TravelLog: An Exercise in Scalable Cloud Data Services

Yash Patil

1 Introduction

TravelLog's microservices are designed in such a way that ensure the service to be scalable, available, and reliable. CDC guidelines state that people should try to maintain a distance of at least 6 feet. We assume that close contact with someone is being within this distance, $d = 6$. The CDC also defines a close contact period of having 10 minutes of contact with that person, so we will match *contacts* based off of a time threshold of 10 minutes. Given that users will have to constantly be tracked, we assume that the most common operation for this system is the *update* operation, which pings the users location to our service. Furthermore, if this application were to be used before stay at home orders have been issued, we assume that the amount of *update* calls will be even greater. We assume that the $person - interval$, $place - time$, and *contacts* operations will be called far less than the *update* operation; however, the $person - interval$ and $place - time$ operations will be computationally expensive. One assumption we do make is that since the *update* operations will primarily be during the day when users are awake and moving, we can transfer the load of our system to handle the data analytics and processing work for the latter three operations during the night hours. A key assumption for TravelLog is that the web analytics components are not trying to constantly show where a user is moving; rather, TravelLog is a tool for public health officials and governments to get insights on the movement of people, how their policies are performing, and potential hotspots.

2 SLOs

- *update*: This operation will need to have high availability. In addition, this service will need to have high throughput because it is serving other requests where that it is a requirement.
- $person - interval$ and $place - time$: The former gives a path of a person's location in a given interval and the latter gives a paths of all individuals in a given location at a certain time. Both of these operations will be client facing and involve heavy data processing. Therefore, we optimize for low latency, high throughput, and high availability. We should respond fast to each request and be able to cache them. The data-processing underlying these operations are grouped into a service of their own and will need to have high throughput and end-end latency since it is accessing many parts of this system.
- *contacts*: This is also a user-facing operation and will need to have low latency and high throughput. In addition, there needs to be a high level of consistency and low error-rate because the incorrect *contacts* should not be shown.

3 Services

This service will be deployed on Kubernetes, which allows TravelLog to add policies to the services and scale those as needed. The services described below describe an instance of TravelLog in a single data center. As the service grows to expand more people in different regions, we can add different availability zones for the service. Furthermore, if an entire availability zone were to fail, we can route requests to another data center.

3.a Load Balancer

When a request comes into TravelLog after getting a VIP from the Edge Point of Presence (discussed in section 4), it will hit an L4 Load Balancer. This LB will route requests based on the incoming traffic, and not based off of information in the request itself. We choose not to route requests based off the information in the request itself for this first layer because we want to route requests to the Forward Tier as fast as possible, which we will spend more resources into dynamically scaling and monitoring. One LB would be a single point of failure in this system, so TravelLog will have multiple of load LBs. The LB should have very low latency in our system, as it is just forwarding requests - if it becomes a bottleneck, we can easily add more.

3.b Forward Tier (Backend for Load Balancer)

The Forward Tier is a ReplicaSet. This tier will look at the request of the incoming packets and forward them to the appropriate service (Location Tracking, WebUI, User Management), which will each have their own elastic policies, sharding policies, etc. Since Surveillance will be accessing this service from a different endpoint than WebUI and Analytics, we can maintain an LRU cache of clientIds for *update* requests, to efficiently route location updates to the location service. Although the LB will be trying to get requests of the same type to the same Server in our ReplicaSet, that is not a guarantee, so the cache may not be useful in all scenarios. If one of the replicas were to fail, they would lose the entire cache, which is just soft state and can be built up again. The primary control triggers to grow/shrink this service will be the throughput and utilization. If certain replicas have very high utilization, we will provision more servers. Likewise, if the throughput of the system is a bottleneck for SLOs, we will provision more servers. Failures for a single server are handled because this service is replicated.

3.c Location Tracking

TravelLog assumes that the *update* operation will be the most used operation, since their will be many locations to track and update. Since location data is constantly being collected, we want this service to be highly available, even in the presence of partitions. As such, we propose using DynamoDB, which is a system that allows for many reads and writes with high availability. A majority of the operations on this system will be puts. When an update request is processed, a notification will be sent to the surveillance base that it has been handled.

People move erratically. They can be traveling for hours and then be idle for hours on end after. To reduce the load on TravelLog, a person's location will only be updated either when the person has traveled x units, every 4 hours during the day hours of that timezone, or when the WebUI/Analytics wants the most recent update on a person. The latter two updates necessitate a tight connection between the Location Services and other Services, which can be implemented with data streams and the ability for other services to ping the Surveillance unit to send location data.

When an Update request comes into the Location Service, it will first enter the service gateway. This gateway will partition the request based on where the request came from. More specifically, we can take the location coordinates that are in the request and map them onto squares of an $N \times N$ size on the earth through Google's S2 service. Based on the id of the square (key), the Location Service uses consistent hashing to map the User's location to a particular Kubernetes StatefulSet, each of which has an instance of the DynamoDB. In order to handle more requests coming from a certain locale, the service will monitor incoming requests and see if particular cells need to have more servers allocated. In this case, we can add more nodes to the StatefulSet. We will add more replicas dynamically as the utilization of the servers for a particular cell becomes too high. On the contrary, if the number requests coming in from a particular location is low, a single StatefulSet can be allocated to hundreds of keys (cells). Each shard is a key-value store, which allows each person to be mapped from a uniqueId to their location history.

When there are no partitions in the service, the service will work normally, logging a person's location data and then transferring data in batches to a replicated Person-NoSQL data store periodically. This allows

the services to access location data for people without putting too much load on the DynamoDB itself. This data store is highly consistent in the presence of partitions, which ensures that the services that call it have the correct data. Thus, updates to this data store will need to be coordinated through a coordination service such as Zookeeper/Chubby in order to not break any notions of consistency. Another reason to use this data store for reads instead of specific people instead of DynamoDB is that it does not have to be sharded at the granularity of a cell. Rather, we propose that it be sharded (database sharding) by a person's city (discussed further in subsection e), since we assume that the Analytics and WebUI will primarily doing requests over municipalities and people in general will not be travelling too much outside of their city (especially during a pandemic). As such, when data is written to this Person-NoSQL data store, we first must get the corresponding shard based on the person's uniqueId and municipality uniqueId (explained in subsection e) and then write to that shard. Even in the case of a person traveling through multiple cells, since the operations are committed by DynamoDB and coordinated by Zookeeper/Chubby the location data of that person can still be reconciled.

In the case of a partition, after the partition heals, the Location Tracking service will look at causal orders of where the person was and timestamps of their location to generate a history. In the case that there are still conflicts, there can be a program that traces where the person was before the partition and after to try and trace the path of the person with whatever information it has. Since the system is highly available, there is the question of how we distinguish between when a person turns their surveillance device off and when the location of the service has not been committed. For the former, we will not have any updates for the user at that time. For the latter, since the Location Tracking service is highly available, if the person is near a base station, we will have updates. These updates will be flagged as tentative and not be written to the Person-NoSQL data store until they are committed. In the case when the WebUI calls $person-interval$ or $place-time$, it can retrieve these tentative paths if needed, as they will be cached. If the system does fail, this cache will be lost, but services can still access these tentative data points directly through DynamoDB, which is hard state.

All the components of the Location Tracking service that have hard state are replicated (Data Store, DynamoDB), which will allow the service to function in the case of failures to a singular server. The soft state caches will be lost, but can be rebuilt as needed.

3.d WebUI

The WebUI service will handle $person-interval$ and $place-time$ requests. When a request first hits the WebUI service, it will go to a ReplicaSet (L7 load balancer) that forwards the request either to either the the $person-interval$ or $place-time$ request handlers.

For the $person-interval$ request, the StatefulSet will use consistent hashing to map the uniqueID of the person to a particular StatefulSet pod. Pods are associated with municipality, so the uniqueId will be mapped to a municipality to be associated with the pod. This pod will then take the uniqueID of the person to get a key for the corresponding Person-NoSQL data shard (this Person-NoSQL database is the same as described in the Location Service). Note: the consistent hashing to map the uniqueID to a shard and the hashing to get a shard in the database follow the same sharding rules but one is consistent hashing and one is database sharding. This Person-NoSQL data shard is associated with the StatefulSet's pod and the Location Service is writing to this. We can shard by uniqueId and municipality by associating a uniqueId to a municipality and updating that mapping as needed (described in User/Person Management Service). If more requests are coming in for a particular municipality, we can add more nodes to that specific municipalities' shards. We can also change our hashing rules for the shards in the database to spread out the data. A query will be done to retrieve the path within the given time interval, including however much can be pulled whilst maintaining a response time under a certain threshold. The query within the time interval will be returned to the WebUI, with the query and the path outside the interval being cached. When another query from the same WebUI client for the same person comes in, the WebUI service handler will likely route the request to the same $person-interval$ ReplicaSet request handler, allowing the cache to be used if needed. Since the system is highly available, there is the question of how we distinguish between when a person turns their surveillance

device off and when the location of the service has not been committed. The former will be reflected in the WebUI by showing that we do not have data for the user for that particular time after a request has been made to the surveillance unit to retrieve a person's location data. Since the Location Tracking service is available even with partitions, it can still accept put and get requests. As such, any put requests that are not committed for the person can be queried from the cache that correspond to DynamoDB and then be displayed as a tentative path on the WebUI. We will only need to access the cache if it the bounds of the Person's movement are not in the Person-NoSQL data store. If the person's location data is not available, then a request will be routed to the Surveillance Request service to get the most recent location data. The WebUI can return the data it has on the person and update it as the most recent location data comes in. If there is no location data even after this request, then the WebUI will display that no data is available for the person.

The $place - time$ request is much more data intensive than the $person - interval$ request. The best design for this request handler is to have location regions map to a specific key that is associated with a shard. This will allow us to shard based on location. As such, when two locations are near each other, they will likely be sent to the same shard to get paths, which may have already done computation on some of that region. As such, when a request is made for $place - time$, ReplicaSets will have a key-value store from regions to a location key, which is then hashed and associated with a shard. We decided not to use slicer for this implementation because we did not want to split a region up into different processing units; rather, we will add nodes to the pod if there is too much load. The region to location key map can be updated as well. The $place - time$ request will use a Location-NoSQL Data store and the DynamoDB and not the Person-NoSQL data store because it is looking for paths in a particular region. However, this query cannot follow the same hashing rules to index into the DynamoDB because the location may be much bigger than an $N \times N$ cell. Thus, when a request comes in for a specific region, it will first check the Location-NoSQL data store cache to see if it has the details of the region (explained more in subsequent paragraph). If not, the $place - time$ request will send the coordinates to a Region Matching service that will look up the region space and get the appropriate cells that correspond to it. Then, those cells will be queried in the DynamoDB with the given time frame. The read-quorum for this operation will be significantly smaller than the write quorum. In addition, the operation will place all paths (tentative and confirmed) regardless, given that this visualization tool is looking at paths in aggregate and does not care about a single person's movement.

Given the computational complexity of this call, TravelLog will run batch jobs overnight and map the paths of regions that are the most commonly used. The entities that use this feature will likely be people government officials within cities, counties, and states. Overnight, the TravelLog will find all paths of the most commonly used regions from the day to a one-month period, and place them in the Location-NoSQL data store. This data store will be replicated and also have a coordination service managing updates, since the batch update jobs will include multiple processes handling writes. In the case of a failure, this Location-NoSQL data store will be fine.

The WebUI request handler, the $place - time$ ReplicaSet, and the $person - interval$ ReplicaSet handle failures due to them being replicated. $place - time$ and $person - interval$ handlers can provision for new nodes for specific shards until they are unable to handle a certain number of requests/second. Within these systems, the Person-NoSQL database the $person - interval$ ReplicaSet will be polling from can be sharded further to keep throughput above a certain threshold.

3.e Analytics, Job Scheduling, and User/Person Management

The heart of the TravelLog system is the Job scheduling system. TravelLog operates under the assumption that the load on the system will be quite heavy during the day, and significantly smaller during the night, since users will be asleep and not updating location, and the number of calls to the WebUI and analytics platforms will be fewer. As such, for the nighttime in given timezones, jobs will run to update the data stores and hashing rule as needed.

This entire process starts with User/Person management, where users are the users of the system and persons are the people being queried. Data processing jobs in this system will record persons, what mu-

nicipalities they are in, and which surveillance bases they were last associated with. This allows services to associate a person with a municipality for consistent hashing, to a surveillance base for the Surveillance Request service, and to a list of *contacts*. If a person were to become infected, that will also be recorded. Users can be government officials that are using TravelLog. (Note: We will have to build a user management service on top of this, which will be have profiles and login information. This doesn't necessarily need to be sharded. We can a service that manages sign up, log in, and session state). Each user can be associated with a specific municipality or set of municipalities. Their queries can also be logged and ordered by the most common people and areas they search for.

With the user/person management system, data processing jobs can be run when the LocationService and WebUI services have significantly less load. These jobs will work on mapping people to municipalities in order to help the $person - interval$ requests map users in order to access the proper shard. Furthemore, it will also look at municipalities that are the most commonly searched by users and get paths for those for the previous day, and update the Location-NoSQL data store. Similarly, it can also update the region to location keys for the $place - time$ request to balance the load across more shards. These updates will likely be done by multiple processes, so will need a coordinator to manage updates to the data stores. Another job will be the find *contacts* job, which will do a search of people by looking into the Person-NoSQL database and looking at paths the person traveled where there was a threshold of $t >= 10$. The paths can then be associated with regions and then we can query the corresponding Location-NoSQL database to find other paths that intersect for $d <= 6$, and then associate those to another person. Since these are grouped by municipality, most of the queries will stay within that shard. A query can move out of the shard if needed. A key feature for governments would be to update who the infected people are and and update who they have been in contact with recursively (ie. if Person A is infected and is with Person B for more than 10 minutes, all of Person B's *contacts* will be updated to be potentially infected).

The *contacts* operation will have to be exact, so there will be no uncommitted/tentative locations - we will only be using committed locations. Given the data analytics work done in the above paragraph to figure out *contacts* for a particular person, all the service will have to do is update *contacts* for the person from the last time that job was run, which is a much more manageable task. If the person's most recent location is not up to date, the analytics service will make a request to the surveillance request service to get the most up to date location. We could use transactions, but the process of a 2-phase commit is cumbersome for a system like this. Rather, when a request comes into a ReplicaSet, it will send the hash of the userId to the particular StatefulSet, which has access to the *contacts* (this can be replicated via Paxos). If the contact list needs to be updated, the service can query the most recent locations for the person from through the Person-NoSQL data store. It can then run a smaller job on the lists of locations this person traveled, as described in the previous paragraph.

3.f Surveillance Request

The surveillance request service will be called by the WebUI or Analytics services to get more up to date data on the person's location, if they need a time that is not in the range, because an idle person's location will not constantly be updated. This service will take in a request from one of the services in its ReplicaSet and then that node will ping the Surveillance Base corresponding to the person. The base will then respond with a specific Surveillance-Update operation, updating the location of the person. If there is an update to the location, this request is then wrapped as an update request, sent to the Location Tracking service and the WebUI or Analaytics services will be updated as needed. If the node does not receive a Surveillance-Update operation request after 3 tries or if there is no location to be updated, it will let either the WebUI or Analytics service know that there is no location update.

4 Requests and Client Replies

When users, access the application, they will be directed to an Edge Point of Presence, which will generate a Virtual IP address to the user based on the client request and where the request originates. This will hit the data center that is closest to the client. In the case that a data center has failed, a different VIP can

be generated corresponding to an alive data center. Once a TravelLog instance is hit, the request will be routed based on the operation and parameters of the operation (location, region size, person, etc).

5 Implementation Ideas

There are several technologies that we have discussed in CS 512 that are applicable to the design. At a high level, we can deploy this system on Kubernetes. In addition, we will use sharding and load balancing, particularly we can take advantage of DynamoDB's sharding and MagLev's L4 load balancing. We also use Chubby to coordinate writes to our data stores, either from the data analytics pipeline or from DynamoDB. To cache results, we can use MemCache as a fast key value store. We decide not to use an implementation of Slicer in our service tiers because we don't see the need to invest resources in auto-sharding to move our state further up the systems, and they generally work fine with a stateless system. In terms of consensus, the Data stores can use paxos for replication and the DynamoDB can use a primary and vector-clocks to handle state updates, as discussed in class. Finally, technologies that we have not discussed that may be useful in TravelLog are Kafka and Apache Spark/Hadoop. The former can be used to send data streams between our different services and data stores and the latter for large scale data processing.

6 Concluding Remarks

There are many optimizations and updates that we can consider in our system. For example, if the service were to completely fail, we could keep location data locally on the surveillance infrastructure or a user's mobile phone, so that when the service comes back up, we have can do one large scale update (the potential load on the system would be very high, though). Furthermore, we could also consider anonymizing location data on the system such that people would place more trust in the government now knowing their whereabouts. If there is a person in contact with other people who are infected, this service can perhaps talk to a managing service that keeps track of a person's uniqueID and contact information to contact them if needed. Regardless, TravelLog is a very interesting and practical system with many potential design choices.