in
Yandex.Metrica

Monday
Sep142015

How Uber Scales Their Real-Time Market Platform

MONDAY, SEPTEMBER 14, 2015 AT 8:56AM

Reportedly Uber has grown an astonishing 38 times bigger in just four years. Now, for what I think is the first time, Matt Ranney, Chief Systems Architect at Uber, in a very interesting and detailed talk--Scaling Uber's Real-time Market Platform---tells us a lot about how Uber's software works.

If you are interested in Surge pricing, that's not covered in the talk. We do learn about Uber's dispatch system, how they implement geospatial indexing, how they scale their system, how they implement high availability, and how they handle failure, including the surprising way they handle datacenter failures using driver phones as an external distributed storage system for recovery.

The overall impression of the talk is one of very rapid growth. Many of the architectural choices they've made are a consequence of growing so fast and trying to empower recently assembled teams to move as quickly as possible. A lot of technology has been used on the backend because their major goal has been for teams to get the engineering velocity as high as possible.

After a understandably chaotic (and very successful) start it seems Uber has learned a lot about their business and what they really need to succeed. Their early dispatch system was a typical just make it work type affair that assumed at a deep level it was moving only people. Now that Uber's mission has grown to handle boxes and groceries as well as people, their dispatch system has been abstracted and put on very solid and smart architectural foundation.

Though Matt thinks their architecture might be a little crazy, the idea of using a consistent hash ring with a gossip protocol seems spot on for their use case.

It's hard not to be captivated for Matt's genuine enthusiasm for what he's working on. When talking about DISCO, their dispatch system, he says in an excited tone

that it's like the traveling salesman problem from school. It's a cool Computer Science thing. Even though the solution isn't optimal, it's the traveling salesman at an interesting scale, in real-time, in the real-world, built out of fault tolerant scalable components. How cool is that?

So let's see how Uber works on the inside. Here's my gloss on Matt's talk:

Stats

- The goal for Uber's geospatial index is a million write per second. Many multiples of that for reading.
- The dispatch system has thousands of nodes.

Platform

- Node.js
- Python
- Java
- Go
- Native applications on iOS and Android
- Microservices
- Redis
- Postgres
- MySQL
- Riak
- Twitter's Twemproxy for Redis
- Google's S2 Geometry Library
- ringpop - consistent hash ring
- TChannel - network multiplexing and framing protocol for RPC
- Thrift

General

- Uber is a transportation platform for connecting riders with driver partners.
- Challenge: **matching a dynamic demand with a dynamic supply in realtime**. On the supply side drivers are free to do whatever they want. On the demand side riders need transportation whenever they want.
- Uber's Dispatch system is a real-time market platform that matches drivers with riders using mobile phones.

- New Year's Eve is the busiest time of the year for Uber.
- It's easy to lose track of how quickly the industry has made such enormous progress. Technology is getting so good so fast that things that were recently amazing quickly fade into the background. Twenty-thirty years ago mobile phones, Internet, and GPS were barely science fiction, now we barely notice it.

Architecture Overview

- What drives it all are riders and drivers on their mobile phones running native applications.
- The backend is primarily servicing mobile phone traffic. Clients talk to the backend over mobile data and the best effort Internet. Can you imagine 10 years ago basing a business on mobile data? It's awesome that we can do this sort of thing now. No private networks are used, no fancy QoS (Quality of Service), just the open Internet.
- Clients connect to the dispatch system which matches drivers and riders, **the supply and demand**.
- Dispatch is written almost entirely in node.js.
    - Plans were to move it to io.js, but since then io.js and node.js have merged.
    - You can do interesting distributed systems work in javascript.
    - It's hard to underestimate **the productive powers of enthusiasm and node developers** are very enthusiastic. They can get a lot done very quickly.
- The whole Uber system probably seems pretty simple. Why do you need all these subsystems and all these people? As long as it seems that way then that is the mark of success. There's a lot to do and as long as it seems simple they've done their job.
- **Maps/ETA** (estimated time of arrival). For Dispatch to make an intelligent choice it's necessary to get maps and routing information.
    - Street maps and historical travel times are used to estimate current travel times.
    - The language depends a lot on what system is being integrated with. So there's Python, C++, and Java

- **Services**. There are vast amounts of business logic services.
    - A microservice approach is used.
    - Mostly written in Python.
- **Databases**. A lot of different databases are used.
    - The oldest systems were written in Postgres.
    - Redis is used a lot. Some are behind Twemproxy. Some are behind a custom clustering system.
    - MySQL
    - Uber is building their own distributed column store that's orchestrating a bunch of MySQL instances.
    - Some of the Dispatch services are keeping state in Riak.
- **Post trip pipeline**. A lot of processing must happen after a trip has completed.
    - Collect ratings.
    - Send emails.
    - Update databases.
    - Schedule payments.
    - Written in Python.
- **Money**. Uber integrates with many payment systems.

The Old Dispatch System

- Limitations in the original Dispatch system were **starting to limit the growth of the company**, so it had to change.
- Mostly rewrote the whole thing, despite what Joel Spolsky said. All the other systems weren't touched and even some services within the Dispatch system survived.
- The old system was designed for private transportation, which made a lot of assumptions:
    - One rider per vehicle, which wouldn't work for Uber Pool.
    - The idea of moving people was baked deep into the data models and interfaces. This constrained moving into new markets and new products, like moving food and boxes.

    - The original version was sharded by city. Which is good for scalability because each city can run independently. As more and more cities were added however it became increasingly hard to manage. Some cities are big and some are small. Some cities have big load spikes and other don't.

- Because so much is being built so fast they don't have single points of failure, they have multiple points of failure.

The New Dispatch System

- To fix city sharding and support more products the idea of supply and demand had to be generalized, so a **Supply Service and a Demand Service** were created.
- The Supply Service tracks capabilities and the state machines of all of the supply.
    - To track vehicles there are many attributes to model: number of seats, type of vehicle, presence of a car seat for children, can a wheelchair be fit, and so on.
    - Allocation needs to be tracked. A vehicle, for example, may have three seats but two of those are occupied.
- The Demand Service tracks requirements, orders, and all of the aspects of demand.
    - If a rider requires a car seat that requirement must be matched against inventory.
    - If a rider doesn't mind sharing a car for a cheaper rate that must be modeled.
    - What if a box needs to be moved or food delivered?
- The logic to match all of supply and demand is a service called DISCO (dispatch optimization)
    - The old system would only match on currently available supply, that means cars that are on the road currently awaiting for work.
    - DISCO supports planning into the future and making use of information as it becomes available. For example, revising a route on an in progress trip.
    - **geo by supply**. A geospatial index is required for DISCO to make its decisions

based on where all the supply is and where it is expected to be.

- **geo by demand**. A geo index is also required for demand
- A better routing engine is required to make use of all this information.

Dispatch

- As vehicles move around location updates are sent to geo by supply. To match riders to drivers or just display cars on a map, DISCO sends a request to geo by supply.
- Geo by supply makes a coarse first pass filter to get nearby candidate that meet requirements.
- Then the list and requirements are sent to routing / ETA to compute the ETA of how nearby they are not geographically, but by the road system.
- Sort by ETA then send it back to supply to offer it to a driver.
- In airports they have to emulate a virtual taxi queue. Supply must be queued in order to take into account the order in which they arrive.

Geospatial Index

- Must be super scalable. Design goal is to **handle a million writes per second**. The write rate is derived from drivers that send update every 4 seconds as they move around.
- The read goal is for many more reads than writes per second because everyone with an open app is doing reads.
- The old geospatial index worked well by making a simplifying assumption, it only tracked dispatchable supply. The majority of the supply was busying doing something, so the subset of available supply was easy to support. There was a global index stored in memory in a handful of processes. It was easy enough to do very naive matching.
- In the new world **all supply in every state must be tracked**. In addition, their projected route must also be tracked. This is much more data.
- The new service **runs on hundreds of processes**.
- The earth is a sphere. It’s hard to do summarization and approximation based purely on longitude and latitude. So Uber divides the earth into tiny cells using the Google S2 library. Each cell has a unique cell ID.

- Using an int64 every square centimeter on earth can be represented. Uber uses a level 12 cell, which are 3.31 km(2) to 6.38 km(2), depending on where on earth you are. The boxes change shape and size depending on where on the sphere they are.
- S2 can give the coverage for a shape. If you want to draw a circle with a 1km radius centered on London, S2 can tell what cells are needed to completely cover the shape.
- Since each cell has an ID the ID is used as a sharding key. When a location comes in from supply the cell ID for the location is determined. Using the cell ID as a shard key the location of the supply is updated. It is then sent out to a few replicas.
- When DISCO needs to find the supply near a location, a circle's worth of coverage is calculated centered on where the rider is located. Using the cell IDs from the circle area all the relevant shards are contacted to return supply data.
- It's all scalable. Even though it's not as efficient as you might like, and since fanout is relatively cheap, the write load can always be scaled by adding more nodes. The read load is scaled through the use of replicas. If more read capacity is needed the replica factor can be increased.
- A limitation is the cell size is fixed at the level 12 size. Dynamic cell size might be supported in the future. There's a tradeoff as the smaller the cell size the greater the fanout for queries.

Routing

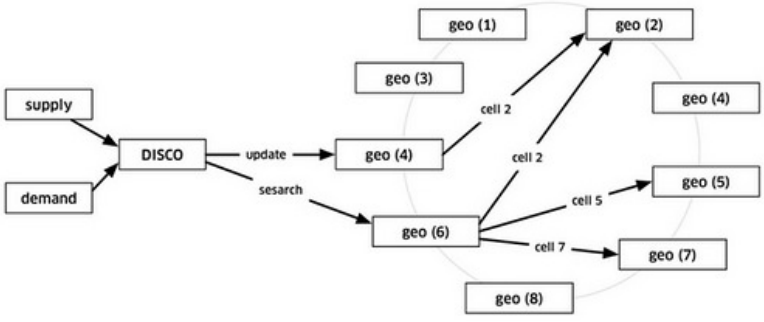
- After there's an answer from geospatial the options must be ranked.
- The are several high level goals:
    - **reduce extra driving**. Driving is people's jobs so there's a desire for them to be productive. They are getting paid for extra driving around. Ideally drivers would be on a continuous trip. A bunch of work would be queued up and they would get paid for all of it.
    - **reduce waiting**. Riders should wait as little as possible.
    - **lowest overall ETA**.

- The old system was to let demand search currently available supply, match and be done with it. It's easy to implement and is simple to understand. And it worked pretty well for private transportation.
- Good choices can't be made by looking only at current availability.
- The idea is that a driver currently transporting a rider may be a better match for a customer asking for a ride than is a currently idle driver that is farther away. Picking the on-trip driver minimizes the customer wait time and minimizes the amount of extra driving time for a more remote driver.
- Dynamic conditions are better handled by this model of trying to see into the future.
    - For example, if a driver comes online near a customer but another driver was already dispatched from further away there's no way to change the dispatch decisions.
    - Another example involved customers that are willing to share a ride. It's possible to a lot more optimizations by trying to predict the future in very complicated scenarios.
- All these decisions become more interesting when considering delivering boxes or food. In these cases people generally have something else to do, so there are different tradeoffs involved.

Scaling Dispatch

- Dispatch is built using node.js.
- They are building a stateful service so stateless approaches to scaling won't work.
- Node runs in a single process so some method must be devised to run Node on multiple CPUs on the same machine and to multiple machines.
- There's a joke about reimplementing all of Erlang in Javascript.
- The solution for scaling Node was *ringpop*, a consistent hash ring with a gossip protocol, implementing a scalable, fault-tolerant application-layer sharding.
- In CAP terminology ringpop is an **AP system**, trading consistency for availability. It's better to explain away a few inconsistencies than it is to have a down service. It's better to be up and occasionally make an error.

- ringpop is an embeddable module that's included in each Node process.
- Node instances gossip around a membership set. Once all the nodes agree who each other are, they can make lookup and forwarding decisions independently and efficiently.
- This is really scalable. Add more processes and more work gets done. It can be used to shard data, or as a distributed locking system, or coordinating a rendezvous point for pub/sub or a long-poll socket.
- The gossip protocol is based on SWIM. A few improvements have been made to improve convergence time.
- A list of members that are up are gossiped around. As more nodes are added it is scalable. The 'S' in SWIM is for scalable and really does work. It has **scaled to thousands of nodes so far**.
- SWIM combines health checks with membership changes as part of the same protocol.
- In a ringpop system there are all these Node processes containing ringpop modules. They gossip around the current membership.
- Externally, if DISCO wants to consume geospatial, every node is equivalent. A random healthy node is selected. Wherever the request lands is responsible for forwarding the request to the right node by using the hash ring lookup. It looks like:



- It may sound crazy to have all these hops and peers talking to each other, but it yields some really nice properties, like services can be scaled by adding instances on any machine.
- ringpop is built on Uber's own RPC mechanism called *TChannel*.
  - It's a bidirectional request/response protocol that was inspired by Twitter's Finagle.
  - An important goal was to control performance across a lot of different languages. Especially

in Node and Python a lot of the existing RPC mechanisms don't work very well. Wanted redis level performance. **TChannel is already 20 times faster than HTTP**.

- Wanted a high performance forwarding path so intermediaries could make forwarding decisions very easily, without having to understand the full payload.
- Wanted proper pipelining so there wasn't head-of-line blocking, requests and responses could be sent in either direction at any time, and every client is also a server.
- Wanted to bake-in payload checksums and tracing and first class features. Every request should be traceable as it wends its way through the system.
- Wanted a clean migration path off of HTTP. HTTP can be encapsulated very naturally in TChannel.
- **Uber is getting out of the HTTP and Json business**. Everything is moving to Thrift over TChannel.

- ringpop is doing all its gossip over TChannel based persistent connections. These same persistent connections are used to fanout or forward application traffic. TChannel is also used to talk between services.

Dispatch Availability

- **Availability matters a lot**. Uber has competitors and switching costs are very low. If Uber is down even briefly that money goes to someone else. Other products are more sticky and customers will just try again later. That's not necessarily true with Uber.
- **Make everything retryable**. If something doesn't work it has to be retryable. That's how to route around failure. This requires all requests to be idempotent. Retrying a dispatch, for example, can't dispatch them twice or charge someone's credit card twice.
- **Make everything killable**. Failure is a common case. Killing processes randomly shouldn't do damage.
- **Crash only**. There are no graceful shutdowns. Graceful shutdowns are not what needs to be practiced. What needs to be practiced is when unexpected things break.

- **Small pieces**. To minimize the cost of things failing break them into smaller pieces. It might be possible to handle global traffic in one instance, but what happens when that dies? If there is a pair of them and one fails then capacity is cut in half. So services need to be broken up. It sounds like a technology problem, but it's more of a cultural problem. It's just easier to have a pair of databases. It's a natural thing to do, but pairs are bad. Randomly killing them it's really risky if you'll be able to automatically promote one and restart the new secondary.

- **Kill everything**. Even kill all the databases to make sure it's possible to survive that kind of failure. This required changing decisions about what database to use.They chose Riak instead of MySQL. It also means using ringpop instead of redis. Killing a redis instance is an expensive operation, they are usually pretty big and expensive to have go away.

- **Break it up into smaller chunks**. Talking about cultural shift. Typically Service A will talke to Service B through a load balancer. What if the load balancer dies? How are you going to deal with it? If you don't ever exercise that path you'll never know. So you have to kill the load balancer. How do you route around the load balancer? Load balancing logic has to be put in the service itself. Clients are required to have some intelligence to know how to route around problems. This is philosophically similar to how Finagle works.

- To make the whole system scale and handle back pressure, a service discovery and routing system has be created out of a cluster of ringpop nodes.

Total Datacenter Failure

- It doesn't happen very often, but there could be an unexpected cascading failure or an upstream network provider could fail.

- Uber maintains a backup datacenter and the switches are in place to route everything over to the backup datacenter.

- The problem is the data for in-process trips may not be in the backup datacenter. Rather than replicate data they use driver phones as a source of trip data.

- What happens is the Dispatch system **periodically sends an encrypted State Digest down to driver phones**. Now let's say there's a datacenter failover. The next time the driver phone sends a location update to the Dispatch system the Dispatch system

will detect that it doesn't know about this trip and ask the for the State Digest. The Dispatch system then updates itself from the State Digest and the trip keeps on going like nothing happened.

The Downside

- The downside to Uber's approach to solving scalability and availability problems is potentially high latencies with Node processes forwarding requests to each other and sending messages with big fanouts.
- In a fanout system tiny blips and glitches have a surprisingly large impact. The higher the fanout is in a system the better the chance of having a high latency request.
- A good solution is to have **backup requests with cross server cancellation**. This is baked-in to TChannel as a first class feature. A request is sent to Service B(1) along with the information that the request is also being sent to Service B(2). Then some delay later the request is sent to Service B(2). When B(1) completes the request it cancels the request on B(2). With the delay it means in the common case B(2) didn't perform any work. But if B(1) does fail then B(2) will process the request and return a reply in a lower latency than if B(1) was tried first, a timeout occurred, and then B(2) is tried.
- See Google On Latency Tolerant Systems: Making A Predictable Whole Out Of Unpredictable Parts for more background on all this.

Related Articles

- On HackerNews
- S2map.com - see S2 coverage and draw shapes

Todd Hoff | 1 Comment | Permalink | Share Article Print Article Email Article

in Example

Like 531 people like this. Be the first of your friends.

Tweet

Reader Comments (1)

Thank you for writing this article. It was captivating reading it!

September 17, 2015 | elias