20 are handled by Server1, Cells 21-40 are handled by Server2 etc). Also one server talks to the other server using RPC(Remote Procedure) calls and also Ringpop uses **SWIM** protocol, which enables each server to know the responsibility of every other server, so that each of them discovers others and also for the easy detection of failures.

When a user requests the cab, the particular cell id of the user is passed to the Demand Service of Matching System. Then the matching system, with the help of Ringpop, assigns the matching responsibility to the respected server-holding the cellId that of the user. The assigned server then finds out the nearby drivers and sends the list to the Supply Service of the Matching System, in the sorted order, by calculating the smallest ETA(Estimated Time of Arrival). The Supply Service, in turn sends the notifications to the listed nearby drivers and whoever accepts the first, is assigned to the customer.

Comparison of Standard Web Architectures with Uber Architecture:

- **Load Balancer:** Similar to the standard web architecture, the KAFKA REST API'S (as thousands of cab drivers share locations) and also the web sockets(as thousands of customers requests for the cab, and these requests are routed to Web Sockets) are Load Balanced in Uber Architecture.
- **Web App Servers:** Uber's matching system(also termed as Dispatch Optimizer – multiple servers) serves as a web app server, which processes the requests from the users and maps the drivers with the customers.

- **Caching Service:** Uber uses **Redis** for the purpose of caching and queuing. Also for reducing the number of connections to the cache servers, uber uses **Twemproxy.** So, this provides Scalability to the cache layer.
- **Job Queues:** Uber uses KAFKA Messaging Queue. So, it is the responsibility of KAFKA to route the data(that is collected via driver and rider applications), to different consumers.
- **Databases & Data Warehouses**: Like standard architectures, Ubers uses many databases such as MySQL/PostgreSQL, Key-Value datastores, Cassandra etc(as shown in the figure). It uses Hadoop for performing analytics on data.
- **Cloud Storage:** Uber uses its own cloud storage apart from AWS and Google Cloud Storage.
- **Full Text Search Service:** Uber uses Elasticsearch for many of the analytical services. This Elasticsearch was built on Apache Lucene, for performing full text keyword searching, which is used for storing documents and inverted indexes. So as in standard web architectures, this inverted index is used for filtering the text.

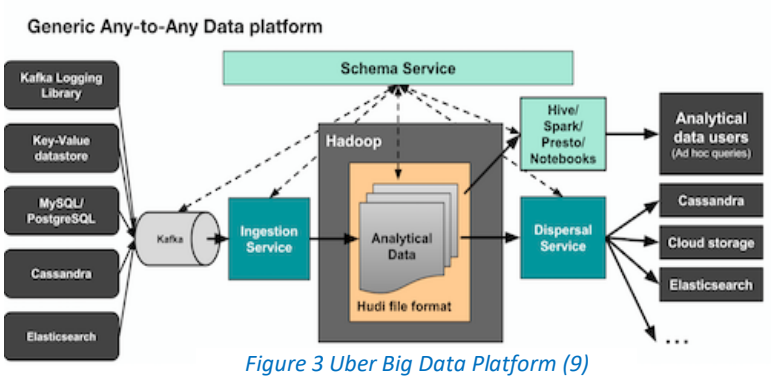

Figure 3 Uber Big Data Platform (9)

- **Services:** Uber has several services such as Payments Service, Supply service, Demand service, Marketplace services etc.
- **CDN:** Uber has servers hosted in different parts of the world. They build data centers nearest to the regions, so as to ensure faster response to their uses.

Differences Observed:

- The only difference that I have observed is that Uber has **Web Application Firewall(WAF)**, placed in front of load Balancer, which I have not seen in Standard web architecture design. I observed that overall system design of any Application follows the same set of components. So, some of the interesting new things that I have learned here are about Kafka, Ringpop, Redis and Elasticsearch.

References

1. “Web Scalability for Startup Engineers,” *O'Reilly Online Learning*. [Online]. Available: https://learning.oreilly.com/library/view/web-scalability-for/9780071843669/ch01.html. [Accessed: 10-Sep-2020].

2. J. Fulton, “Web Architecture 101,” *Medium*, 12-Dec-2018. [Online]. Available: https://engineering.videoblocks.com/web-architecture-101-a3224e126947. [Accessed: 10-Sep-2020].

3. N. L, “UBER system design,” *Medium*, 08-Sep-2018. [Online]. Available: https://medium.com/@narengowda/uber-system-design-8b2bc95e2cfe. [Accessed: 12-Sep-2020].

4. E. Haddad, “Service-Oriented Architecture: Scaling the Uber Engineering Codebase As We Grow,” *Uber Engineering Blog*, 11-Nov-2019. [Online]. Available: https://eng.uber.com/service-oriented-architecture/. [Accessed: 13-Sep-2020].

5. Gluck, “Introducing Domain-Oriented Microservice Architecture,” *Uber Engineering Blog*, 07-Aug-2020. [Online]. Available: https://eng.uber.com/microservice-architecture/. [Accessed: 10-Sep-2020].

6. Lozinski, L. (2020, March 03). How Ringpop from Uber Engineering Helps Distribute Your Application. Retrieved September 8, 2020, from https://eng.uber.com/ringpop-open-source-nodejs-library/.[Accessed: 10-Sep-2020].

7. J. Shen, “Introducing AresDB: Uber's GPU-Powered Open Source, Real-time Analytics Engine,” *Uber Engineering Blog*, 12-Aug-2019. [Online]. Available: https://eng.uber.com/aresdb/. [Accessed: 15-Sep-2020].

8. Y. N. Chinmay Soman, “uReplicator: Uber Engineering's Robust Apache Kafka Replicator,” *Uber Engineering Blog*, 27-Feb-2020. [Online]. Available: https://eng.uber.com/ureplicator-apache-kafka-replicator/. [Accessed: 14-Sep-2020].

9. R. Shiftehfar, “Uber's Big Data Platform: 100+ Petabytes with Minute Latency,” *Uber Engineering Blog*, 06-Apr-2020. [Online]. Available: https://eng.uber.com/uber-big-data-platform/. [Accessed: 15-Sep-2020].