2020 Duke Distributed Systems Midterm: 
A Privacy-aware TravelLog Service

Author: Jiali Xing

April 6, 2020

Contents

1 Introduction .................................................. 1

2 Baseline Model ............................................... 1
   2.1 Data Structure ............................................ 1
   2.2 Operations .............................................. 1
   2.3 Service Architecture and Deployment ............... 2
   2.4 Concerns ................................................. 3

3 Privacy-aware TravelLog Service ......................... 4
   3.1 Data Structure ........................................... 4
   3.2 Sharding ................................................ 4
   3.3 Operations .............................................. 5
   3.4 Analysis ................................................ 5

4 Summary ..................................................... 5
1 Introduction

It is not that difficult to design a TravelLog service that supports the 4 required APIs, as illustrated in my baseline model in section 2. However, it severely jeopardizes citizens’ privacy, in the worst way: it expands mass surveillance. To this end, I spent most of my week to design a system that can not only achieve the same functionality and efficiency but also minimize the personal information stored at our servers. This complicates the our system design, but it is not a bug, it is the core feature of privacy-aware TravelLog Service.

2 Baseline Model

In this section we introduce the services that are borrowed from lecture slides, and how to incorporate them into our system with a baseline TravelLog.

2.1 Data Structure

Let’s start from the data structure of our service: key-value stores. We have 3 key-value stores for baseline model:

1. K: Person P’s UUID  
   V: A list of StartTime of paths.

2. K: (Person P’s UUID, StartTime of path)  
   V: A path, i.e., a sequence of timestamped coordinate triples.

3. K: A timestamped coordinate triples in three-dimensional space  
   V: A list of (Person P’s UUID, StartTime of path).

They are quite self-explanatory: I used StartTime to index each path for a given Person.

2.2 Operations

Each of the four operations can be seen as a transaction, we use RIFL to achieve Linearizability.

1. Update:
   First, If Person P is currently on a path, Get P’s StartTime from KVStore1 and then Append the new timestamped coordinate to KVStore2 with Key (Person P’s UUID, StartTime of path); Otherwise, start a new path in KVStore1.
   Second, Append the Value (Person P’s UUID, StartTime of path) with Key (timestamped coordinate triples) in KVStore3.
2. **Person-interval**: Get the StartTime preceding the specified interval of time from KVStore1, then Get the path within the specified interval of time from KVStore2.

3. **Place-time**: For a given region of space and an interval of time, Get a list of (Person P’s UUID, StartTime of path) from KVStore3. Then Get a list of paths from KVStore2.

4. **Contacts**: First, execute **Person-interval** operation and get a path, then calculate proximity regions for the time interval from the path. Then, Get the list of contacts from KVStore3.

### 2.3 Service Architecture and Deployment

Then let’s describe component services. Blue nodes are services borrowed from lecture notes; Red means stateless, and yellow means stateful.

- **First tier** is a Maglev load balancer at Internet ingress.

- **Second tier** is the Web tier, backend of the first tier. Web tier has per-flow state, serving the **Person-interval** and **Place-time** APIs directly with Web UI, and serving the other two APIs by forwarding the requests to next tier. Note that even though Web servers have soft states, they could be launched from same pod template with an auto-scaling policy, which ensures scalability.

- **Between second and third tier** is Chubby and Slicer. Slicer auto-shards the forwarded requests first by operations, and then by the key for each operation. Specifically, KVStore1 and KVStore2 are sharded by P’s UUID; KVStore3 are sharded by coordinate triples (e.g.,
Chubby provides consistent configure files, locks with mutual exclusion, and coordinates rules, membership and aliveness.

- Third tier is service tier. In this tier we have replication + sharding. Specifically, we should have two groups of shards: ShardGroupA and ShardGroupB.
  ShardGroupA is sharded by P’s UUID (this shard group might be in/stored in users’ devices), and can handle transaction Person-interval independently.
  ShardGroupB is sharded by coordinate triples, need to communicate with ShardGroupA to handle transaction Update, Place-time, and Contacts.

- Fourth tier is Database, where we store the states, snapshots, logs, and locks of our application in hardware. Preferably a database located in Switzerland or Iceland.

All component services have Raft inside to achieve replication, so is each shard within ShardGroupA or ShardGroupB.

2.4 Concerns

This baseline model should satisfy our purpose of providing scalable cloud TravelLog services. Nonetheless, as you can see from subsection 2.1 this design (and potentially other students’ similar design) stores each user’s location and path data in our server (which will in turn be exposed to CIA, advertisers, all governments and all hackers around the world) and sacrifices users’ precious privacy. This concern also severely reduce users’ willing to participate. In order to protect users’ privacy and human rights, I propose the Privacy-aware TravelLog Service in next section.
3 Privacy-aware TravelLog Service

In line with other privacy paper for Location Based Service (LBS), we try to provide end users full control over their path data. The following collaborative privacy-aware key-value store follows the conventional means–anonymization, and it especially resembles [1], where they designed a collaborative trajectory privacy preserving (CTPP) scheme that is “the first user-collaboration technique to provide trajectory privacy for continuous LBSs using non-centralized architectures.” Our key-value store is much more naive than

3.1 Data Structure

We have 3 key-value stores for this extended model:

1. K: Person P’s UUID
   V: A list of StartTime of paths.

2. K: Person P’s UUID
   V: A list of Relays’ UUID (e.g., friends of P).

3. K: hash(Person P’s UUID, StartTime of path)
   V: A path, i.e., a sequence of timestamped coordinate triples.

4. K: hash(Person P’s UUID, StartTime of path)
   V: Encrypted(P’s UUID) with the public key$^1$ of analytics.

5. K: A timestamped coordinate triples in three-dimensional space
   V: A list of hash(Person P’s UUID, StartTime of path).

6. K: A timestamped coordinate triples in three-dimensional space
   V: A list of (Relay’s UUID). In order to track back the owner of paths.

They are quite self-explanatory: I used Relays to obfuscate the true identity of P, while still supporting Contacts transaction, where our servers need to have a minimum amount of knowledge in order to retrieve the list of owners for a given list of paths.

3.2 Sharding

KVStore1, KVStore2, and KVStore3 are sharded by each user P’s UUID, and stored in P’s devices. They can handle transaction Person-interval independently. Admittedly, the replication becomes problematic because a user may not have multiple devices (somewhat unlikely). It is vulnerable to failures, but it is the cost we pay to guarantee strong privacy.

Notice that KVStore3 is also stored in the shard of Relays of P, while KVStore4 is only stored in the shard of Relays of P. Combined with KVStore4 and KVStore5, KVStore3 is used to handle Place-time transaction. Contacts requires all key-value store except KVStore2.

KVStore1 to KVStore4 are not auto-sharded by slicer, they are sharded by users and their choice of Relays. Only KVStore5 and KVStore6 are on our cloud server and are auto-sharded by slicer with geolocation key.

$^1$Using Public Key Traitor Tracing Scheme proposed in [2].
3.3 Operations

Please refer to subsection 2.2 before proceed to these transactions, as we are building on top of subsection 2.2 (expect some reordering of index of key-value stores).

1. **Update:** In addition to the *Update* in subsection 2.2:
   - First, KVStore4 needs to be updated when new path is started.
   - Second, Append the Value (Reply’s UUID) with Key (timestamped coordinate triples) in KVStore6, where the Value (Reply’s UUID) can be found in KVStore2 given the owner of path.

2. **Person-interval:**
   - Same as the *Person-interval* in subsection 2.2. As discussed in subsection 3.2, this transaction is done solely on P’s shard.

3. **Place-time:** For a given region of space and an interval of time, Get a list of hash(Person P’s UUID, StartTime of path) from KVStore5 as well as a list of (Relay’s UUID) from KVStore6. Then retrieve a list of paths from KVStore3 hosted on that list of Relays’ shards.

4. **Contacts:** First, execute *Person-interval* operation and get a path, then calculate proximity regions for the time interval from the path. Then, execute *Place-time* with the calculated proximity regions for each timestamp in the time interval, and Get a list of hash(Person P’s UUID, StartTime of path) from KVStore5 and a list of (Relay’s UUID) from KVStore6. Group the list of hashed path index and send to corresponding lists of Relays by geolocation and timestamps, i.e., fetch the encrypted(P’s UUID) from KVStore4.

3.4 Analysis

This model gives a answer to the key question “Who should have access to what location information under which circumstances” asked in [3]: Each user P fully controls all P’s trajectory data; Our cloud server only stores KVStore5 and KVStore6, which tells nothing about P’s identity (UUID) or P’s path/location (becaused they are hashed); Relays, seemingly important, stores KVStore3 and KVStore4, but knows nothing about P’s identity (UUID) or P’s path/location either.

In short, “who is at that given place” and “where is P”, these two questions can not be infered by anyone except P, unless P grants permission to certain queries. By these means, we guarantee absolute privacy for each users (of course have to trust the hardware).

Note that we rely on users’ devices a lot, but most of the storage space is done in our cloud, which limits the storage in users’ devices to a reasonable level–namely each user only stores her path data and P’s path data, when P nominate her as P’s Relay.

4 Summary

What are the component services (e.g., tiers)? See subsection 2.3.
Outline the functions of each component service. See subsection 2.3.
How does each component service scale? We use auto-scaling policy for Web servers and Slicer for our cloud. See subsection 2.3.

Outline how to partition data and functions within each service (sharding). See subsection 3.2.

What keys are used for sharding? Two keys are used separately for different key-value stores. P’s UUID, and district. See subsection 3.2.

What are the SLOs for each operation? For the 4 transactions, latency should be less than 5 seconds. We value privacy most, users will understand us if we show them this report while they are waiting.

How to handle failures within each component service? For our cloud’s services, we have Raft replications inside. For each user, they need to find their way to backup and replicate.

How to distribute incoming requests across the servers? We have Maglev! See subsection 2.3.

How to map each request to a specific pod(s) of each component service that it touches? We have Slicer! See subsection 2.3.

What controller trigger conditions would you use to grow/shrink each component service? If latency is more than 5 seconds, and bottleneck is our servers (unlikely), we launch new pods.

How to protect the service state against failures? We store data/snapshot and locks in disk. See subsection 2.3. For each user’s shard, they need to find their way to backup and replicate.

What is the hard state that must be protected, and what soft state (e.g., caching) is present? All states are stored, we don’t have cache I think.

What specific techniques and technologies from CPS 512 are applicable for your design, and which are not? Most are applicable!

References

